Dercuano

Dercuano is a self-contained downloadable HTML tarball containing a book's worth of disorganized notes I've made over the last few years. As an alternative option for computer systems incapable of handling a downloadable HTML tarball, I've hacked together an inferior PDF rendering of it as well, which comes to some 4000 pages, formatted for comfortable reading on hand computers.

Buried among the errors, red herrings, and ratholes, there are numerous wonderful insights (perhaps even a few of them original), many fascinating facts about the world (many of which are true, and a few of which are original observations), and a wide variety of inventive ideas about what is possible and what could be done, in particular ideas about how to improve the world with new hardware and software — a few of them workable. I've published little of it previously.

Disclaimer, preface, and warning

Mostly, I made these notes for myself, though with the intention of someday getting most of them into shape for publication, but lacking the discipline imposed by regular publication, that's probably not going to happen. It may not happen anyway. So, fuck it! Here it is, incomplete as it is — I hope you enjoy it!

Beware, this is (almost) all wrong

Much of what is written here is wrong in a variety of ways.

- Some of it is factually wrong (for example, on many occasions I confused the vector space $GF(2)^{32}$ with the very different field $GF(2^{32})$ of degree-32 polynomials over GF(2));
- some of it was factually correct at one point but has since become outdated;
- some of it is okay at a factual level but has led me to incorrect conclusions due to my misunderstanding of the relationships between the facts;
- some of it is just a farrago of incoherent sentence fragments;
- some of it is a collection of atomic facts that are individually coherent and could, in theory, be assembled into a meaningful whole, but so far have not been;
- some of it documents the embarrassingly long path of reasoning by which I eventually argued myself around to a reasonable conclusion which was, in retrospect, obvious from the start; and
- some of it, perhaps most of it, amounts to getting distracted from the most important aspects of an issue by some minor detail.

On the other hand, some of it is correct. Of the correct part, most is unoriginal — sometimes I'm just taking notes on well-established concepts, and sometimes I'm laboriously rediscovering things that are already obvious to others — while some small part is original. Unfortunately, I don't know which part.

Most of these notes are about things I barely understood, or didn't really understand at all, when I wrote the notes. In some cases, I later came to understand them better, but in other cases I've lost even what

understanding I had. Nearly every note is incomplete; of those that are complete, very few have been checked for correctness or revised for readability. So, beware.

Many of the dates are only approximate.

Dercuano is scholarly work in progress, but not a completed scholarly publication

One of the distinguishing features of scholarly publications, as currently understood, is that they are consciously situated with regard to the existing state of knowledge: they are aware of the state of the art; build on its successes (rather than falling victim to known pitfalls); they explicitly describe how they relate to that existing knowledge, declaring which pieces of its foundation are sourced from existing work and what its novel contributions are; and they give credit to existing scholarly work.

By and large, I appreciate these values, and I would like to do work that practices them. Sometimes, in the past, I have. Dercuano is not such a work. It is full of cases where I rediscovered known ideas (sometimes incorrectly) and cases where I think something is true, due to other people's previous work, but I don't remember who demonstrated it, or in many cases, precisely what they demonstrated. In many cases there's existing work in a field that I haven't done the work to understand; often I find that attempting to rederive such work from first principles is the best way for me to understand it, and much of Dercuano consists of such attempts. This is not due to malice, but simply because doing scholarly work properly is a lot of effort, and I haven't finished that work, and in fact I've given up on ever finishing it for most of the notes in Dercuano. From a scholarly perspective, Dercuano is best understood as a collection of unfinished notes on ideas that mostly seem promising and merit further investigation, which could lead to a scholarly publication, rather than a scholarly publication in itself.

The work that leads up to a scholarly publication invariably involves a great deal of information-gathering, experimentation, thinking, revision, and usually discussion before reaching the point of actually representing an advance on the state of the art. When you begin learning about a topic, you have no idea what the state of the art is, what is true or false, or what will work; bit by bit, you find these things out. Sometimes this process is recorded, for example in laboratory notebooks, but it usually remains secret, in part because of all of the embarrassing errors during the process. Preregistration of clinical trials is starting to reduce this secrecy in medicine, but it would be wonderful to see more people doing more of their thinking in the open. Dercuano is an example of what I would like to see more of: scholarly work exposed and done in the open even before reaching the level of a scholarly publication. I am fortunate to have been in the position where I could do this.

Size and public-domain dedication

On 2019-12-28 as I write this, the Dercuano tarball is 3.6 megabytes and contains some 1.2 million words in 882 notes, about 3500 paperback pages' worth of text. The PDF rendering mentioned above uses a page size slightly smaller than standard for improved readability on hand computers.

As far as I'm concerned, everyone is free to redistribute Dercuano, in whole or in part, modified or unmodified, with or without credit; I waive all rights associated with it to the maximum extent possible under applicable law. Where applicable, I abandon its copyright to the public domain. I wrote and published Dercuano in Argentina.

The exception to the above public-domain dedication is the ET Book font family used, licensed under the X11 license (p. 3783). This doesn't impede you from redistributing or modifying Dercuano but does prohibit you from removing the font's copyright notice and license (unless you also remove the font).

(The source repository also contains some other fonts which are used to produce a PDF, but those are not included in the HTML tarball.)

Gitlab

At this writing, there's a replica of this repo on Gitlab.

Notes

2007

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Some notes from playing 20q.net (p. 1246) 2007 to 2009 (22 minutes)
• Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
• I think I understand how to use libart's antialiased rendering API now (p. 409) 2007 to 2009 (10 minutes)
• Barcode receipts (p. 2359) 2007 to 2009 (6 minutes)
• A 2007 overview of matrix barcodes (p. 1094) 2007 to 2009 (2 minutes)
• Bicicleta maps (p. 582) 2007 to 2009 (2 minutes)
• Using bytecode won't make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• C bad (p. 2832) 2007 to 2009 (4 minutes)
• The coolest bug in Ur-Scheme (p. 1007) 2007 to 2009 (2 minutes)
• A stack of coordinate contexts (p. 2987) 2007 to 2009 (9 minutes)
• A cute algorithm for card-image templates (p. 1946) 2007 to 2009 (2 minutes)
• Double ended log structured filesystem (p. 1653) 2007 to 2009 (4 minutes)
• Notes on reading eForth (p. 1398) 2007 to 2009 (9 minutes)
• Notes on reading eForth 1.0 for the 8086 (p. 541) 2007 to 2009 (5 minutes)
• Emacs22 annoyances (p. 3197) 2007 to 2009 (4 minutes)
• A comparison of prices for different forms of energy (p. 3097) 2007 to 2009 (2 minutes)
• Enumerating binary trees and their elements (p. 1445) 2007 to 2009 (4 minutes)
• Erlang musings (p. 2789) 2007 to 2009 (3 minutes)
• Error Reporting is Part of the Programmer's User Interface (p. 2323) 2007 to 2009 (18 minutes)
• Eur-Scheme: a simplified Ur-Scheme (p. 876) 2007 to 2009

(13 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• Free software debugging (p. 136) 2007 to 2009 (2 minutes)
• IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
• Gaim group chat (p. 2677) 2007 to 2009 (3 minutes)
• Interesting features of the GNU assembler Gas (p. 3000) 2007 to 2009 (2 minutes)
• The Gelfand Principle, or how to choose educational examples (p. 1967) 2007 to 2009 (8 minutes)
• Git data (p. 2823) 2007 to 2009 (5 minutes)
• Git learnings (p. 3268) 2007 to 2009 (3 minutes)
• High-risk behavior in context (p. 1485) 2007 to 2009 (5 minutes)
• HTML is terser and more robust than S-expressions (p. 562) 2007 to 2009 (4 minutes)
• Learning low level stuff is not just fun, but also useful (p. 815) 2007 to 2009 (5 minutes)
• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• Designing a Scheme for APL-like array computations, like Lush (p. 661) 2007 to 2009 (4 minutes)
• Microfinance (p. 2875) 2007 to 2009 (6 minutes)
• Developing Win32 programs on Debian (p. 609) 2007 to 2009 (12 minutes)
• Nested inheritance (p. 340) 2007 to 2009 (2 minutes)
• Copyright status of the Oxford English Dictionary: relevant data (p. 82) 2007 to 2009 (3 minutes)
• Polycaprolactone (p. 1813) 2007 to 2009 (3 minutes)
• Notes on running QEMU on Debian Etch (p. 1646) 2007 to 2009 (3 minutes)
• Running your regular desktop in QEMU? (p. 292) 2007 to 2009 (3 minutes)
• Quasiquote patterns (p. 3021) 2007 to 2009 (9 minutes)
• Notes on Raph Levien's "Io" Programming Language (p. 1740) 2007 to 2009 (10 minutes)
• Rich programmers (p. 805) 2007 to 2009 (4 minutes)
• Does SAAS make it harder to ship? I doubt it. (p. 1049) 2007 to 2009 (7 minutes)
• Schimmler parallelism asymptotic gain (p. 294) 2007 to 2009 (1 minute)
• Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
• A survey of small TCP/IP implementations (p. 2616) 2007 to 2009 (4 minutes)
• Food miles imply insignificant energy costs (p. 2187) 2007 to 2009 (4 minutes)
• Maybe Counting Characters in UTF-8 Strings Isn't Fast After All! (p. 2992) 2007 to 2009 (15 minutes)
• Tagged dataflow (p. 405) 2007 to 2009 (2 minutes)
• Why Thunderbird is inadequate for opening a 7-gigabyte mbox (p. 980) 2007 to 2009 (2 minutes)
• The Problem: Writing With One Access Pattern, Reading With Another (p. 3004) 2007 to 2009 (19 minutes)
• Programming paradigms for tiny microcontrollers (p. 2104) 2007 to 2009 (6 minutes)

• The AL programming language, dimensional analysis, and typing: do different dimensions really exist? (p. 731) 2007 to 2009 (2 minutes)

• User-per-group (UPG), umask, and "Permission denied" on shared Git repos via ssh (p. 2481) 2007 to 2009 (4 minutes)
• ML's value restriction and the Modula-3 typing system (p. 1763) 2007 to 2009 (3 minutes)
• Vanagon mail (p. 2032) 2007 to 2009 (3 minutes)
• What's wrong with ../../? (p. 2304) 2007 to 2009 (2 minutes)
• Win32 startup (p. 2439) 2007 to 2009 (2 minutes)
• wood and stone personal digital assistants (p. 3191) 2007 to 2009 (6 minutes)
• Writing math in Unicode with the Compose key (p. 1863) 2007 to 2009 (2 minutes)
• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• Additive smoothing for Markov models (p. 2429) 2007 to 2009 (updated 2019-05-19) (11 minutes)
• OMeta contains Wadler's "Views" (p. 842) 2007 to 2009 (updated 2019-05-20) (13 minutes)
• Improving "science" in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)

2008

• The economics of solar energy (p. 1175) 2008 (27 minutes)
• On hanging out with cranks (p. 1715) 2008-04 (4 minutes)
• Smoky day (p. 3010) 2008-04-19 (4 minutes)
• Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)

2010

• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)

2012

• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• Pensamientos acerca de diseñar un calefón solar (p. 117) 2012-10-15 (2 minutes)
• Más pensamientos acerca de diseñar un calefón solar (p. 1713) 2012-10-15 (5 minutes)
• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Storing dry bulk foods in used Coke bottles (p. 2145) 2012-10-15 (updated 2012-10-21) (5 minutes)
• In what sense is e the optimal branching factor, and what does it mean for menu tree design? (p. 2483) 2012-12-04 (3 minutes)
• Worst-case-logarithmic-time reduction over arbitrary intervals over

arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• How can we take advantage of 16:9 screens for programming? (p. 1982) 2012-12-17 (2 minutes)
• Giving Golang a second look for writing a mailreader (in 2012) (p. 290) 2012-12-17 (updated 2013-05-17) (2 minutes)

2013

• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Food storage (p. 2706) 2013-05-11 (updated 2013-05-17) (54 minutes)
• Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder (p. 527) 2013-05-17 (5 minutes)
• Alastair thesis review (p. 1784) 2013-05-17 (1 minute)
• Charge transfer servo (p. 3017) 2013-05-17 (2 minutes)
• Cheap shit ultrawideband (p. 2776) 2013-05-17 (10 minutes)
• Harvesting energy with a clamp-on transformer (p. 1952) 2013-05-17 (7 minutes)
• Clickable terminal patterns (p. 859) 2013-05-17 (2 minutes)
• Only a constant factor worse (p. 1648) 2013-05-17 (16 minutes)
• Use crit-bit trees as the fundamental string-set data structure (p. 1498) 2013-05-17 (3 minutes)
• Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
• Cycle sort (p. 2344) 2013-05-17 (1 minute)
• How can we usefully cache screen images for incrementalization? (p. 1077) 2013-05-17 (18 minutes)
• Dollar auctions and tournaments in human society (p. 884) 2013-05-17 (7 minutes)
• Evaporation chimney (p. 2147) 2013-05-17 (13 minutes)
• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• You're pretty much fucked if you want to build an oscilloscope on the AVR's ADC (p. 1269) 2013-05-17 (3 minutes)
• Who is inventing the future in 2013? (p. 897) 2013-05-17 (1 minute)

• Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Achieving smooth curves in scanline image generation (p. 1507) 2013-05-17 (1 minute)
• The delta from QEmacs,with only 88 commands, to a usable Emacs, is small (p. 1543) 2013-05-17 (2 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Improvising high-temperature refractory materials for pottery kilns (p. 2701) 2013-05-17 (4 minutes)

• Saturation detector (p. 1588) 2013-05-17 (3 minutes)
• Simple system language (p. 1659) 2013-05-17 (7 minutes)
• Time domain lightning triggering (p. 2534) 2013-05-17 (4 minutes)
• APL with typed indices (p. 3264) 2013-05-17 (11 minutes)
• A unicast phased-array ultrasonic "radio" (p. 3077) 2013-05-17 (4 minutes)
• Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation (p. 1508) 2013-05-17 (updated 2013-05-20) (21 minutes)
• Constructing error-correcting codes using Hadamard transforms (p. 1474) 2013-05-17 (updated 2013-05-20) (22 minutes)
• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• Ultraslow radio for resilient global communication (p. 2071) 2013-05-17 (updated 2013-05-20) (26 minutes)
• The Tinkerer's Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)
• A proposal to support hypertext links in ANSI terminals (p. 486) 2013-05-17 (updated 2019-12-26) (13 minutes)
• Personal notes from 2013-06-06 (p. 2673) 2013-06-06 (updated 2014-04-24) (11 minutes)

2014

• When and why exactly should your code "tell, not ask"? That is, use CPS? (p. 1051) 2014-01-08 (4 minutes)
• Some personal notes from February 2014 (p. 2134) 2014-02-13 (8 minutes)
• Constant-space grep (p. 296) 2014-02-24 (3 minutes)
• Full res globe (p. 1255) 2014-02-24 (1 minute)
• Forth with named stacks (p. 2101) 2014-02-24 (7 minutes)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• Stuff I've posted to kragen-tol over the years about post-HTTP (p. 1815) 2014-02-24 (12 minutes)
• Simple persistent in-memory dictionaries with $\log^2$ lookups and logarithmic insertion (p. 3110) 2014-02-24 (6 minutes)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• A Sunday in 2014 (p. 1089) 2014-02-24 (3 minutes)
• Twingler (p. 2547) 2014-02-24 (7 minutes)
• Compression with second-order diffs (p. 2152) 2014-04-24 (3 minutes)
• Notes on 3-D printing a mechanical LUT (p. 1326) 2014-04-24 (3 minutes)
• Jim Weirich's death and my daily life (p. 829) 2014-04-24 (5 minutes)

• What would a basic income guarantee for Argentina cost? (p. 2409) 2014-04-24 (7 minutes)
• Bike charger (p. 2099) 2014-04-24 (2 minutes)
• Notes from a Buenos Aires blackout, summer 2013–2014 (p. 267) 2014-04-24 (15 minutes)
• Bottle washing (p. 921) 2014-04-24 (7 minutes)
• Cristina Fernández de Kirchner tweets about the attempt to kidnap Assange (p. 985) 2014-04-24 (3 minutes)
• The future of the human energy market (2014) (p. 1846) 2014-04-24 (19 minutes)
• Fixed point (p. 807) 2014-04-24 (1 minute)
• Fukushima leak (p. 2905) 2014-04-24 (6 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• Holographic archival (p. 766) 2014-04-24 (10 minutes)
• Inflatable stool (p. 1047) 2014-04-24 (6 minutes)
• Handling Landsat 8 images in limited RAM with netpbm (p. 1884) 2014-04-24 (4 minutes)
• lattices, powersets, bitstrings, and efficient OLAP (p. 2345) 2014-04-24 (17 minutes)
• Nobody has yet constructed a mechanical universal digital computer (p. 1053) 2014-04-24 (6 minutes)
• Offline datasets (p. 599) 2014-04-24 (15 minutes)
• Precisely how is 3 "optimal" for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• Ostinatto (p. 2780) 2014-04-24 (4 minutes)
• How to use "correct horse battery staple" as an encryption key, including a recommended 4096-word list (p. 2522) 2014-04-24 (44 minutes)
• What might Diamond-Age-like phyles look like in the real 21st century? (p. 1599) 2014-04-24 (9 minutes)
• Planar lookup tables (p. 3105) 2014-04-24 (2 minutes)
• Plato was not particularly democratic; ἄρχειν is not "participating in politics" (p. 1707) 2014-04-24 (5 minutes)
• Polynomial-spline FIR kernels by integrating sparse kernels (p. 1819) 2014-04-24 (12 minutes)
• Randomizing delta-sigma conversion to eliminate aliasing (p. 2834) 2014-04-24 (7 minutes)
• Range literals (p. 1719) 2014-04-24 (6 minutes)
• Simplifying code with concurrent iteration (p. 3293) 2014-04-24 (2 minutes)
• Some speculative thoughts on DSP algorithms (p. 2590) 2014-04-24 (20 minutes)
• In a world with ubiquitous surveillance, what does politics look like? (p. 1615) 2014-04-24 (11 minutes)
• Very composite numbers (p. 2490) 2014-04-24 (4 minutes)
• An extremely simple electromechanical state machine (p. 50) 2014-04-24 (16 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)

• Comparison of the PCO-1810 and PCO-1881 plastic bottlecap standards (p. 3223) 2014-05-25 (updated 2016-07-27) (2 minutes)
• An algebraic approach to 3D geometry (p. 669) 2014-06-03 (updated 2014-06-29) (22 minutes)
• Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
• Division (p. 2181) 2014-06-05 (14 minutes)
• Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
• The Dontmove archival virtual machine (p. 2113) 2014-06-29 (5 minutes)
• He listened to the human intently (p. 2543) 2014-06-29 (4 minutes)

• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• XCHG: An Archival Swap Machine (p. 2997) 2014-06-29 (7 minutes)
• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• Slotted tape with skewed involute roulette bristles as an alternative to hose clamps and possibly screws (p. 395) 2014-07-02 (6 minutes)
• Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
• How to generate unique IDs for IMGUI object persistence? (p. 2035) 2014-09-02 (3 minutes)
• Rendering iterated function systems (IFSes) with interval arithmetic (p. 2433) 2014-09-02 (6 minutes)
• Buenos Aires seen from behind the mirror (p. 809) 2014-09-02 (7 minutes)
• A reactive crawler using Amygdala (p. 2492) 2014-09-02 (updated 2014-09-19) (4 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)

2015

• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• Alphanumerenglish (p. 2882) 2015-04-06 (updated 2016-07-27) (6 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• You can't sort a file whose size is cubic in your RAM size in two passes, only quadratic (p. 2311) 2015-05-28 (5 minutes)
• Ndarray java (p. 2261) 2015-05-28 (1 minute)
• Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
• Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870) 2015-05-28 (updated 2016-06-22) (16 minutes)
• Practically decodable random error correction codes with popcount (p. 606) 2015-07-01 (updated 2015-09-03) (6 minutes)

• Editor buffers (p. 1328) 2015-07-15 (updated 2015-09-03) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Bayesian and Gricean programming (p. 711) 2015-08-20 (3 minutes)

• Cobstrings (p. 1312) 2015-08-21 (updated 2015-08-31) (5 minutes)
• Raggedcolumns (p. 2703) 2015-08-28 (3 minutes)
• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• Alien game challenge (p. 2313) 2015-09-03 (6 minutes)
• Implementing flatMap in terms of call/cc, as in Raph Levien's Io (p. 3248) 2015-09-03 (3 minutes)
• Parsing a conservative approximation of a CFG with a FSM (p. 159) 2015-09-03 (7 minutes)
• A simple content-addressable storage-server protocol (p. 774) 2015-09-03 (3 minutes)
• Desbarrerarme: a UI for speaking to people (p. 186) 2015-09-03 (5 minutes)
• drag-and-drop calculator for touch devices (p. 1045) 2015-09-03 (5 minutes)
• An IMGUI-style drawing API isn't necessarily just immediate-mode graphics (p. 2671) 2015-09-03 (3 minutes)
• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Storing CSV records in minimal memory in Java (p. 524) 2015-09-03 (6 minutes)
• Memoize the stack (p. 2021) 2015-09-03 (5 minutes)
• A one-motor robot (p. 118) 2015-09-03 (13 minutes)
• Optical lever thermometer (p. 2624) 2015-09-03 (1 minute)
• Assigning consistent order IDs (p. 1042) 2015-09-03 (3 minutes)
• Quadratic opacity holograms (p. 3073) 2015-09-03 (7 minutes)
• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• Can you read the lunar lander's plaque from Earth? Or write a new one? (p. 125) 2015-09-03 (9 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Would Synthgramelodia make a good base for livecoding music? (p. 2540) 2015-09-03 (8 minutes)
• Ternary mergesort (p. 2161) 2015-09-03 (2 minutes)
• Waterproofing (p. 429) 2015-09-03 (4 minutes)
• Very fast FIR filtering with time-domain zero stuffing and splines (p. 1146) 2015-09-03 (updated 2015-09-07) (13 minutes)
• Convolution surface plotting (p. 2264) 2015-09-03 (updated 2015-09-13) (2 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Tapered thread (p. 363) 2015-09-03 (updated 2019-06-10) (4 minutes)

• Bitstream dsp (p. 3153) 2015-09-03 (updated 2019-06-23) (3 minutes)
• Convolution with intervals (p. 1044) 2015-09-07 (1 minute)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) 2015-09-11 (updated 2019-12-20) (49 minutes)
• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• Hash feature detection (p. 3294) 2015-09-17 (5 minutes)
• Interactive calculator 0 (p. 1453) 2015-09-17 (2 minutes)
• Interval filters (p. 1282) 2015-09-17 (2 minutes)
• Piano synthesis (p. 1655) 2015-09-17 (updated 2017-07-19) (6 minutes)
• Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Viral wiki (p. 1024) 2015-10-15 (3 minutes)
• Fast geographical maps on Android (p. 455) 2015-10-16 (9 minutes)
• José, the Galician mover (p. 3076) 2015-11-09 (2 minutes)
• Virtual instruments (p. 658) 2015-11-09 (3 minutes)
• Minimal GUI libraries (p. 663) 2015-11-14 (updated 2015-11-15) (5 minutes)
• Hash gossip exchange (p. 1470) 2015-11-19 (4 minutes)
• Logarithmic maintainability and coupling (p. 2341) 2015-11-23 (7 minutes)
• Hot wire saw (p. 3159) 2015-12-28 (updated 2019-06-02) (10 minutes)

2016

• Writing hypertext is still a pain (p. 715) 2016-02-18 (6 minutes)
• Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
• Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
• ¿Qué necesito para relación de pareja? (p. 1298) 2016-03-09 (6 minutes)
• Improving LZ77 compression with a RET bytecode (p. 964) 2016-04-05 (updated 2016-04-06) (3 minutes)
• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• A type-inferred dialect of JS (p. 265) 2016-04-22 (4 minutes)
• Material merits (p. 1496) 2016-05-08 (6 minutes)
• Trees as code (p. 2488) 2016-05-10 (4 minutes)
• Designing an archival virtual machine (p. 3203) 2016-05-12 (6 minutes)
• A variety of code fragments for testing proposed language designs (p. 2560) 2016-05-18 (19 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Linear trees (p. 1811) 2016-05-19 (updated 2016-05-20) (6 minutes)
• Spring energy density (p. 3106) 2016-05-28 (updated 2016-06-06) (13 minutes)
• Wikipedia people (p. 948) 2016-06-01 (6 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)

• Do visually expanding images evoke an orienting response, or the startle response, and what does that mean for ZUIs? (p. 1805) 2016-06-03 (14 minutes)
• Gaussian spline reconstruction (p. 656) 2016-06-05 (updated 2016-06-06) (5 minutes)
• The book written in itself (p. 2400) 2016-06-12 (updated 2016-06-14) (18 minutes)
• Phase-change heat reservoirs for household climate control (p. 2257) 2016-06-14 (updated 2016-06-17) (13 minutes)
• Mechanical buck converter (p. 1876) 2016-06-20 (5 minutes)
• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• Making a logic gate of a single MOSFET (p. 167) 2016-06-28 (5 minutes)
• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• Transmission line computer (p. 509) 2016-07-11 (updated 2019-07-23) (7 minutes)
• How can we build an efficient microcontroller-based amplifier? (p. 2821) 2016-07-13 (5 minutes)
• Jellybean ICs 2016 (p. 817) 2016-07-14 (updated 2019-05-05) (17 minutes)
• Statically bounding runtime (p. 2398) 2016-07-19 (4 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• Append only unique string pool (p. 1797) 2016-07-27 (2 minutes)
• How would you maximize the energy density of a capacitor? (p. 42) 2016-07-27 (5 minutes)
• Electroluminescent matrix (p. 974) 2016-07-27 (2 minutes)
• Interval radiosity (p. 1544) 2016-07-27 (1 minute)
• Matrix memory (p. 503) 2016-07-27 (1 minute)
• Vitruvius could have taken photographs (p. 992) 2016-07-30 (1 minute)
• Arduino radio (p. 169) 2016-07-30 (4 minutes)
• Coinductive keyboard (p. 1893) 2016-07-30 (4 minutes)
• Solar-cell Geiger counters (p. 3241) 2016-07-30 (1 minute)
• Transmission line diode computation (p. 3090) 2016-07-30 (3 minutes)
• Algorithm time capsule (p. 2263) 2016-08-11 (1 minute)
• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)
• Calculations about desalination in Israel (p. 2827) 2016-08-11 (3 minutes)
• The etymology of "tradeoff" (p. 165) 2016-08-11 (5 minutes)
• Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
• Prototyping stuff (p. 176) 2016-08-11 (1 minute)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)

• Opacity holograms (p. 1448) 2016-08-16 (8 minutes)
• Argentine oscilloscope pricing 2016 (p. 1965) 2016-08-16 (4 minutes)
• Phosphorescent laser display (p. 1987) 2016-08-16 (8 minutes)
• Hot oil cutter (p. 3287) 2016-08-16 (updated 2016-08-17) (8 minutes)
• Internal determinism (p. 2803) 2016-08-17 (2 minutes)
• Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) 2016-08-17 (updated 2016-08-30) (24 minutes)
• Flexures (p. 2211) 2016-08-24 (updated 2016-08-26) (6 minutes)
• Affine arithmetic has quadratic convergence when interval arithmetic has linear convergence (p. 1029) 2016-08-24 (updated 2017-01-18) (10 minutes)
• Time series data type (p. 304) 2016-08-26 (3 minutes)
• Starfield servo (p. 1709) 2016-08-30 (updated 2018-11-07) (13 minutes)
• Pulley generator (p. 3148) 2016-09-05 (2 minutes)
• Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
• Robust hashsplitting with sliding Range Minimum Query (p. 733) 2016-09-05 (7 minutes)
• State of the world 2016 (p. 2973) 2016-09-05 (10 minutes)
• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• Spring energy density (p. 1010) 2016-09-05 (updated 2019-04-20) (3 minutes)
• Low-cost green thread locks (p. 252) 2016-09-06 (2 minutes)
• The internet is probably not going to collapse for economic reasons (p. 3194) 2016-09-06 (9 minutes)
• Intro to algorithms (p. 2625) 2016-09-06 (4 minutes)
• Notes on higher-order programming on the JVM (p. 1355) 2016-09-06 (6 minutes)
• Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times? (p. 1027) 2016-09-06 (3 minutes)
• Regenerative fuel air cutting (p. 2622) 2016-09-06 (4 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Sun cutter (p. 56) 2016-09-06 (9 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• Lithium fission energy (p. 3285) 2016-09-06 (updated 2019-09-16) (6 minutes)
• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)
• DHT bulletin board (p. 3001) 2016-09-07 (7 minutes)
• Filling hollow FDM things with other materials (p. 2119) 2016-09-07 (5 minutes)
• An almost-in-place mergesort (p. 740) 2016-09-07 (5 minutes)
• Microprint visor (p. 523) 2016-09-07 (2 minutes)
• ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
• Mic energy harvesting (p. 2583) 2016-09-07 (updated 2016-09-08) (5 minutes)
• Solving the incentive problem for censorship-resistant DHTs (p.

923) 2016-09-07 (updated 2019-05-21) (3 minutes)
• High academic achievement almost certainly depends more on tutoring than group averages by race or sex (p. 113) 2016-09-08 (3 minutes)
• Toward a language for hacking around with natural-language processing algorithms (p. 1681) 2016-09-08 (7 minutes)
• Circuit notation (p. 1161) 2016-09-08 (updated 2017-04-18) (7 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Kinect modeling (p. 164) 2016-09-16 (1 minute)
• Reconstructing a 3-D Lambertian surface from video with a moving light source (p. 3296) 2016-09-16 (1 minute)
• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• DReX and "regular string transformations": would an RPN DSL work well? (p. 453) 2016-09-19 (3 minutes)
• License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162) 2016-09-19 (5 minutes)
• Laser ablation of zinc or pewter for printed circuit boards (p. 2799) 2016-09-19 (4 minutes)
• Simple state machines (p. 760) 2016-09-19 (updated 2016-09-24) (8 minutes)
• Gradient rendering (p. 583) 2016-09-24 (11 minutes)
• Hybrid RAM (p. 2877) 2016-09-24 (5 minutes)
• Immersion plating of copper on iron with blue vitriol (p. 1099) 2016-09-24 (8 minutes)
• Changing the basis to a more expressive one with better affordances (p. 1389) 2016-09-29 (5 minutes)
• Usability of scientific calculators (p. 2379) 2016-09-29 (19 minutes)
• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Marking metal surfaces with arcs (p. 1993) 2016-10-06 (4 minutes)
• Compressed sensing microscope (p. 306) 2016-10-06 (7 minutes)
• Cross current zone melting (p. 1872) 2016-10-06 (1 minute)
• Freeze distillation at 1 Hz (p. 2796) 2016-10-06 (5 minutes)
• 3-D printing glass with continuously varying refractive indices for optics without internal surfaces (p. 1156) 2016-10-06 (3 minutes)
• Hot air ice shaping (p. 864) 2016-10-06 (4 minutes)
• Texture synthesis with spatial-domain particle filters (p. 857) 2016-10-06 (2 minutes)
• Spark particulate sieve (p. 2047) 2016-10-06 (updated 2016-10-11) (7 minutes)
• La vibración del hierro, ¿es de baja frecuencia o qué? (p. 2231) 2016-10-07 (3 minutes)
• Current hardware trends tend toward raytracing (p. 1351) 2016-10-07 (4 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)

• What's the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• Surrealist code (p. 1325) 2016-10-11 (3 minutes)
• Statement from the Confederation of Teachers (p. 725) 2016-10-11 (updated 2016-10-12) (4 minutes)
• Generalizing my RPN calculator to support refactoring (p. 969) 2016-10-17 (12 minutes)
• World War III is starting (?) (p. 346) 2016-10-17 (2 minutes)
• Installing Debian GNU/Linux on an ASUS E403S (p. 336) 2016-10-23 (10 minutes)
• Chintzy depth of field (p. 629) 2016-10-27 (1 minute)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Analogies between spring–mass–dashpot systems, electrical systems, and fluidic systems (p. 1472) 2016-10-30 (4 minutes)
• Academic lineage (p. 2292) 2016-10-30 (updated 2019-11-24) (15 minutes)
• Recuperator heat storage (p. 594) 2016-11-01 (updated 2019-08-21) (4 minutes)
• Selfish conformity (p. 622) 2016-11-15 (5 minutes)
• The problem is not that people are not turning to real journalism anymore (p. 2934) 2016-11-15 (8 minutes)
• Clanking replicators (p. 171) 2016-11-30 (3 minutes)
• Approaches to limiting self-replication (p. 1004) 2016-11-30 (7 minutes)
• Bitsliced operations with a hypercube of shuffle operations (p. 2363) 2016-11-30 (2 minutes)
• Jello printing (p. 2426) 2016-12-14 (8 minutes)
• Nonlinear differential amplification (p. 2949) 2016-12-14 (2 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Passive ultrasound sonar (p. 295) 2016-12-28 (1 minute)
• MiniOS (p. 1091) 2016-12-28 (updated 2017-01-03) (6 minutes)

2017

• The paradoxical complexity of computing the top N (p. 1890) 2017-01-04 (7 minutes)
• The ultimate capacity of human memory if spaced-practice memorization works as advertised (p. 2442) 2017-01-04 (updated 2017-01-08) (14 minutes)
• One-line thoughts that don't merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• Using Aryabhata's pulverizer algorithm to calculate multiplicative inverses in prime Galois fields and other multiplicative groups (p. 2255) 2017-01-06 (updated 2019-07-05) (4 minutes)
• What is the type of lerp? (p. 1985) 2017-01-08 (5 minutes)
• Where did the Rubius comic book come from? (p. 1564) 2017-01-10 (4 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• Similarities between Golang and Rust (p. 1523) 2017-01-11 (updated 2017-01-17) (7 minutes)

• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• Clay fabrication objectives (p. 2111) 2017-01-16 (updated 2017-01-17) (3 minutes)
• Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
• Bubble display (p. 1542) 2017-01-24 (updated 2017-08-03) (1 minute)
• Constant time sets for pixel painting (p. 1484) 2017-02-07 (2 minutes)
• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• My attempt to learn about jellybean electronic components (p. 1974) 2017-02-08 (updated 2019-09-29) (22 minutes)
• Servoing a V-plotter with a webcam? (p. 62) 2017-02-16 (3 minutes)
• Wang tile addition (p. 3201) 2017-02-16 (3 minutes)
• Finite function circuits (p. 2050) 2017-02-16 (updated 2019-05-17) (29 minutes)
• Non-inverting logic (p. 861) 2017-02-18 (updated 2019-07-20) (8 minutes)
• A 7-segment-display font with 68 glyphs (p. 1798) 2017-02-21 (4 minutes)
• Lab power supply (p. 2421) 2017-02-21 (updated 2018-06-18) (17 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• Vibratory powder delivery (p. 1747) 2017-02-25 (2 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
• Passivhaus seasonal thermal store (p. 1723) 2017-03-02 (updated 2017-03-07) (2 minutes)
• Set hashing (p. 2485) 2017-03-09 (9 minutes)
• Burst computation (p. 1500) 2017-03-20 (13 minutes)
• Cartesian product storage (p. 3012) 2017-03-20 (3 minutes)
• Passive dehumidifier (p. 3256) 2017-03-20 (14 minutes)
• Augmenting a slow precise ADC with a sloppy fast high-pass filtered parallel ADC (p. 1469) 2017-03-20 (2 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• The continuous-web press and the continuous press of the World-Wide Web (p. 1134) 2017-03-20 (6 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Loading new firmware on an AVR (p. 88) 2017-03-31 (3 minutes)
• Could you do DDS of comprehensible radio signals with an Arduino? (p. 1829) 2017-03-31 (4 minutes)
• Amnesic hash tables for stochastically LRU memoization (p. 502) 2017-04-03 (1 minute)
• Can you bitbang USB with an ATMega's RC oscillator? (p. 1990) 2017-04-04 (1 minute)
• Ice pants (p. 298) 2017-04-04 (updated 2019-01-22) (17 minutes)
• The Magic Kazoo: a synthesizer you stick in your mouth (p. 1873) 2017-04-04 (updated 2019-05-12) (6 minutes)
• The history of NoSQL and dbm (p. 45) 2017-04-10 (16 minutes)
• Byte-stream pipe and antipipe façade objects for editor buffers (p. 950) 2017-04-10 (3 minutes)

• Incremental persistent binary array sets (p. 1008) 2017-04-10 (4 minutes)
• Studies support authority (p. 1541) 2017-04-10 (2 minutes)
• Disk oscilloscope (p. 713) 2017-04-10 (updated 2017-06-20) (3 minutes)
• TV oscilloscope (p. 1253) 2017-04-10 (updated 2017-06-20) (4 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Secure, self-describing, self-delimiting serialization for Python (p. 243) 2017-04-11 (8 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)
• Solar system scale model (p. 2678) 2017-04-18 (1 minute)
• Quasicard: a hypothetical reimagining of HyperCard and TiddlyWiki (p. 416) 2017-04-18 (updated 2017-06-09) (18 minutes)
• Laser printer oscilloscope (p. 449) 2017-04-18 (updated 2017-06-20) (2 minutes)
• Minimum hardware and software to get a flexible notetaking device running (p. 535) 2017-04-28 (4 minutes)
• String tuple encoding (p. 2419) 2017-04-28 (2 minutes)
• Hipster stack 2017 (p. 2242) 2017-04-28 (updated 2017-05-04) (26 minutes)
• Can a simple nonlinear VCO enable super cheap oscilloscopes? (p. 1357) 2017-05-04 (updated 2017-05-10) (5 minutes)
• Dumb vocoder (p. 1391) 2017-05-10 (2 minutes)
• Generic programming with proofs, specification, refinement, and specialization (p. 958) 2017-05-10 (6 minutes)
• Adding GPIO lines over USB with a Saleae logic analyzer (p. 628) 2017-05-10 (1 minute)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)
• Sparkle wheel display (p. 1738) 2017-05-10 (6 minutes)
• VCR oscilloscope (p. 213) 2017-05-10 (updated 2017-06-20) (2 minutes)
• Flying spot reilluminatable stage (p. 2358) 2017-05-15 (1 minute)
• Relational modeling (p. 1102) 2017-05-17 (updated 2017-06-01) (6 minutes)
• A plotter language of 9-bit bytes (p. 2154) 2017-05-29 (updated 2017-06-01) (11 minutes)
• High-precision control of low-stiffness sytems with bounded-Q resonances (p. 1002) 2017-05-29 (updated 2017-06-01) (4 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Illumination cost (p. 1242) 2017-05-31 (3 minutes)
• Fast sea salt evaporator (p. 1087) 2017-06-01 (3 minutes)
• How cheap can laser-cut boxes be? (p. 2545) 2017-06-01 (2 minutes)
• Karplus–Strong PLLs (p. 2100) 2017-06-09 (1 minute)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• ASCIIbetically homomorphic encodings of general data structures

(p. 3261) 2017-06-15 (2 minutes)
• What's wrong with CoAP (p. 560) 2017-06-15 (3 minutes)
• Lexical gaps (p. 1408) 2017-06-15 (1 minute)
• Micro pubsub (p. 1504) 2017-06-15 (8 minutes)
• A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206) 2017-06-15 (4 minutes)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• Paper editing (p. 1761) 2017-06-15 (3 minutes)
• Web prefetch (p. 3046) 2017-06-15 (1 minute)
• Golomb-coding operands as belt offsets likely won't increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

• Pixel stream (p. 617) 2017-06-15 (updated 2018-10-26) (4 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Database explorer (p. 225) 2017-06-20 (2 minutes)
• Service-oriented email (p. 1302) 2017-06-20 (updated 2017-06-21) (15 minutes)
• CCD oscilloscope (p. 1861) 2017-06-20 (updated 2017-07-04) (7 minutes)
• Compressing a screen update with a tree of dirty bits (p. 303) 2017-06-21 (1 minute)
• A REST interface to a software transactional memory (p. 3014) 2017-06-21 (2 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• A stack of stacks for simple modular electronics (p. 1779) 2017-06-27 (5 minutes)
• CIC-filter fonts (p. 1229) 2017-06-28 (1 minute)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• FM chirp sonar (p. 351) 2017-07-04 (1 minute)
• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)
• Japan can achieve energy autarky via solar energy, but not much before 2027 (p. 2819) 2017-07-12 (4 minutes)
• Binary translation register maps (p. 2080) 2017-07-19 (1 minute)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)
• Compact code cpu (p. 397) 2017-07-19 (3 minutes)
• Complementary goods in home economics (p. 1878) 2017-07-19 (3 minutes)
• Constant current switching capacitor charging (p. 605) 2017-07-19 (1 minute)
• Differential spiral cam (p. 512) 2017-07-19 (9 minutes)
• Distributed computing environment (p. 776) 2017-07-19 (8 minutes)
• Double heap sequence (p. 2521) 2017-07-19 (2 minutes)
• Dyneema (p. 123) 2017-07-19 (2 minutes)
• Ideal language syntax (p. 3260) 2017-07-19 (1 minute)
• The imbalance inherent in copyright systems (p. 2158) 2017-07-19 (2 minutes)
• Parametric polymorphism and columns (p. 1835) 2017-07-19

(2 minutes)
• Piezoelectric engraving (p. 1070) 2017-07-19 (4 minutes)
• Pipe dome (p. 3068) 2017-07-19 (7 minutes)
• Rasterizing polies (p. 2023) 2017-07-19 (3 minutes)
• Replicating education (p. 557) 2017-07-19 (7 minutes)
• Rubber air conditioner (p. 2791) 2017-07-19 (2 minutes)
• Options for bootstrapping a compiler from a tiny compiler using Brainfuck (p. 2958) 2017-07-19 (2 minutes)
• Solar computer 2 (p. 414) 2017-07-19 (3 minutes)
• Term rewriting (p. 3221) 2017-07-19 (3 minutes)
• A tournament to decide which notes to devote attention to polishing (p. 1195) 2017-07-19 (2 minutes)
• Vector instructions (p. 2977) 2017-07-19 (2 minutes)
• JIT-compiling array computation graphs in JS (p. 155) 2017-07-19 (1 minute)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)
• Energy storage in a personal water tower: pretty impractical (p. 2044) 2017-07-19 (2 minutes)
• The Z-machine memory model (p. 2903) 2017-07-19 (4 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Xor 1 to 1 hashing (p. 1595) 2017-07-19 (updated 2017-08-03) (10 minutes)
• Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
• Multiplication with squares (p. 1983) 2017-07-19 (updated 2019-07-09) (5 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Affine arithmetic optimization (p. 2801) 2017-07-19 (updated 2019-09-15) (3 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Interactive bandwidth (p. 779) 2017-08-03 (2 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• Another candidate lightweight frequency tracking algorithm (p. 2069) 2017-08-18 (4 minutes)
• Hammering toolhead (p. 3297) 2017-08-18 (6 minutes)
• What does a futuristic OS look like? (p. 2163) 2017-08-18 (updated 2019-05-05) (6 minutes)
• Notes on scraping the Codex Arundel to preserve it (p. 330) 2017-08-22 (1 minute)
• Deep freeze (p. 1465) 2017-08-22 (updated 2019-01-22) (7 minutes)

• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• A minimal dependency processing system (p. 911) 2017-09-21 (3 minutes)
• Minimal transaction system (p. 2460) 2017-09-21 (5 minutes)
• General purpose layout syntax (p. 3117) 2017-11-10 (updated 2019-09-01) (34 minutes)

• Querying a pile of free-text strings with quasi-Prolog (p. 811) 2017-11-17 (6 minutes)

2018

• Interactive calculator (p. 2771) 2018-04-26 (16 minutes)
• Interactive geometry (p. 508) 2018-04-26 (1 minute)
• Two-thumb quasimodal multitouch interaction techniques (p. 1765) 2018-04-26 (11 minutes)
• Some notes on reverse-engineering The Wizard's Castle (p. 1970) 2018-04-26 (9 minutes)
• Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)
• The tangent of the sum of two angles (p. 2191) 2018-04-27 (1 minute)
• Bench trash power supply (p. 1457) 2018-04-27 (9 minutes)
• Cassette tape capacity (p. 2079) 2018-04-27 (1 minute)
• Constant space flexible data (p. 352) 2018-04-27 (5 minutes)
• A brief note on autonomous cyclic fabrication systems from inorganic raw materials (p. 2965) 2018-04-27 (1 minute)
• Earring computer (p. 2505) 2018-04-27 (1 minute)
• Optimization-based painting software (p. 1158) 2018-04-27 (1 minute)
• Rarely are function-local variables in Forth justified (p. 1055) 2018-04-27 (10 minutes)
• Framed-belt DSP (p. 3095) 2018-04-27 (3 minutes)
• Frustration (p. 3263) 2018-04-27 (2 minutes)
• Gradient overlay (p. 1587) 2018-04-27 (2 minutes)
• A sentence-granularity hypertext editor (p. 2290) 2018-04-27 (4 minutes)
• Incremental recomputation (p. 1184) 2018-04-27 (12 minutes)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Literate programs should include example output, like Jupyter, but Jupyter is imperfect (p. 1308) 2018-04-27 (3 minutes)
• Low-carbohydrate diets are ecologically sustainable (p. 540) 2018-04-27 (2 minutes)
• Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
• Obscurity platform (p. 2991) 2018-04-27 (1 minute)
• Some notes on FullPliant and Pliant (p. 866) 2018-04-27 (9 minutes)
• How inefficient is SNAT hole-punching via random port trials? (p. 1155) 2018-04-27 (2 minutes)
• Compressing REST transactions with per-connection state (p. 1117) 2018-04-27 (11 minutes)
• Urban autarkic network (p. 1026) 2018-04-27 (1 minute)
• Laser cut next step (p. 824) 2018-04-27 (updated 2018-04-30) (7 minutes)
• How can we do online pitch detection? (p. 1869) 2018-04-27 (updated 2018-04-30) (6 minutes)
• Mail reader (p. 3290) 2018-04-27 (updated 2018-06-18) (7 minutes)

• 2017 [Provisional English translation of intercepted transmission] (p. 2192) 2018-04-27 (updated 2018-07-14) (13 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Composing code gobbets with implicit dependencies (p. 2437) 2018-04-27 (updated 2019-05-21) (3 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642) 2018-04-30 (updated 2018-07-05) (29 minutes)
• Notes on a possible household air filter (p. 1961) 2018-05-05 (updated 2018-05-15) (10 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• You can stuff a UHMWPE hammock in your wallet (p. 799) 2018-05-15 (updated 2018-10-28) (11 minutes)
• Gradient descent beyond machine learning (p. 2310) 2018-05-18 (2 minutes)
• Radiant heating (p. 493) 2018-05-20 (3 minutes)
• Home dehumidifier (p. 3131) 2018-05-20 (updated 2019-04-02) (12 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)
• Dutch auction raffle (p. 2474) 2018-06-05 (3 minutes)
• Oscilloscope screens (p. 578) 2018-06-05 (2 minutes)
• UHMWPE clothes could be lightweight and sturdy (p. 3071) 2018-06-05 (3 minutes)
• Clisweep (p. 2705) 2018-06-06 (3 minutes)
• Toward a minimal PEG parsing engine (p. 955) 2018-06-06 (4 minutes)
• Whistle detection (p. 357) 2018-06-06 (updated 2018-12-02) (18 minutes)
• Arduino curve tracer (p. 591) 2018-06-17 (10 minutes)
• Diode logic (p. 272) 2018-06-17 (16 minutes)
• Multitouch livecoding (p. 122) 2018-06-17 (1 minute)
• Resistor assortment (p. 310) 2018-06-17 (4 minutes)
• Snap logic (p. 2580) 2018-06-17 (3 minutes)
• Heating my apartment with a plastic tub of hot water (p. 1310) 2018-06-17 (3 minutes)
• Word stream architecture (p. 2215) 2018-06-17 (13 minutes)
• Transistors vs. Microcontrollers (p. 2918) 2018-06-17 (updated 2018-07-05) (8 minutes)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Turning off the power supply for every sample to reduce noise (p. 239) 2018-06-18 (2 minutes)
• Why is there so much anti-plastic sentiment? Visibility, Arcadian primitivism, conspicuous consumption, and profit. (p. 3270) 2018-06-21 (7 minutes)
• Lithium battery welder (p. 2846) 2018-06-21 (updated 2019-01-22)

(2 minutes)
• The TWI and I²C buses and better alternatives like CAN and RS-485 (p. 1638) 2018-06-28 (updated 2018-07-05) (24 minutes)
• Hacking a buck converter into a class-D amplifier? (p. 2109) 2018-06-30 (4 minutes)
• The Adafruit Feather (p. 2966) 2018-06-30 (1 minute)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Electric hammer (p. 1865) 2018-07-02 (updated 2018-07-05) (14 minutes)
• Capacitors: some notes on tradeoffs (p. 134) 2018-07-05 (5 minutes)
• Barrel safety (p. 1097) 2018-07-14 (3 minutes)
• Flexible text query (p. 2900) 2018-07-14 (4 minutes)
• Hot water bottles (p. 3066) 2018-07-14 (4 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Can you turbocharge the STM32 ADC to build an oscilloscope? (p. 137) 2018-07-14 (5 minutes)
• Agenda hypertext (p. 1836) 2018-07-14 (updated 2018-07-15) (2 minutes)
• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)
• Microlens vibrating lightfield (p. 1219) 2018-07-14 (updated 2018-07-15) (11 minutes)
• Top algorithms (p. 913) 2018-07-29 (4 minutes)
• Comparable counters (p. 2441) 2018-08-16 (1 minute)
• Notes on circuitry for the Nutra seed activator (p. 3099) 2018-08-16 (20 minutes)
• Wang tile font (p. 1463) 2018-08-16 (5 minutes)
• Gradient pixels (p. 2202) 2018-08-16 (updated 2018-10-28) (9 minutes)
• Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
• Notes on QR code capabilities on typical Android hand computers (p. 2972) 2018-09-10 (2 minutes)
• You can't construct optical systems with arbitrary light transfers, but you can do some awesome shit (p. 981) 2018-09-10 (11 minutes)
• Caustic simulation (p. 1454) 2018-09-10 (updated 2018-11-04) (2 minutes)
• Immediate mode productive grammars (p. 898) 2018-09-13 (8 minutes)
• Golang bugs (p. 3087) 2018-09-13 (updated 2018-10-28) (6 minutes)
• Window systems (p. 1335) 2018-10-26 (1 minute)
• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Bit difference array (p. 1748) 2018-10-28 (10 minutes)
• Quintic upsampling of time-series with 1½ multiplies per sample (p. 2844) 2018-10-28 (2 minutes)
• Digital noise generators (p. 1137) 2018-10-28 (2 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Hand drawn font compositing (p. 1810) 2018-10-28 (2 minutes)

• Life octaves (p. 173) 2018-10-28 (2 minutes)
• Three phase oscillating belt (p. 214) 2018-10-28 (4 minutes)
• Time domain analog chaos (p. 2198) 2018-10-28 (4 minutes)
• Electrolytic anodizing, with a small movable electrode (p. 3059) 2018-10-28 (2 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033) 2018-10-28 (updated 2019-05-05) (3 minutes)
• Cheap textures (p. 736) 2018-10-28 (updated 2019-05-05) (5 minutes)
• The details of the GPU in this laptop (p. 2970) 2018-10-29 (2 minutes)
• Dilating letterforms (p. 651) 2018-11-04 (15 minutes)
• Gauzy shit (p. 2985) 2018-11-04 (4 minutes)
• Performance properties of sets of bitwise operations (p. 636) 2018-11-06 (updated 2018-11-07) (16 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)
• Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) 2018-11-09 (3 minutes)
• Recurrent comb cascade (p. 483) 2018-11-09 (updated 2018-11-10) (2 minutes)
• Antialiased line drawing (p. 1803) 2018-11-13 (updated 2019-09-01) (4 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Leconscrip: a family of JS subsets for BubbleOS (p. 2126) 2018-11-23 (2 minutes)
• Tagging parsers (p. 208) 2018-11-23 (updated 2018-12-10) (9 minutes)
• Fast gsave (p. 1593) 2018-11-27 (5 minutes)
• Parallel register file (p. 2952) 2018-11-27 (2 minutes)
• What would a better Unix shell look like? (p. 2831) 2018-11-27 (1 minute)
• The Stretch book is truly alien (p. 1888) 2018-11-27 (6 minutes)
• How small can we make a comfortable subset of JS? (p. 1348) 2018-11-27 (updated 2018-12-02) (3 minutes)
• What can you build out of 256-byte ROMs? (p. 1468) 2018-12-02 (1 minute)
• Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)
• Sparse filters (p. 834) 2018-12-02 (4 minutes)
• Stereographic map app (p. 1281) 2018-12-02 (2 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)
• Constant space lists (p. 3062) 2018-12-10 (10 minutes)
• Turning a delay line into a counter with a FSM (p. 1680) 2018-12-10 (1 minute)
• Minimal imperative language (p. 2175) 2018-12-10 (7 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)

• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• Broadcast ECC with graceful degradation, or avoiding the cliff effect (p. 2045) 2018-12-18 (5 minutes)
• Improving Lua #L with incremental prefix sum in the ∧ monoid (p. 2008) 2018-12-18 (7 minutes)
• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Sample reversal (p. 1353) 2018-12-18 (updated 2019-01-17) (5 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Commentaries on reading Engelbart's "Augmenting Human Intellect" (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)
• Yeso notes (p. 2585) 2018-12-25 (updated 2019-01-01) (11 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)

2019

• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• Supervisor children for fault-tolerant Unix command-line programs (p. 3224) 2019-01-04 (3 minutes)
• Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216) 2019-01-04 (updated 2019-11-12) (26 minutes)
• Median filtering (p. 3155) 2019-01-17 (11 minutes)
• Raid zim (p. 253) 2019-01-17 (updated 2019-02-08) (1 minute)
• Transactional event handlers (p. 139) 2019-01-24 (14 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)
• A review of Wirth's Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• Fast secure pubsub (p. 545) 2019-02-04 (updated 2019-12-03) (2 minutes)
• My notes from learning the Golang standard library (p. 2739) 2019-02-08 (20 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)
• Hardware multiplication with square tables (p. 1886) 2019-02-08 (updated 2019-07-09) (4 minutes)
• Balcony battery (p. 2377) 2019-02-11 (updated 2019-12-06) (6 minutes)

• Friction-cutting plastic (p. 2412) 2019-02-25 (8 minutes)
• Ultralight tunnel personal rapid transit (p. 706) 2019-03-11 (15 minutes)
• Single-point incremental forming of aluminum foil (p. 769) 2019-03-11 (updated 2019-06-10) (14 minutes)
• What are Bitcoin's uses other than sidestepping the law? (p. 2159) 2019-03-11 (updated 2019-07-05) (6 minutes)
• Elastic metamaterials (p. 719) 2019-03-19 (17 minutes)
• India rubber memory (p. 579) 2019-03-19 (4 minutes)
• Tabulating your top event of the month efficiently using RMQ algorithms (p. 619) 2019-03-19 (8 minutes)
• Mayonnaise (p. 1320) 2019-03-19 (updated 2019-06-10) (10 minutes)
• Honk development (p. 1188) 2019-03-21 (2 minutes)
• Weregild (p. 3149) 2019-03-24 (3 minutes)
• Accelerating Euler's Method on linear time-invariant systems by exponentiating matrices (p. 348) 2019-03-24 (updated 2019-04-02) (7 minutes)
• Solving initial-value problems faster and with guaranteed error bounds with affine arithmetic (p. 836) 2019-04-02 (5 minutes)
• Fractal palettes (p. 202) 2019-04-02 (7 minutes)
• Groping toward a high-efficiency speaker driver (p. 1212) 2019-04-02 (15 minutes)
• Sous vide (p. 2921) 2019-04-02 (2 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)
• Paper/foil relays (p. 3273) 2019-04-02 (updated 2019-10-23) (13 minutes)
• Audio video boustrophedon sync (p. 858) 2019-04-03 (2 minutes)
• Macroscopic capacitive DLP (p. 1834) 2019-04-08 (1 minute)
• Caustic business card (p. 255) 2019-04-08 (3 minutes)
• An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)
• Progressive revealment crypto (p. 2068) 2019-04-10 (2 minutes)
• Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias (p. 309) 2019-04-10 (updated 2019-04-16) (2 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Karatsuba (p. 2090) 2019-04-20 (2 minutes)
• A note on meditation (p. 494) 2019-04-20 (1 minute)
• Why Minetest is so addictive (p. 1781) 2019-04-20 (8 minutes)
• Notch scorn (p. 115) 2019-04-20 (5 minutes)
• Plastic cutters (p. 1074) 2019-04-20 (5 minutes)
• Waterfryer (p. 1462) 2019-04-20 (1 minute)
• When should you give up waiting for the bus and just walk? (p. 2280) 2019-04-24 (5 minutes)
• Fencepost cognitive interface errors in text editing (p. 993) 2019-04-24 (24 minutes)
• Hall-effect Wheatstone bridges for impractical steampunk electronic logic gates (p. 2351) 2019-04-24 (2 minutes)
• Plasma glazing (p. 71) 2019-04-24 (1 minute)
• Dercuano stylesheet notes (p. 374) 2019-04-28 (updated 2019-05-09) (72 minutes)
• Dercuano formula display (p. 495) 2019-04-30 (5 minutes)

• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• Dercuano calculation (p. 3135) 2019-05-01 (3 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Why the Cartesian product of fields isn't a field (p. 2660) 2019-05-02 (2 minutes)
• Measuring submicron displacements by pitch bending a slide guitar (p. 905) 2019-05-05 (18 minutes)
• Some musings on applying Fitts's Law to user interface design and data compression (p. 1164) 2019-05-06 (updated 2019-05-09) (27 minutes)
• Designing a drawing editor for well-factored drawings (p. 2115) 2019-05-07 (9 minutes)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• An algebra of textures for interactive composition (p. 1283) 2019-05-08 (4 minutes)
• Free space optical coding gain (p. 1244) 2019-05-08 (updated 2019-05-09) (4 minutes)
• Granite texture (p. 1991) 2019-05-08 (updated 2019-05-09) (5 minutes)
• A phase-change soldering iron (p. 2270) 2019-05-08 (updated 2019-05-09) (14 minutes)
• Dercuano rendering (p. 2300) 2019-05-11 (updated 2019-05-12) (3 minutes)
• A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Dercuano search (p. 1532) 2019-05-16 (2 minutes)
• On influencers (p. 660) 2019-05-16 (3 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)
• First impressions on using the μMath+ calculator program for Android (p. 195) 2019-05-21 (13 minutes)
• Dercuano backlinks (p. 475) 2019-05-22 (7 minutes)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Microsoft Windows uses \ for filenames because OS/8 programs used / for switches (p. 3098) 2019-05-25 (2 minutes)
• Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)
• On the method of finite differences used in Babbage's Difference Engine (p. 827) 2019-05-31 (6 minutes)
• Inductor thermocouple sensing (p. 2037) 2019-06-01 (21 minutes)
• Midpoint method texture mapping (p. 1837) 2019-06-01 (3 minutes)
• Induction kiln (p. 2352) 2019-06-02 (19 minutes)
• Notes on SIP VoIP in 2019 (p. 1064) 2019-06-07 (updated

2019-06-28) (8 minutes)
• How to get 6 volts out of a 7805, and why you shouldn't (p. 537) 2019-06-08 (updated 2019-06-10) (8 minutes)
• Recursive curves (p. 1948) 2019-06-10 (5 minutes)
• Drone cutting (p. 1106) 2019-06-11 (12 minutes)
• Smooth hysteresis (p. 422) 2019-06-11 (13 minutes)
• Computation with strain (p. 2812) 2019-06-13 (17 minutes)
• Foil origami robots (p. 2286) 2019-06-13 (updated 2019-06-14) (10 minutes)
• Observable transaction possibilities (p. 2086) 2019-06-15 (10 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Better be weird (p. 1831) 2019-06-17 (updated 2019-06-24) (9 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Kernel code generation (p. 2302) 2019-07-02 (6 minutes)
• Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) 2019-07-03 (6 minutes)
• Analemma sundial (p. 1955) 2019-07-05 (11 minutes)
• Prolog table outlining (p. 2837) 2019-07-05 (11 minutes)
• Fermat primes (p. 1451) 2019-07-07 (4 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• A failed attempt to make squares cheaper to compute (p. 1622) 2019-07-09 (updated 2019-07-11) (4 minutes)
• Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)
• Some extensions of William Beaty's scratch holograms (p. 2536) 2019-07-11 (9 minutes)
• Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033) 2019-07-16 (updated 2019-07-17) (28 minutes)
• Assembler bootstrapping (p. 2922) 2019-07-18 (updated 2019-12-08) (16 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• Phase relations (p. 2200) 2019-07-23 (updated 2019-07-24) (4 minutes)
• Spiral chinese windlass (p. 2915) 2019-07-23 (updated 2019-07-24) (7 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)
• Energy storage efficiency (p. 1300) 2019-07-30 (4 minutes)
• Cardboard furniture (p. 742) 2019-08-01 (updated 2019-08-11) (15 minutes)
• Needle binder injection printing (p. 1492) 2019-08-05 (12 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29)

(31 minutes)
• Human memorable secret sharing (p. 426) 2019-08-10 (2 minutes)
• Broken computer frustrations (p. 102) 2019-08-11 (2 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• Printed circuits on fired-clay ceramic (p. 960) 2019-08-13 (11 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Harmonic motion chain robot (p. 2197) 2019-08-16 (2 minutes)
• Rubber wheel pinch drive (p. 2912) 2019-08-16 (updated 2019-08-18) (8 minutes)
• The fable of the specialized fox (p. 1076) 2019-08-17 (1 minute)
• Complex linear regression (in the field ℂ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Some notes on the landscape of linear optimization software and applications (p. 1285) 2019-08-21 (updated 2019-08-25) (35 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated 2019-08-28) (37 minutes)
• Text relational query (p. 1223) 2019-08-28 (10 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)
• Multitouch and accelerometer puppeteering (p. 1785) 2019-08-29 (updated 2019-09-01) (12 minutes)
• Query evaluation with interval-annotated trees over sequences (p. 1423) 2019-08-30 (updated 2019-09-03) (30 minutes)
• Autism is overfitting (p. 2692) 2019-08-31 (1 minute)
• Differentiable neighborhood regression (p. 2944) 2019-08-31 (15 minutes)
• Everything is money? (p. 1859) 2019-08-31 (4 minutes)
• Gold leaf trusses (p. 3055) 2019-08-31 (11 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• Cloth structure from shading (p. 84) 2019-09-01 (2 minutes)
• Processing halftoning (p. 915) 2019-09-01 (15 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)
• Debokehfication (p. 473) 2019-09-01 (updated 2019-09-12) (4 minutes)
• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)
• A formal language for defining implicitly parameterized functions (p. 144) 2019-09-05 (updated 2019-09-30) (29 minutes)
• Can artificially-lit vertical farming compete with greenhouses? (p. 2064) 2019-09-08 (12 minutes)
• Bokeh pointcasting (p. 92) 2019-09-08 (updated 2019-09-09)

(16 minutes)
• Hearing aids for disability compensation, protection, and augmentation (p. 764) 2019-09-08 (updated 2019-09-09) (4 minutes)

• Lenticular deflector (p. 2612) 2019-09-08 (updated 2019-09-09) (9 minutes)
• Pythagorean cement pipes for your shower singing (p. 156) 2019-09-08 (updated 2019-09-09) (7 minutes)
• What it means that HTML is "not a programming language", and why the ignorant sometimes think otherwise (p. 1555) 2019-09-09 (updated 2019-10-01) (24 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)
• Nonlinear bounded leaky integrator (p. 3150) 2019-09-11 (8 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)
• Sparse sinc (p. 1880) 2019-09-15 (10 minutes)
• Notes on local file browsing (p. 484) 2019-09-15 (updated 2019-09-28) (4 minutes)
• Capacitive droppers and transformerless power supplies (p. 887) 2019-09-18 (11 minutes)
• B-Tree ropes (p. 2762) 2019-09-24 (updated 2019-09-25) (19 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)
• Audio tablet (p. 2178) 2019-09-28 (7 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• Dercuano grinding (p. 2475) 2019-10-01 (12 minutes)
• Expanded mineral beads (p. 2166) 2019-10-01 (12 minutes)
• Is there an incremental union find algorithm? (p. 1602) 2019-10-01 (8 minutes)
• Notes on Óscar Toledo G.'s bootOS (p. 277) 2019-10-07 (updated 2019-10-08) (28 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Examination of a shitty USB car charger (p. 286) 2019-10-24 (13 minutes)
• Resurrecting duckling hashing (p. 3128) 2019-10-26 (updated 2019-11-10) (8 minutes)
• Comb filtering PWM (p. 40) 2019-10-28 (4 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)
• Hadamard rhythms (p. 831) 2019-11-01 (6 minutes)
• Hot lye granite cutting (p. 581) 2019-11-01 (2 minutes)
• Negative weight undirected graphs (p. 1590) 2019-11-01 (8 minutes)

• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)
• Interval raymarching (p. 1342) 2019-11-02 (updated 2019-11-10) (6 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30)

(29 minutes)
• Shaped hammer face giant pressure (p. 3278) 2019-11-10 (21 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)
• Audio logic analyzer (p. 1801) 2019-11-12 (3 minutes)
• Camera flash extrapolation (p. 2792) 2019-11-12 (6 minutes)
• Derivative based control (p. 2829) 2019-11-12 (6 minutes)
• Applying FM synthesis to natural sounds such as voices (p. 596) 2019-11-12 (2 minutes)
• Kerr snow display (p. 3220) 2019-11-12 (3 minutes)
• Nonconductive relays (p. 3262) 2019-11-12 (3 minutes)
• Approximate optimization (p. 517) 2019-11-13 (3 minutes)
• Arcadian plastics (p. 1340) 2019-11-19 (3 minutes)
• Heliogen (p. 597) 2019-11-19 (6 minutes)
• GPT-2 sets the scene (p. 2978) 2019-11-22 (updated 2019-12-01) (22 minutes)
• Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) 2019-11-24 (2 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Bootstrapping rope bridges and other tensile structures with UHMWPE-bearing drones (p. 2950) 2019-11-25 (5 minutes)
• Underwater energy autonomy (p. 1662) 2019-11-25 (9 minutes)
• Extending Heckballs (p. 3239) 2019-11-26 (6 minutes)
• Oval cam lock (p. 3060) 2019-11-26 (5 minutes)
• Rediscovering successive parabolic interpolation: derivative-free optimization of scalar functions by fitting a parabola (p. 727) 2019-11-26 (updated 2019-11-27) (8 minutes)
• Incremental roller comb forming (p. 3146) 2019-11-27 (4 minutes)
• Byte stream gui applications (p. 128) 2019-11-29 (updated 2019-11-30) (17 minutes)
• Backwards cockcroft walton (p. 2282) 2019-12-01 (2 minutes)
• High temperature semiconductors (p. 2436) 2019-12-01 (2 minutes)

• Transmitting low-power TV signals around your house via RF modulation with an SDR (p. 1950) 2019-12-01 (6 minutes)
• English diphones (p. 2061) 2019-12-03 (5 minutes)
• Bytecode pubsub (p. 205) 2019-12-04 (6 minutes)
• Memory safe virtual machines (p. 975) 2019-12-04 (14 minutes)
• 10tcl ui (p. 1823) 2019-12-06 (17 minutes)
• Introduction to closures (p. 1403) 2019-12-07 (5 minutes)
• Forth assembling (p. 940) 2019-12-08 (updated 2019-12-11) (18 minutes)
• Really simple lab power supply (p. 240) 2019-12-10 (7 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)
• Short words (p. 184) 2019-12-10 (updated 2019-12-11) (4 minutes)
• My very first toddling steps in ARM assembly language (p. 1684) 2019-12-10 (updated 2019-12-13) (46 minutes)
• Berlinite gel (p. 848) 2019-12-14 (updated 2019-12-15) (10 minutes)
• Nomadic furniture optimization (p. 1658) 2019-12-15 (2 minutes)
• Phase change unplugged oven (p. 1433) 2019-12-15 (0 minutes)

• Can you eliminate backpatching? (p. 1769) 2019-12-17 (8 minutes)
• Hypothesis evolution (p. 2143) 2019-12-17 (4 minutes)
• Magic sinewave filter (p. 200) 2019-12-17 (6 minutes)
• Argentine electric bill (p. 2898) 2019-12-18 (3 minutes)
• Sulfuric acid dehydration printing (p. 174) 2019-12-18 (updated 2019-12-19) (3 minutes)
• Sorting in logic (p. 498) 2019-12-28 (2 minutes)

Topics

Comb filtering PWM

Kragen Javier Sitaker, 2019-10-28 (4 minutes)

A phase-correct PWM signal has a lot of its energy at the fundamental PWM frequency and, typically, more at its harmonics --- though *which* harmonics depends on the duty cycle it's producing at the moment. A unity-gain feedforward comb filter has nulls at a fundamental frequency and its harmonics --- if the gain is negative, then it has a null at DC and at all multiples of the fundamental, while if the gain is positive, then it has a null at only odd multiples of the fundamental (and a gain of 2 at DC).

Typically the PWM harmonics are thought of as undesirable noise on the PWM signal. The good news is that, at a constant PWM output level, *all* of the PWM noise is at these harmonics; so, if we could notch them out, PWM could perfectly reproduce our desired output voltage or current.

There are many ways to do this, but in particular I wanted to explore using transmission-line comb filters. A transmission line leading to an open circuit will feed a delayed copy of its input back to that input; this is a unity-gain feedforward comb filter. A 10-meter coaxial transmission line with a typical propagation speed of 0.5*c* will add a signal delayed by about 130 ns to its input.

(I'm a little bit vague on exactly how this needs to be hooked up in a circuit to get the desired effect, in particular for a high-power signal where efficiency is important.)

Suppose that your PWM frequency is 7.5 MHz, carefully locked to this delay. Then the transmission-line comb filter will cancel the first, third, fifth, seventh, and higher odd harmonics from the PWM signal, leaving only the even harmonics.

If this filter can be cascaded with a second similar filter made with a 5-meter coaxial transmission line, that will cancel the second, sixth, tenth, and higher 2*odd harmonics from the PWM signal, leaving only the harmonics divisible by 4.

A third such filter made of 2.5 meters of such line will cancel the remaining harmonics not divisible by 8: 4, 12, 20, 28, and so on.

A fourth such filter made of 1.25 meters will cancel the remaining harmonics not divisible by 16: 8, 24, 40, and so on.

At this point the first 15 harmonics of the PWM signal have been perfectly reactively canceled; the first unfiltered harmonic is the 16th, at 120 MHz.

By adding a delayed copy of the PWM signal to itself four times, we can make a stairstep approximation of the desired signal, with I think any of $3^4 = 243$ different voltage levels; there is a small remaining amount of quantization noise remaining at 16 and more times the PWM carrier frequency.

Ten-meter-long delay lines might sound impractically large, but Horowitz & Hill tell us that old oscilloscopes commonly used a dual-core coax with two signal lines in a double helix inside the outer shield; the signal propagation along the helix was what you would normally expect for signal propagation along a normal coaxial cable center. If this helix were 20 mm in diameter and the coiled wire were 1 mm wide (including insulation), each 2 mm of delay line would

provide 31.4 mm of delay, so 10 m of delay would fit into a length of only 640 mm.

Topics

- Electronics (p. 3430) (138 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Pwm

How would you maximize the energy density of a capacitor?

Kragen Javier Sitaker, 2016-07-27 (5 minutes)

How would you maximize the energy density of a capacitor?

This is the formula for capacitance:

$$C = \varepsilon A / d$$

That A/d is "area over distance". So permittivity $\varepsilon = C d / A$, which means it has units equivalent to farads per meter. Specifically, $\varepsilon_0 \approx 8.85$ pF/m. A farad is a joule per square volt.

The energy content of a capacitor at a given voltage V is $E = CV^2/2$; its maximum energy capacity is $CV^2/2$ where V is its maximum voltage.

Permittivity, then, is one characteristic of a dielectric that determines the energy capacity of the capacitor. Energy stored scales linearly with permittivity; double the permittivity means double the joules per square volt. The other relevant characteristic is its dielectric strength, which is measured in volts per meter; when the field strength reaches its dielectric strength, you get avalanche breakdown and your capacitor is, usually, destroyed. Energy capacity is quadratic in dielectric strength: twice the dielectric strength means that an otherwise-unchanged capacitor can withstand twice the voltage, and thus contain four times the energy.

The maximum voltage is $V = d S$, where S (a variable name I just made up) is the dielectric strength (or "breakdown field" or "breakdown voltage") of the dielectric.

So the energy capacity of a capacitor is proportional to the square of the dielectric strength and of the permittivity of its dielectric. What else does it depend on?

Well, clearly it depends on the area of the plates and the separation between them, since those are also factors in the capacitance. It would seem that you can make your capacitor arbitrarily large in capacitance by making its plates bigger and closer together, and indeed real-world capacitors range over 12 orders of magnitude of capacitance, of which almost all is due to variations in these factors. You eventually run into a limit in capacitance once your dielectric layer is only a few atoms thick, as in double layer capacitors.

But, holding the dielectric *volume* constant, the area of the plates and the separation between them doesn't affect the *energy capacity* at all!

Consider how the energy capacity varies with plate area and plate separation.

As I said before, the maximum energy capacity is $E = CV^2/2$, where $V = d s$. So this works out to

$$E = d^2 S^2 \varepsilon A / 2d = d A S^2 \varepsilon / 2$$

Now d A is just the volume of the dielectric! So for a given volume of dielectric, it doesn't matter whether you have a high-voltage, low-capacitance capacitor or a low-voltage, high-capacitance one; the energy stored is the same. And $S^2 \varepsilon / 2$ gives you the energy per unit volume!

That means that, for example, air has a dielectric energy capacity of

about 40 joules per cubic meter (½(3 MV/m)² · ε_0/2), while glass, with a relative permittivity of 4.7 and dielectric strength of about 10 MV/m, can hold ½(10 MV/m)² · 4.7 · ε_0 ≈ 2000 joules per cubic meter. Tantalum pentoxide has not only a remarkable relative permittivity of about 25, but also a dielectric strength of 400 MV/m, giving about 18 MJ/m³. (And that's half of why tantalum capacitors explode when you overload them. The other half is that most tantalum capacitors are made of an explosive mixture of manganese dioxide and tantalum.)

Diamond's dielectric strength is supposedly 20 MV/cm, or 2000 MV/m, and so even at its lower relative permittivity of 5.7, diamond-dielectric capacitors should have a higher energy density of 100 MJ/m³. Diamond dielectrics are not in current use, perhaps because as a semiconductor, even slight impurities give it unacceptably high conductance. (Or perhaps even without them? I'm not sure.)

More typical dielectrics include PZT (relative permittivity 300–5000, dielectric strength 10–25 MV/m), mica (relative permittivity 3-6, dielectric strength 118 MV/m), and fused quartz (relative permittivity 3.75, dielectric strength 30 MV/m).

Summarizing:

material	relative permittivity	dielectric strength (MV/m)	energy density (J/ℓ)
air	1	3	0.04
glass	4.7	10	2
fused quartz	3.75	30	15
PZT (low)	300	10	130
mica	3-6	118	250
PZT (high)	5000	25	14000
tantalum oxide	25	400	18000
diamond	5.7	2000	100 000

PZT is a little weird because its breakdown behavior is very time-dependent, and in a temperature-dependent way. Water, which I didn't include in the table, is even more so; it has a very promising relative permittivity of around 80, but its breakdown voltage goes to zero as time goes to infinity — eventually, apparently, a streamer will form through the water and discharge your capacitor. There are nevertheless water-dielectric capacitors in use for special purposes such as particle accelerators.

It's instructive to compare the capacitive energy densities above with more typical bulk energy storage systems: gasoline is 36 MJ/ℓ, lithium batteries are 4.3 MJ/ℓ, and lead-acid batteries are 340 kJ/ℓ. The above table suggests that diamond-dielectric capacitors might be in the ballpark of lead-acid batteries for energy storage.

The bottom line is that, among the solid capacitors commercially available today, tantalum capacitors offer an order of magnitude higher energy density for long-term energy storage.

Topics

The history of NoSQL and dbm

Kragen Javier Sitaker, 2017-04-10 (16 minutes)

The current fashion of "NoSQL" key-value stores reminds me that Unix has shipped with a NoSQL key-value store since Seventh Edition Unix in 1979, written by Ken Thompson and called dbm. Dbm files are on-disk hash tables mapping strings to strings, and they are used by many Unix programs — Sendmail and Postfix, for example, support storing arbitrary tables related to mail delivery in dbm files, and Apache supports using dbm files for many purposes. They're supported by a dbm library, which generally only works properly if at most one program has them open at a time. You don't normally connect to a "dbm server", but rather open the file and lock it.

History

The 1979 dbm interface looks like this, in modern ANSI C parlance:

```
typedef struct { char *dptr; int dsize; } datum;
int   dbminit  (const char *name);
int   store    (datum key, datum content);
datum fetch    (datum key);
int   delete   (datum key);
datum firstkey ();
datum nextkey  (datum key);
int   dbmclose ();
```

(Adapted from the GDBM docs.)

This is very simple, and it's easy to figure out how to use it, aside from an allocation issue I mention below, but you can perhaps see some problems right there in the interface — not only do `store()` and `fetch()` and the like not have a namespace prefix, making it easy to have collisions with your functions, but also they don't have a parameter to tell you which dbm file to access! That means you can only have one dbm file open at a time in a single process, but on the PDP-11 that V7 Unix ran on, the process's entire address space was only 64KiB, so you couldn't do too much in one process anyway.

The dbm file uses a scheme called "extendible hashing" to allow the on-disk hash table to grow smoothly as data is added to the file, though in a way that requires the underlying filesystem (and your backup programs!) to handle sparse files efficiently.

The original dbm was more or less replaced with Berkeley ndbm in 1986, which solved those interface problems, but, as I understand it, still used the same limited disk file format. Other more or less enhanced clones included sdbm (1987), GDBM (1990–2002), Berkeley DB (1991 to present), QDBM (2000, QDBM's successor Tokyo Cabinet, TDB, its variant ntdb, tdbm, Larry McVoy's memory-mapped MDBM (significantly enhanced this millennium by Yahoo), and a pure-Python implementation called dumbdbm.

In the original dbm interface, it isn't obvious from the API where the buffer space for the fetched data comes from, and in particular

when it will be reused — in GDBM, at least, it's `malloc`ed, and the caller must free it, even if they only cared about testing whether the key is present or not. But that would be an unlikely thing for 1979 Unix to do (it was very shy about dynamic allocation) and GDBM's compatibility ndbm interface frees it for you on the next call, whether you like it or not. I infer that probably dbm and ndbm returned you a pointer to a static buffer. And, indeed, we can see the fetch function in 1979 dbm doing exactly that; `pagbuf` is a static buffer defined (!) in `dbm.h`.

cdb

One dbm replacement is particularly interesting, because instead of being an enhanced version of dbm, it was a deliberately more limited version.

When Daniel Bernstein decided to write a secure replacement for Sendmail in 1995, called qmail, he was faced with the problem that existing C libraries were full of unreliabilities, poor performance, and security holes, just like Sendmail itself. He solved this problem by writing replacements for all of the standard C library functionality that he needed, from scratch, without any functionality he did not need, and without bugs. One of the things he needed was a rough equivalent of dbm, but he did not need dbm's ability to incrementally update an existing database. So the qmail equivalent of dbm is called "cdb", "constant database", and it consists of 329 lines of C. The read interface consists of these two functions:

```
int cdb_seek(int fd, char *key, unsigned int len, uint32 *dlen);
int cdb_bread(int fd, char *buf, int len);
```

`cdb_seek` returns 1 if `key` of length `len` is present in the file open on file descriptor `fd`, 0 if not, and -1 on I/O error, storing the length at `dlen`; `cdb_bread` then reads the corresponding value, if desired, into `buf`, returning -1 on error, including truncated files, or 0 on success.

Rather than using the extendible-hashing algorithms used by dbm, the cdb file is always divided into 256 hash buckets, described by a 2KiB table at the beginning of the file, whose format limits it to 4GiB. And the code to generate the file builds a hash table with separate chaining in memory, then writes it to the file once insertion is complete.

cdb is less featureful and presumably less performant than other variants of dbm, but because it likely has no bugs and is only about 2 kilobytes of executable code, it may be preferable at times.

Language integration

Perl 4 had native support for dbm files, and a lot of websites that graduated from storing their data in static text files started to use dbm files instead. Perl 4 was the first widely-used garbage-collected language; most of the shift from static websites to web applications in 1994 and 1995 was implemented in Perl, and many sites still didn't have Perl 5 installed.

Python, too, has shipped with support for dbm files for a very long time.

Why not just use a filesystem directory?

You might reasonably ask why people used dbm files, which after all merely map sequences of bytes to sequences of bytes, when the filesystem already performs this function. The reason is that, at the time, most Unix filesystems still used sequential search in filesystem directories for filenames, and as a result, directories with more than a few dozen files in them started to get slow. If you are going to maintain a table and then sequentially search it, you can just use a text file for that. (And Unix does, all over the place.) Also, in most Unix filesystems, files take up at least 256 bytes or so.

The Pick operating system and the ReiserFS filesystem for Linux were based on making the normal filesystem apt for this kind of purpose, rather than trying to build the facilities in userspace. Various forms of Pick are still around, but ReiserFS ran into performance limitations in the Linux system call interface, then lost its influence after Hans Reiser murdered his wife in 2007 and was not able to effectively lead the project from prison.

The filesystem interface requires at least three system calls to get the contents of a file: open(), read(), and (in the steady state, anyway) close(). Even today, Linux system calls require on the order of a microsecond, about 300ns on my machine, so about 1 µs for the three put together, which will limit you to about 300k such file reads per second on a single thread. By comparison, MDBM can manage about 450 ns per random read. One of the last controversial projects of Reiser's company was a system for batching up a whole sequence of Reiser4 operations into a single system call.

Why not SQL?

So we used dbm files for all kinds of things, including things where a relational database would worked a lot better. You might wonder why we didn't just use relational databases.

The problem is, there were no decent free software SQL databases. University INGRES was, as I recall, available, but it had its own query language ("QUEL") and didn't support SQL; its development had ended in 1985, though Wikipedia tells me that Sybase and Microsoft SQL Server were developed from that codebase. At Berkeley they were developing Postgres, which was licensed to Illustra about 1994 as an "object-relational database," and eventually sold to INFORMIX, but Postgres was slow and unreliable, and it didn't support SQL either, yet — its query language was a thing called "POSTQUEL". SQLite and MySQL and MariaDB and even Gadfly didn't exist yet.

A lot of people did build web sites and things with SQL database backends, including well-known companies like Amazon and eBay and lesser-known companies like ArsDigita, which was in many ways the prototype for Google. But they had to license proprietary databases to do it.

There was additionally the problem that relational databases were very heavyweight, like Cassandra today — you couldn't run them at all on a low-end machine, and you universally had to run them in a separate process and connect your app to them via IPC — sockets or whatever. (This was partly ideological. As Stonebraker said in the original Postgres — excuse me, POSTGRES — paper, "DBMS code must run as a sparate process from the application programs that access the database in order to provide data protection.") You

probably had to have a separate machine to run the database server on, a workstation or maybe a tower-case PC. This kind of nonsense meant that there was no possible way to use . This may be hard to imagine now that most cellphones have dozens of SQL databases in them (mostly SQLite) but consider that, even today, starting up `sqlite3` linked with glibc and opening a database, you're already using 27 megs of virtual memory:

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
user     26485  0.0  0.0  26916  3728 pts/3    T    01:47   0:00 sqlite3 hello.d
```

Until the DRAM price bubble burst in 1995 or 1996, DRAM cost US$40 per megabyte for years due to collusion in a cartel of DRAM manufacturers, so that was about US$1000 worth of memory.

(This level of bloat is probably mostly glibc's fault. The sqlite3 binary and library are under a megabyte.)

In 1994, David Hughes in Australia sparked a revolution by writing a simple SQL database he called "mSQL" (for "mini SQL") and releasing it with source code, but not as open source — it was "free for noncommercial use" only. In 1995, he founded a company he called Hughes Technologies to commercialize it. But that's another story, so I am going to return to talking about dbm.

Crash safety

In the environment Unix grew up in, a power loss was a catastrophic event, similar to a disk head crash, a datacenter fire, or a memory corruption bug in the kernel. Some other operating systems had crashproof filesystems which were designed not to lose data in the event of power loss, but Unix did not, and indeed this was one of the desiderata in the original plans for the GNU system — GNU would be better than Unix because its filesystem would have versioning and be crashproof.

Corrupting your filesystem and losing files on power loss was still common in Linux up until about 2004 or 2005, at which point ext3fs and other journaled filesystems put an end to that problem, for the most part.

Nowadays, by contrast, it is very common for Unix machines to lose power — their batteries may run out, or you may drop them on the sidewalk and joggle the battery out of touch with the contacts. By contrast, memory corruption bugs in the kernel and cellphone fires are vanishingly rare, and SSDs have no moving parts and therefore don't have head crashes.

However, if there are consistency constraints inside some file that is being written to in random places, it's possible for an inconsistent, acausal snapshot of that file to be what survives. Unix provides an `fsync()` system call to limit the possibilities for such inconsistencies — it doesn't return until all the data for the file in question is safely saved on disk. This guarantees two relevant properties:

- If `fsync()` returns and the program takes some action afterwards, such as displaying a user interface message or sending a packet over the network, then if we observe this action, then we know that the data written before `fsync()` will not be lost.
- Either all data written before `fsync()` is preserved, or no data written

after `fsync()` is preserved, or both. It is never the case that some data written after `fsync()` is preserved, while some data written before `fsync()` is lost. That is, `fsync()` serves as a "write fence".

Property #2 is necessary to guarantee the atomicity and consistency properties of transactions; property #1 is necessary to guarantee the durability property of transactions.

However, because property #1 is very expensive to provide, especially on spinning-rust disks, it is very common that programs do not bother. There was a great deal of controversy a few years back over MongoDB doing this in order to get better performance numbers, but you will of course see that the MDBM numbers I cited earlier are taken under the same conditions, and the GDBM manual explains:

…the following may be added added to `read_write` by bitwise or: `GDBM_SYNC`, which causes all database operations to be synchronized to the disk, and `GDBM_NOLOCK`, which prevents the library from performing any locking on the database file. The option `GDBM_FAST` is now obsolete, since gdbm defaults to no-sync mode.

…

Unless your database was opened with the `GDBM_SYNC` flag, gdbm does not wait for writes to be flushed to the disk before continuing. The following routine can be used to guarantee that the database is physically written to the disk file.

```
gdbm_sync ( dbf )
```

It will not return until the disk file state is syncronized [sic] with the in-memory state of the database.

Unfortunately, because omitting `fsync()` endangers not only property #1 but also property #2, it is entirely possible for a GDBM file to be corrupt after a power outage. In the case of GDBM, although I haven't verified this, I think this means that some previously stored data that wasn't being modified could be irretrievable, but not necessarily all of it.

Worse, making this work properly typically involves inserting an `fsync()` as a write-fence in the *middle* of a sequence of write operations. Doing an `fsync()` at the end can ensure that, if the execution gets that far, the data stored is not lost; but it cannot ensure that a power failure at some other point does not leave the disk file in an inconsistent state.

Topics

An extremely simple electromechanical state machine

Kragen Javier Sitaker, 2014-04-24 (16 minutes)

As I was walking out my door earlier tonight to get an empanada, I noticed that the keys had left a mark by brushing against the paint on the door, and I thought, wow, I can make pretty intricate patterns in brass entirely by accident this way. Wouldn't it be awesome if we could somehow just *draw* a computer as a two-dimensional pattern on paper and have it work?

And it occurred to me that it *is* actually practical to simply draw a state transition table and have a simple electromechanical machine implement the state machine you specified. Here's the idea.

The idea (here)

Suppose you want to implement an arbitrary finite-state machine with a minimum number of "active" electronic elements (things like vacuum tubes or transistors). Here's an interesting approach, sort of inspired by Shamir's TWINKLE.

You maintain your current N-bit state in N flip-flops with complementary Q and /Q outputs; typically each flip-flop requires one or two vacuum tubes, or a locking relay, or two to six transistors, depending on your logic family and other details. You hook up the Q and /Q outputs to lamps: these days, you would surely use laser diodes or other LEDs, but in the 1940s timeframe where this scenario makes the most sense, you'd have to use something else, maybe neon lamps. The lamps give you a matrix of N rows of two lamps, one of which is on at any given time.

A reasonable value for N here might be 3 to 20.

Now you put these lamps close to a strip of paper. The lamps reflect diffusely off the paper and illuminate the environment. If there are dark blots on the paper under an illuminated lamp, that will diminish the overall illumination reflected from the lamps.

With a lens per lamp --- not necessarily a very good lens --- you can do this trick with the lamps some distance from the paper. The idea is that each lamp just illuminates a corresponding spot on the paper.

Suppose the paper is a vertical strip, divided into rows of two squares, one square for each lamp; and just as one of the two lamps in each row is lit, one of the two squares in each row is light, while the other one is dark. Now the illumination reflected from each row is the XOR or XNOR of a bit of our state and the corresponding bit on the paper. If we slide the paper vertically, the illumination will fluctuate with time as the Hamming distance between the current state and the selected subset of the bit pattern on the paper varies.

Now, instead of sliding bits of paper around, we can wrap this strip of paper around a rotating drum. As the drum rotates, the number of matching bits on these two tracks of the drum will vary, and so will the reflected light. It will reach a minimum when every lamp is illuminating a black area.

How small is that minimum? Asphalt has a visible-light albedo of

0.04; I think carbon black is a little darker than that. Polished aluminum has a reflectance of about 0.95, although I think it gets a little worse when unpolished. But basically we can have a contrast ratio of about 20 or 25 between black and white on paper without doing anything exotic. That means that if you have 5 bits and all 5 of them match, you'll have about $0.04 \times 5 = 0.2$ of the light from one bit; while if only 4 match, you'll have $0.04 \times 4 + 0.95 \times 1 = 1.11$ of the light from one bit, about 4½ times as much. So this is an easily detectable event: the ratio between a perfect match and a near match is about a factor of 4. You should be able to detect this event with a photodiode or even an electric eye.

Suppose that an adjacent track of the drum carries another pattern of bright and dark squares, containing desired new states for your flip-flops. If you have more lamps and photodetectors close to it, one per bit, they can read the pattern without leaking much light into the environment.

Now, as the drum turns, the apparatus searches for a matching state on the drum; when the reflected light level falls to a minimum, it knows it has found it, and it loads the specified new state into the flip-flops from the other track. It's probably desirable for the "pattern" bits being matched on the track to be somewhat short, so that the new-state bits are well centered under their detectors by the time the detector fires, as follows:

```
 Q /Q  newstate
--     ##
   --
   --  ##
--     ##
```

You probably do need a mechanism that prevents a misaligned match: say, N-1 bits of one pattern and 1 bit of the following one. One way is to have an extra framing "bit" that is always illuminated on both sides, marking the paper dark on both tracks:

```
 Q /Q  newstate
--     ##
   --
   --  ##
--     ##
-----
```

Alternatively, you could just space the rows unevenly, or even leave a fractional-row-sized space between patterns.

The (N+1)×3 arrangement described above is just one of many possibilities. You could just as well put all the bits on one track, and the lamps in a line; or you could put each bit in a separate track, producing 3N tracks and one bit-height per pattern, which is probably the highest-performance option; here we have three transitions:

```
Q1 /Q1 Q2 /Q2 Q3 /Q3 Q4 /Q4 Q1' Q2' Q3' Q4'
--        --  --     --     ##  ##  ##
--        --  --         -- ##  ##  ##  ##
--        --     --      -- ##  ##
```

Simple latches, without edge-triggering, probably suffice for the flip-flops, because each transition will be either to the same state or to a different state. If it's to the same state, it's idempotent, so it's harmless to execute it continuously until the marks pass; and if it's to a different state, then it will execute until the lamps display enough of the new state that the light level rises above the level that triggers the state-transitioning logic. You just have to be careful that the transition is sufficiently well-established that all the bits have changed as they should, and none is left in a metastable state.

Further variations

• The natural approach to arranging the patterns on the drum is just to put them in numerical sequence, evenly spaced around the drum, so that you execute about two transitions per revolution. But you can also repeat them so that some transitions have the opportunity to execute many times, and so that transitions that happen in sequence are placed in sequence, and of course you can repeat the entire sequence of transitions. Furthermore, you can use the drum position as an additional, less flexible state variable, so that the entire state machine cycles through different transition graphs.
• Of course, one or more of the bits used to select the pattern can actually be an input, rather than a stored bit. You probably want to make sure the input doesn't change while you're using it to choose your next state; you can sample-and-hold it, or use an actual edge-triggered flip-flop, if it can change state arbitrarily.
• It's possible to encode "don't care" bits by coloring both sides of the pattern bit, potentially reducing the number of patterns dramatically.
• If you have both "new Qi" and "new /Qi" bits on the paper, you may be able to simplify the transitioning logic to simply illuminating them when the reflected "pattern" light falls below the threshold. That is, the nonlinear element that switches between retaining the existing state and adopting a new one can be simply the power supply of the lamps illuminating these bits.
• If you have both "new Qi" and "new /Qi" bits as described above, you have the additional option of *not* switching some bits, allowing them to retain a memory independent of the state transition being executed; this can allow a dramatic reduction in the number of transitions by, among other things, allowing you to preserve "don't care" bits, and allowing different subsets of the state independently.
• Your state variables don't need to be binary --- they can have more than two stable states. Indeed, ternary variables will give you slightly more states for patterns of the same size, and may be more convenient to work with; and quaternary variables (one-hot out of four) will give you patterns of the same size, but half the lamps illuminated, and so they may be easier to do the threshold logic with.
• You can use a tape or disc rather than a drum for a more compact machine.
• For a better contrast ratio and fewer materials, you can use punched or drilled holes rather than marks.
• You could use the state where all of the bits are *reflected* (or transmitted) instead of all *absorbed* as the trigger state, but I figured that was probably harder, since the difference between

almost-all-bits-reflected and all-bits-reflected will be, say, 10% to 33%, which will be harder to discriminate reliably than the 350% or more between almost-all-bits-absorbed and all-bits-absorbed.
• Rather than using diffuse reflection and absorption, you could use specular reflection of collimated light and diffuse reflection. For example, if you're using something like a hard disk platter, a scratched or etched spot in its surface will scatter light at random, while an untouched spot will produce a collimated reflected beam from a collimated incident beam.
• If you have an additional hardware budget, you could of course replace the pattern tracks and the light detector with a counter circuit and digital comparator, leaving only the new-state tracks and a timing mark for restarting the counter.
• If you do the whole thing in a hard vacuum, you could use electron beams instead of light, which might make it faster, depending on your light source. You'd have to use an unlubricated bearing to spin the drum if you were doing this on Earth.
• Of course you can connect some of your state bits to control some other device, such as a memory, which would then provide some of the input bits.

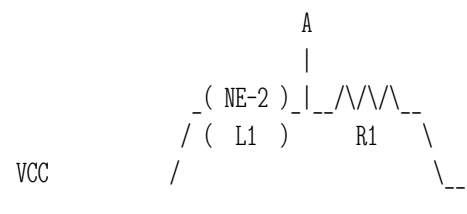
What this gets you

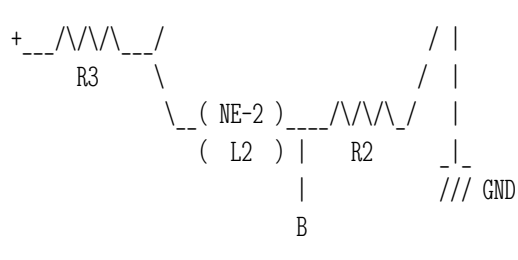
If N=4 plus one bit of input, this should give you an arbitrary sixteen-state state machine for the cost of somewhere around five to forty active elements, depending on what they are; and its speed should be limited to something like one transition per sixteen times its lamps' response time. That is, if your indicator lamps respond in 100ns, you should be able to do an arbitrary transition in 1600ns. This is dependent on actually testing a pattern every 100ns; at 5400rpm and a radius of 4cm, your patterns will need to be about 2 microns across, which means your lamps' illumination spots will need to be about that size too, and you might need to make sure they were aligned to within that precision, too. 200 microns is probably more practical if you want to be able to construct the thing without resorting to a microscope, and you can probably spin a 40-cm-radius drum at 5400rpm, practically speaking, giving you a microsecond per candidate phase transition.

The great disadvantage of this approach is that the state machine's real-time performance is fairly poor, so it can't interface directly with things like delay lines unless they're very slow indeed.

A potential 1920s realization using only neon glow lamps

I think that if you accept some further slowdown, although not quite to the level of magnetic relay logic, you can build it with a couple of 1920s neon glow lamps for each flip-flop. I haven't really played with the things, but I think that this topology gives you a flip-flop:





If "VCC" is above the striking voltage of the lamps (say 110 volts), then either L1 or L2 will ionize and start to conduct at about three milliamps; at this point the voltage across it will drop, to about 70 volts IIRC, and if enough of the remainder is dropped across R3, the other lamp will not ionize. You can measure the voltage at points A and B to see which lamp is conducting; the other one will be at zero volts. And if you apply a sufficient positive voltage at A (anything from about 50 volts up to not too far above "VCC"), you can get L1 to stop conducting if it's conducting beforehand, ensuring that L2 is conducting afterwards; and likewise for B.

This might avoid the need for separate indicator lamps, but NE-2s are pretty small, dim, and red, so your light detector has to be pretty good. However, there are other neon glow lamps that work the same way that produce more light. In extreme steampunky cases, you could perhaps even use a carbon arc lamp, which displays more or less the same bistable negative-resistance behavior that neon glow lamps do, so much so that the Pearson arc oscillator used it to power an RF oscillator circuit.

This flip-flop can be extended to multiple branches; as I sort of mentioned before, three alternatives gives you slightly better efficiency than two; six lamps in two three-branch circuits of this type can encode nine alternatives, compared to eight if they're grouped into three two-branch circuits.

So how do you connect a high voltage to A or B when one of the next-state bits gets illuminated? In today's world you'd probably amplify a signal from a photodiode to trigger a triac or SCR or something. But using neon lamps? They're slightly light-sensitive, and in their negative-resistance region, capable of amplification, so theoretically this ought to be possible, but I don't know enough of the details.

You do have the advantage that the light from the "new state" bits can be as strong as you like (although only 20 or 25 times stronger than their dark counterparts), since you're just turning the light source off when you're not transitioning to a new state, so let's suppose that we can do it with three resistors and two neon lamps for each of A and B. That means each of our N=4 bits of state consists of 6 lamps and 9 resistors, for a total of 24 lamps and 36 resistors; our bit of input requires perhaps two lamps and two or three resistors.

That leaves only the problem of detecting the darkness that triggers the transition and firing up the transition circuits. I'll guess arbitrarily that that will need five more lamps and eight more resistors, for a total of 29 lamps and 44 resistors.

It's also feasible to use electromechanical relays, of course. Modern mercury-wetted reed relays can run up to 40kHz. But they can't detect light; for that you need an electric-eye vacuum tube at least, if not an avalanche gas tube.

Topics

Sun cutter

Kragen Javier Sitaker, 2016-09-06 (9 minutes)

Could you replace a laser cutter with a focused-sunlight cutter?

I mean obviously you can burn through things with focused sunlight — I've been doing it since I was a kid — but I'm asking if you can match the performance of off-the-shelf low-power laser cutters.

The laser cutter I've been using to cut MDF cuts a 100-µm kerf positioned with 60-µm precision using a 60W laser through 3-mm MDF at 24 mm/sec, although it can handle up to I think 12 mm at lower speeds. If we figure that the divergence of the beam can't be more than about 50 µm over those 3 mm, that's about a sixtieth of a radian, 16.7 mrad, almost twice the visible width of the sun, which is about 9.3 mrad.

Optics won't work

So there are two problems here: one is to focus, say, 100 W of sunlight in an area that's less than 100 µm across, and the other is to keep the divergence of that focused beam down below, say, 10 mrad.

The very direct approach is to use a single movable parabolic reflector at a sufficiently large distance. But this will not work very well. 100 W is about 0.1 m² of sunlight at the surface, a mirror of 357 mm diameter; this is 10 mrad at a distance of 35.7 m. A geometrically-perfect image of the sun at that distance would be 9.3 mrad, which would make a 332-mm-wide image of the sun, which is substantially larger than the 0.1 mm we are shooting for.

This amounts to a spot that is 3½ orders of magnitude too wide and is 7 orders of magnitude too dim.

Getting a smaller and thus brighter image from an imaging-optics system involves shortening the focal length, as all photographers know. The desired power density here is 10 GW/m², which is (not totally coincidentally) 7 orders of magnitude brighter than sunlight.

This is somewhat problematic because the illuminance limit imposed by thermodynamic reversibility is for the focal spot to be entirely surrounded by (reflected or refracted) sun surface, that is, 4π steradians of sun. A 9.3 milliradian cone has a solid angle of about 0.27 millisteradians ($2\pi(1 - \cos(9.3\text{ mrad}))$), and so the theoretical maximum is only about 46000 suns, with 23000 suns being the limit for a point on a flat surface only being illuminated from outside the surface. This is still three orders of magnitude dimmer than the laser — and without even being pulsed!

Therefore even non-imaging optics can't help us here. We need stronger stuff than mere optics.

One possibility is to concentrate the light optically as far as possible, then use some other approach to deliver the power to a small area. We don't actually have to violate any laws of thermodynamics to do this; the MDF doesn't have to get hotter than the surface of the sun (5500°), and we don't have to deliver all of the energy to it, but can waste some in pumping heat around. One obviously feasible approach is to use photovoltaic panels to power the existing electric CO_2 laser, but are there more direct routes, maybe more efficient ones?

Hot fluid flow

One obvious (to me) example is to use the sunlight to heat a fluid to a sufficiently high temperature and then cause the fluid to flow through the MDF. Air is one possibility, although it might tend to catch the MDF on fire even more than the laser does. Lead, which doesn't dissolve much iron and doesn't boil until 1749°, is another possibility; at higher temperatures, copper, which doesn't melt until 1084° but doesn't boil until 2562°, or silver, which melts at 962° and boils at 2162°, might resist oxidation better. (Molten silver, however, has a tendency to attack steel.)

Air is probably going to be too difficult

How fast could air deliver power? Suppose the hot air stream is limited to 1600° in order to be able to use ordinary ceramics to control it; air's specific heat is 29.2 J/K/mol, and it weighs about 30 g/mol. Let's suppose that the MDF cools the hot air down to about 500°, so we have 1100 K to play with. Delivering 60 W then requires a flow of about 1.9 millimoles per second, which works out to about 56 mg/s, which would be 56 mℓ/s at normal temperatures — but at 1600° it's more like 360 mℓ/s. Dividing that by the area of a 50-µm-radius circle, we get the utterly implausible air speed of 46 km/s, roughly Mach 138.

(I would have liked to use metals, but even superalloys are limited to about 1000°.)

If we relax the requirements considerably, we can get into a feasible range; suppose that we make our air nozzle out of white-hot quicklime instead of regular ceramics, so that we can use air at 2600° (reducing the flow rate to 290 mℓ/s), and make the air stream 1 mm in diameter instead of 100 µm. Then we can get the air speed down to 370 m/s, Mach 1.1, which is probably feasible. (Turbulence in the nozzle just transforms into more heat in the air!) A slightly wider nozzle of 1.35 mm diameter can get your gas stream speeds down to 200 m/s, which is quite clearly feasible. But if you're blowing white-hot air on it, it's going to be really difficult to keep the MDF from catching on fire; you probably need an inert-gas atmosphere, nitrogen at least, which probably means you need to filter and recirculate it.

Lead is more feasible

Molten metal is probably a much more feasible approach; even at room temperature, it's about ten thousand times as dense as air, and that advantage increases with temperature rather than decreasing.

Lead's vapor pressure is a still-relatively-safe 10 Pa at 814° (an atmosphere is 101 kPa). Tin is a more expensive but nontoxic alternative with an even lower melting point (232°), an even higher boiling point (2602°), and an even lower vapor pressure; I think it is more vulnerable than lead to oxidation in air. The eutectic 63%-tin mixture of the two melts even lower.

If we figure that we can work from 900° down to lead's freezing point of 327°, then take advantage of its 4.8 kJ/mol heat of fusion, how much lead flow do we need? Its molar mass is 207.2 g/mol, so that's 23 kJ/kg of latent heat of fusion. Engineering Toolbox says molten lead's heat capacity is 140 J/kg/K, and we have 573 K here, so that's 80 kJ/kg of sensible heat. Together, they're 103 kJ/kg, so our

flow rate is only 580 mg/s; with molten lead's density of about 10.7 g/cc, that's only 58 microliters per second of lead.

Through a 100-micron-diameter circle, that's still 7.4 meters per second, which is some pretty healthy metal pumping action; I don't know what molten lead's viscosity is, but that won't be easy. But it's surely feasible.

Hmm, this IAEA document on nuclear reactor design says lead's dynamic viscosity is about 1 mPa·s in the temperature range considered. A Poise is 100 mPa·s. Water's viscosity at 20° is 1.002 mPa·s, so we can treat lead as especially dense water in this temperature range. This document also describes in some detail the physical properties of the lead-bismuth eutectic coolant used in many Soviet reactors. Some random engineering calculator site claims that 1 mm of 0.1 mm-diameter pipe at 58 μℓ/s should result in about 13 MPa of head loss; this is a straightforward pressure to achieve, and it amounts to a force of about 0.1 N across the area of the nozzle, and a hydraulic power of 750 milliwatts.

These numbers are all easily feasible.

Solar-pumped lasers

Another possibility is to use the sunlight directly to pump a lasing medium, such as a doped fiber; this is called a "solar-pumped laser", and there are apparently a number of them around, mostly Nd:YAG, including a megawatt one in Uzbekistan (!).

Topics

Making the CPU instruction set a usable interactive user interface

Kragen Javier Sitaker, 2015-09-17 (8 minutes)

In designing simplified computing systems for bootstrapping, one of the problems is that even once you have built a CPU and RAM, you still somehow need to bootstrap useful software into the thing. At one point this was usually done by entering an "IPL" or "bootstrap" program into the memory one bit at a time using switches, then making the CPU run, but this requires the electronic interface to the CPU and RAM to support such direct manipulation, which complicates it somewhat. Also, it's a terrible user interface.

Somewhat later, machines came with a "monitor program" in ROM, which would use the usual I/O mechanisms to interact with a user and allow them to enter the IPL program using, say, a hexadecimal keypad. The Heathkit H8 was an 8080-based computer that worked in this way. The later H89, which was a miniaturized Z80-based H8 built into an H19 terminal, substituted a terminal-based monitor program. Other PCs of the era were less slavish followers of the minicomputer tradition and typically had Microsoft BASIC in ROM, and would run that initially, sometimes giving you the option to use it to load other binary software from other media.

However, all of these alternatives involve a fairly large amount of ROM and RAM: tens of thousands of bits, if not more. An amount that would take an unreasonable amount of time to solder together out of individual transistors, let's say.

The HP 9100A took a different approach that didn't require a lot of ROM (well, except the 32-kilobit microcode!) or RAM. It was sold as a calculator, a forerunner of HP's later popular line of electronic pocket calculators (beginning with the HP-35), but it was in fact a programmable computer, just with a very small amount of read-write memory: each of its 19 registers was 14 6-bit-wide BCD digits, for 1596 total register bits. This is a very small amount of memory, but the 9100A took full advantage of it by using a 6-bit instruction set, thus supporting programs up to 266 instructions long. (I think. Maybe you couldn't store program steps in the X, Y, and Z registers. Certainly it wouldn't be very useful to do so.)

The more interesting thing about this is that the HP 9100A's instruction set was in fact its keyboard: each keystroke, more or less, executed a CPU instruction. This, plus a simple mode for storing a sequence of keystrokes in memory and an instruction set tailored to support use as a user interface, allowed it to survive without a ROM monitor.

(Unfortunately, both its keyboard and its display were tightly coupled to its processor; there was no mode that allowed a program to interactively process keyboard input, as opposed to processing data entered into memory before it was started, or to display data of its choice on the display.)

This might turn out to be a useful way to simplify bootstrapping a computer: rather than toggle data into its memory with a direct

front-panel memory interface, and then providing it with a RUN/STOP switch and a single-step switch so that you can debug it, you can feed it instructions directly, letting the CPU interact with memory on your behalf.

One possible GreenArrays-like way to do this is to memory-map the keyboard, don't increment the program counter when executing from the memory-mapped part of the address space, and make the standard interactive keyboard-read address blocking, so the CPU halts execution until you send it a keystroke. (You could additionally provide a nonblocking-read location or an is-keystroke-ready location.) This still requires hardware to automatically display CPU registers while the CPU is halted on keyboard input in order to be useful, as well as probably some kind of interrupt to send execution back to the keyboard-read address on manual program cancellation or detected program trap.

I suspect that all of that requires less transistors than a ROM monitor routine!

My current Calculus Vaporis design would not be very usable in such a scheme. Its seven instructions are $, ., -, |, @, !, and nop, of which $ is the biggest problem.

$ includes an entire 11-bit immediate constant. You could replace it with eight separate instructions 0, 1, 2, 3, 4, 5, 6, and 7, each of which shifts the accumulator left by three bits and ORs the appropriate octal digit in. This could almost allow you to use four-bit instructions, which could be packed three per 12-bit word; the other 6 instructions would use 6 of the other 8 opcodes. 9 bits of immediate constant per 12-bit word is very nearly as good as 11 bits, although it would take three times as many cycles; also, you'd get the benefit of much denser other instructions.

This would also require a separate PUSH or DUP instruction to achieve the function currently provided by $ of getting a second item onto the stack. A PUSH0 instruction is probably the best way to do this.

However, the rest of the instruction set is not very usable as it is; it lacks unconditional calls or jumps as well as addition (it has subtraction) and bitwise operations other than NAND (|). All three of these are problems in practice for writing programs for it.

Expanding to a 5-bit instruction set, of which half could still be hexadecimal digits, would provide room for 8 bits of constant in a 12-bit word and also provide instruction encoding space for some of those other operations. (I'd vote for unconditional jump-and-link, addition, NOT, and AND, at least.) It's clear that there's a strong tension between usability and hardware simplicity in this context. A separate return stack would permit passing arguments to subroutines on the stack; a third parameter stack register, as on the 9100A, would allow the evaluation of nested expressions without an intermediate store to memory; and so on.

The "dontmove" design I've been playing with as an archival virtual machine may be an interesting alternative. Instead of a stack with different instructions to perform different ALU outputs, dontmove is nearly a MOV machine — except that the load and store parts of the MOV are done in separate instructions. It has 26 immediately-accessible registers, named with the Latin letters, and 52 instructions that load and store them from its single temporary

register. Of these, five are memory-mapped magic: Aa is byte I/O, Bb is a subtractor, Cc and Dd are a PIC-like indirect access and its pointer, and Ee is the program counter. (Writing to it magically saves the return address in the temporary register, so it's a jump-and-link.). Then you have ten more instructions to multiply the temporary register by 10 and add a decimal digit to it. Conditional jumps are obtained by setting Dd to 4 (Ee) or to somewhere else (such as Ff, which is just a normal location), then storing into Cc.

I suspect that such a machine could be roughly as simple as Calculus Vaporis, but perhaps more amenable to interactive instruction entry at the extremely low levels of complexity we're talking about.

Topics

Servoing a V-plotter with a webcam?

Kragen Javier Sitaker, 2017-02-16 (3 minutes)

I was thinking you could use webcam streaming video to provide servoing feedback on a hanging V-plotter. I mean, you could use it to provide servoing feedback for any kind of robot (at least for things that don't need <30ms or so latency) but V-plotters (aka polargraphs, drawbots, wall-plotters, etc.) are particularly promising.

There's a bunch of V-potter videos on YouTube, including some which are just video from a fixed point of view, in real time, without any change in point of view. These are not the most interesting or informative videos to watch as a human being, but they make it possible to test this concept before actually having built a V-plotter.

https://www.youtube.com/watch?v=HU9SaCFnCng shows a deformable, swinging V-plotter, more rapidly than real-time, with changes in lighting in the room, for a couple of minutes.

https://www.youtube.com/watch?v=WyLPpGdfR7s is 5 minutes from a fixed point of view of a somewhat wobbly V-plotter, from a fixed point of view, with lots of texture in the background and a few distractors. Midway through a guy walks in front of the camera, and the lighting changes. Toward the end, the lighting changes again.

https://www.youtube.com/watch?v=i5rxxGuWUo8 has a minute or so in the middle of a somewhat wobbly V-plotter drawing on a shiny whiteboard.

https://www.youtube.com/watch?v=ojcZ7kcklu4 is a sped-up video of a Polargraph drawing Spider-Man, from a single point of view.

https://www.youtube.com/watch?v=aiw3hkDvp-M has about a minute of a Polargraph drawing some classic art, sped up.

https://www.youtube.com/watch?v=qury-oBiURA has another few minutes of a Polargraph drawing. The camera moves a bit toward the end, and lighting flickers throughout.

https://www.youtube.com/watch?v=MbpT907qIHg has a couple of 1-minute-or-so sequences of time-lapse drawing, each from a different points of view. This one uses fishing line for the V, which means the strings won't interfere with tracking.

https://www.youtube.com/watch?v=MG70TvuRU9Y has a couple of 30-second-or-so sped-up segments of a V-plotter drawing from fixed points of view, one with changing lighting.

https://www.youtube.com/watch?v=5z8LTj74uiE is almost a minute of sped-up footage of a V-plotter with a novel design and substantial parallax.

https://www.youtube.com/watch?v=oBOuqLPaEMc is a very short video of a V-plotter made out of a binder clip.

https://www.youtube.com/watch?v=iE7sCiv7VTE is a particularly tricky case: the V-plotter gondola is made from a CD-R, which means that its reflection is almost entirely specular, and furthermore has diffraction rainbows all over it. The video runs in apparently real time for about six minutes.

https://www.youtube.com/watch?v=FmCWx30g7Ks is a

whiteboard V-plotter in real time for about two minutes. It's rather blurry.

https://www.youtube.com/watch?v=jknm5qtgO1A is another blurry whiteboard V-plotter in real time (except for occasional jumps), with conspicuous hanging cables.

https://www.youtube.com/watch?v=u4KyPNoswoA is a V-plotter using a spray-paint can that appears to be using the Hektor constant-speed path-design software. Its speed varies.

https://www.youtube.com/watch?v=OselyTkA6wU seems to be Hektor itself, running in real time for some minutes.

On another topic, https://www.youtube.com/watch?v=ycrGUTGlzik shows the design of the PolargraphSD gondola that keeps its wobble so low. It uses cartridge bearings to pivot the two strings around the precise XY center of the gondola.

Topics

Dercuano drawings

Kragen Javier Sitaker, 2019-04-30 (updated 2019-05-30)
(18 minutes)

At this point I've imported 90 notes into Dercuano, and the lack of images of any kind is sort of annoying. My actual, paper notebooks have a little line-drawing "icon" for each note, maybe 10 millimeters square, but often some kind of graphic would help a lot both with visual appeal and with comprehensibility. But my total byte budget of some 5MB, so that Dercuano remains easily downloadable in full, makes this challenging; I need a way to make very compact graphics.

This is related to Dercuano calculation (p. 3135) and Dercuano formula display (p. 495).

Example uses

To take a few of the notes I currently have in Dercuano at random:

Deep freeze (p. 1465) could benefit from diagrams of one or more of the following: 11 tonnes of foods including soybeans; a cutaway freezer with labels for insulation, refrigeration, and passive thermal storage; freezers of different sizes, with different amounts of surface area per unit volume, including a cubic-meter sphere and a thousand-cubic-meter sphere; food stored at the bottom of a 30-meter well; mounds of sand and locally excavated earth; a heat–temperature curve for water, including enthalpy of fusion and vaporization; plots of heat loss versus surface area and insulation thickness; etc. It would also benefit, I think, from a goofy line drawing of a freezer with a smiley face on it. A three-dimensional rotating rendering of a freezer would maybe be a plus, too; and a model whose parameters you could adjust interactively and see the relationships between the other parameters in the neighborhood would be awesome.

Dehydrating processes and other interaction models (p. 3208) could benefit from sketches of user interfaces; from diagrams of the interaction sequences and data flows being discussed, of FlatBuffers or binary array state, and of the timing of sequences; and from illustrations of punched cards and perhaps a dehydrated window or something.

Executable scholarship, or algorithmic scholarly communication (p. 2137) could benefit from a timeline, at least.

3-D printing by flux deposition (p. 466) could benefit from an illustration/diagram of the binder/powder-bed/tray setup, and maybe of some grains being fluxed — generally mechanical things like this really need diagrams; maybe also a temperature scale showing where different mixes sinter and/or melt would be helpful.

IMGUI programming language (p. 103) would benefit a lot from some sketches of the UI components being programmed, and maybe also some diagrams of activation records allocated on the stack and of Golang-style interfaces.

Transactional event handlers (p. 139) would benefit from some timing diagrams showing priority problems, concurrent transactions, transaction conflicts with pessimistic and optimistic synchronization,

and deadlock.

Possible tools

No photos

A thing I've done in the past is to draw diagrams on paper and photograph them. This allows for pleasingly direct feedback during the drawing process, but even with aggressive compression, the resulting image files are dozens of kilobytes each.

No bloated SVGs

The Web standard format for line drawings is SVG — even hand computers running iOS support it now. SVG is super cool; it supports alpha-blending, Bézier curves, gradients, arbitrary affine transforms, some degree of abstraction and reuse, and text. You can do a lot in very few bytes of SVG, and it's a lot more readable and debuggable than PostScript, my pre-SVG favorite. The standard example is probably something like this 78-byte triangle:

```
<svg width="30" height="40"><path d="M 10,10 L 20,20 10,30"
    fill="red"/></svg>
```

An example diagram in handwritten SVG

And I *have* used SVG for diagrams, for example in A mechano-optical vector display for animation archival (p. 3047):

The source code for that looks like this, which is somewhat verbose but arguably not unreasonably so.

```
<svg width="128" height="128" class="diagram m">
<use xlink:href="#burst" transform="translate(128 60.24)" />
<path class="beam" d="M128,60.24 L60,60.24 l-121.1,-992.5" />
<path class="mirror" d="M90.51,128 A90.51,90.51 0 0,0 0,37.49 v90.51" />
<path class="arrow" d="M20,57.49 a70.51,70.51 0 0 1 50.51,50.51" />
</svg>
<!-- SVG for definitions of common things used in embedded SVGs. -->
<svg style="display: none" class="m"><defs>
<marker id="v" overflow="visible" orient="auto">
  <path d="M-8,-4 0,0 -8,4" />
</marker>
<marker id="vv" overflow="visible" orient="auto">
  <path d="M-8,-4 0,0 -8,4 M4,-5 12,0 4,5" />
</marker>
<symbol id="burst" overflow="visible" class="m">
<path d="M0,0 l-16,-16 16,16  -16,16 16,-16  -16,-8 16,8
        -16,8 16,-8  -8,-16 8,16  -8,16 8,-16" class="beam" />
</symbol>
</defs></svg>
<style>
svg.diagram { margin-left: auto; margin-right: auto; display: block }
.m path { stroke: black; stroke-width: .5px; fill: none }
.m .beam { stroke: red }
.m .mirror { stroke-width: 0; fill: #ccc }
.m .arrow { marker-end: url(#v); marker-start: url(#vv) }
```

```
</style>
```

However, you can probably kind of tell from reading that that it was pretty slow to write, and from looking at it that the workflow doesn't really support rapid iteration to get it to look good.

What you can't necessarily tell from looking was that when I added that diagram to this document, it broke the triangle example higher up, so I hacked it not to interfere, and then I had to spend some time debugging incompatibilities between Firefox and Chromium about when they applied certain styles (arguably a bug in Firefox).

SVG's abstraction capability is very limited

That diagram exploits SVG's capabilities for reducing duplication to the maximum, to the point that I had to bring in code from three separate parts of its source document to use it here (and then, as I said, hack them so they wouldn't break other SVGs). It uses a stylesheet with overrides to specify how lines, arrows, and areas should be drawn, and it uses SVG's `<marker>` and `<symbol>` facilities to define graphic elements that can be used many times in the same drawing or across many drawings.

But those facilities, though complex, expensive in terms of syntactic overhead, and hard to debug, are still very limited. You can reuse a `<symbol>` in different places, for example, but not with different line widths or colors. (In theory I think you can, but it doesn't fucking work, at least in Firefox.) You can apply the same style to different pieces of text or different paths, but as far as I can tell, you can't instantiate a rectangular component at different widths in different places, other than by stretching the whole component, including its line widths. The kind of simple constraint satisfaction we routinely apply to HTML with the CSS box model is entirely outside of our reach in SVG.

WYSIWYG SVG editors produce insanely bloated output

On the other hand, the standard tools for generating SVGs produce data that looks more like this:

```
<path d="M 21.789062 16.140625 L 20.332031 15.941406
        L 20.335938 15.742188 L 20.371094 15.53125
        L 20.453125…
```

That's eight significant figures on every coordinate, absolute coordinates everywhere instead of relative, and using diagonal lines instead of H horizontal or V vertical lines even when the relevant coordinate difference is in the fifth significant figure, and also using unnecessary spaces. The extra significant figures are essentially random, so gzip isn't going to be able to compress them, except by noting that, being digits, they only need four bits each.

The problem with this kind of thing is that, even when it's hand-drawn, it mixes the actual desired signal with a lot of random noise which is hopefully imperceptible but still incompressible. The mouse produces, say, 100 positions per second, each with three significant figures in each of X and Y, though the number is smaller

with relative positions. Crudely, that's 6400 bits of data per second, 800 bytes per second, after compression, that can be added to the Dercuano download package.

I launched Inkscape, plugged in a mouse, and drew with the pencil tool for a while. Running Inkscape for 110 seconds, I produced a 54-kilobyte SVG that compresses to 15.6 kilobytes; most of it looks like this (line breaks added for clarity):

```
<path
   style="fill:none;fill-rule:evenodd;stroke:#000000;
          stroke-width:1px;stroke-linecap:butt;
          stroke-linejoin:miter;stroke-opacity:1"
   d="m 94.285714,946.64792 c 7.539686,-3.23129 21.663356,-10.54083
        31.428576,-11.42857 15.99002,-1.45364 19.9212,2.36631
        34.28571,-2.85715 6.00412,-2.18331 11.32674,-5.92773…
```

That is, Inkscape has converted my three-significant-figure mouse coordinates into coordinates with seven to nine significant figures. Just in case. How helpful.

Why this is too much bloat for Dercuano to tolerate

The trouble with this is that 15.6 compressed kilobytes per 110 seconds is 1.1 kilobit per second, and the entire five-megabyte target size for Dercuano — containing many years of notes — would be completely filled with SVG in less than ten hours of drawing time. I've already spent over 20 hours just getting the first 600 kilobytes of Dercuano to work so far.

The basic reason many years of notes fit into five megabytes is that I can only type about 90 words per minute, which is 72 bits per second uncompressed, 24 bits per second compressed — a bit rate 45 times lower. And then I spend time revising the notes, which often makes them better without making them larger, though as anyone can see, I don't revise my notes nearly enough.

TikZ won't fit

One of the SVGs I excerpted above is a sort of graphic of a comic-book-style explosion or impact, but with a gradient. I generated it with the TikZ graphics system for TEX from the following input:

```
\documentclass{standalone}
\usepackage{tikz}
\begin{document}
\tikz\shade[inner color=yellow,outer color=red](1ex,0)
    \foreach\t in{4, 8, ..., 360}
        {-- (\t:{.1+Mod((\t/17)^3, 57)/100})};
\end{document}
```

This uses $(\theta \div 17)^3 \% 57$ (composed with an affine function) to generate a "random" radius for each of 90 different angles to generate a jagged outer polygon, then fills it with a gradient. This is not something you can do in SVG, although you can do it in something else (like TikZ or JS) and generate SVG programmatically. And

TikZ comes with a massive built-in library of things like arrowheads, directed graph layout algorithms, tree layout, plotting math functions, calendars, finite state machines, ERDs, Petri Nets, and so on. The sections of the TikZ manual about arrowheads (and arrow tails, etc.) total about 10 pages. It's a bit overwhelming, honestly.

Unfortunately, I can't use TikZ directly in Dercuano, because that would involve embedding not only the 5 megabytes of TikZ, but also the rest of TEX and LATEX that it depends on to run, in some kind of browser-executable form, probably compiled with Emscripten. And, while that's probably a feasible thing to do, I'm pretty sure would blow my space budget for Dercuano.

I could draw ("write"?) graphics in TikZ and generate SVGs from them, but that just puts us back at square one: the SVGs are bloated, and I'd blow my space budget even faster that way.

TikZ has another problem, too: it's purely intended for static graphics. But in Dercuano much of the time a dynamic, interactively-responsive graphic would be better, and only marginally harder to write.

d3 may be a good option, but not for illustrations

d3.js is a JS library for dynamic data-driven graphics which generates SVG at runtime using the DOM. One of the minified copies of d3.js I have here is 151 kilobytes, and it gzips to 53 kilobytes; the current v5.9.2 gzips to 79 kilobytes. Another, non-minified copy I have is d3 v3, and it's 315kB, gzipping to 68kB. It's pretty easy to use, and in particular it's pretty easy to get really nice graphical output from, and it's amazing at interactivity. On the downside, the JS code to use it is usually pretty verbose, and it's not well suited for the kinds of sketching and illustration I earlier said were most important, although it's probably fine for timelines.

Some kind of restricted-bandwidth WYSIWYG drawing program

For some things, though, the closer to paper, the better. I just don't want to pay half a megabyte per hour for the privilege of faithfully recording the shaking of my hands as I sketch; shaky lines can be good but we might as well produce the shakiness from a highly-compressible random number generator. One possibility here would be drawing with a line that thrashes around near the mouse, starting at low-entropy points in whatever encoding I end up with, and gradually moving to higher-entropy points. Or maybe you stroke the mouse along a line multiple times to coax it into the shape you want. Maybe snap to a grid that gets finer if you zoom in: then you'd need to zoom in. Or maybe you need to click to nail down a spline point, and there's a snap grid that is finer for smaller displacements.

Another possibility would be something that encourages drawings to be highly factored, perhaps by having clumsy graphical primitives (though primitives that compose flexibly, e.g., adopting the line width and color from their use context) and really low reuse overhead.

A third possibility, and this is kind of cheating in a way, is just to use a textual language to describe the graphics, but shorten the feedback cycle as much as possible so that you can vary things interactively to see the results on the screen. To the extent that you

can map backwards from the screen image to the textual source code, you may be able to alter the textual source code by direct manipulation of the screen image.

Ivan Sutherland's SKETCHPAD was a visual programming language for interactively constructing a set of graphical objects and constraints for them to fulfill, progressively approaching fulfillment, and visualizing the results.

The problem with these approaches is that in some sense you're deliberately impeding the drawing process, which you would think would make it clumsier. But maybe that clumsiness can be minimized or even reversed — the bandwidth from my brain to the mouse is definitely not 1100 bits per second and may not even be the 24 bits per second I get on the keyboard. If the drawing program can somehow filter out the other 1000+ bits per second of pure noise, it might actually make it *easier* to draw things rather than harder.

Penrose looks like an appealing approach, but would need reimplementing

The unfortunately named Penrose (GitHub) diagramming software project is not related to Penrose diagrams (also known as Penrose–Carter diagrams), the Penrose Graphical Notation, the Penrose Project (a band), or the OpenJDK project Penrose. Instead, it's Katherine Ye's research team at CMU, which has produced software that largely consists of three DSLs, with even more unfortunate names — "DDL" defines the primitive objects of some mathematical domain, such as the vector space $\mathbb{R}^2$ or ZFC set theory; "STYLE" explains the desired visual representation of each domain concept in terms of constraints and optimization objectives; and then "SUBSTANCE" describes the particular objects to put into a particular diagram. Further interactive tweaking is then used to adjust one of the various outputs from the constraint solver to look best.

The underlying approach sounds reasonable, and they've gotten some impressive results, but the implementation is 8000 lines of Haskell, plus substantial bits in Java, TypeScript (for a React UI), JS, and Emacs Lisp; they're actually using Alloy (thus the Java) to reduce some kind of problem to SAT and solve it with SAT4J (maybe the layout optimization problem itself, although I don't think SAT4J would be particularly good at that? Or maybe some other problem). Using Penrose itself for diagrams in Dercuano would thus involve generating vector graphics in SVG or whatever during the Dercuano build process, since running the existing Penrose software in the browser would not be practical, even with Emscripten, which itself would probably blow Dercuano's space budget.

Something like Zdog might be interesting for 3D

Zdog is a browser-based 3-D engine using points, lines, arcs, and splines rendered with stroke thickness, with SVG (and, I think, `<canvas>`) output. Its implementation, though lightweight, doesn't run in old browsers, and I'm not that fond of its API.

It does a great job of demonstrating the potential of the approach, which is pretty easy to implement; doing it just for points with diameter ("spheres") took me just a few lines of JS on `<canvas>` a few years ago:

```
var s = Math.sin(theta)
  , c = Math.cos(theta)
;

var ty = y.times(c).plus(z.times(s))
  , tz = z.times(c).plus(y.times(-s))
  , seq = tz.gradeDown()     // painter's algorithm
;

ctx.clearRect(0, 0, canvas.width, canvas.height);
for (var ii = 0; ii < seq.length; ii++) {
    var i = seq[ii];

    circle(ctx, x[i] + 128, ty[i] + 128, 2048 / (tz[i] + 128), color(r[i], g[i], o
ob[i]));
}
```

In that case, though, the "scenegraph" (coordinates and colors) was randomly generated at startup. It doesn't really help with the question of how to get the 3-D models made in the first place.

Topics

- Graphics (p. 3483) (91 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Compression (p. 3384) (28 notes)
- Dercuano (p. 3406) (16 notes)
- Multitouch (p. 3591) (12 notes)
- JS (p. 3533) (12 notes)
- Constraint satisfaction (p. 3387) (9 notes)
- Sketchpad (p. 3713) (3 notes)
- TikZ
- SVG
- Inkscape
- D3

Plasma glazing

Kragen Javier Sitaker, 2019-04-24 (1 minute)

Instead of salt-firing ceramic to glaze it, could you glaze it by producing a sodium-containing plasma on or just above its surface? For example, you could blow powdered sodium chloride into a plasma arc to vaporize it, or you could generate the arc from consumable wire electrodes embedded in a solid rod of sodium chloride, perhaps mixed with fillers to add gas volume — for example, sodium bicarbonate (which would also serve as a chlorine-free source of sodium, but might produce carbon), sodium nitrate (which wouldn't produce carbon but would produce nitrogen oxides that are worse than chlorine), boric acid, or sodium borate (borax).

Topics

• Materials (p. 3560) (112 notes)
• Ceramic (p. 3371) (17 notes)
• Clay (p. 3378) (4 notes)

The Tinkerer's Tricorder

Kragen Javier Sitaker, 2013-05-17 (updated 2014-04-24) (27 minutes)

My friend Nick and I were talking about a tool called the Tinkerer's Tricorder, which, if it existed, would tell you everything interesting about an electrical component as soon as you connected it across its terminals. Inductance, capacitance, resistance, voltage, frequency, diode characteristics, resonant frequency, Q, and so on — all with no danger of damaging the device!

Some of Nick's notes are at http://dangerousprototypes.com/forum/viewtopic.php?f=19&t=53606&p=52238.

I had wasted some time earlier that same day trying to debug a circuit without any test equipment. I ended up programming an Arduino to be a crude volt/ohmmeter. Despite its imprecision and the need to use a laptop to see its readout, it had three big advantages over a standard multimeter:

• It, or rather the laptop, recorded a series of measurements instead of a single one, so I could see not just the current state but recent states.
• It measured both voltage and resistance without any need to switch modes.
• Most of all, it could be simply downloaded and installed on existing hardware, without requiring any additional hardware — not so much as a resistor soldered across a pair of pins!

So I'd like a Downloadable Tinkerer's Tricorder — the closest possible approach to the Tinkerer's Tricorder on common hardware like an AVR microcontroller or an Arduino, without needing any nonstandard hardware — not even so much as a resistor or capacitor soldered across a couple of pins. You just reprogram the flash, connect the device under test between two pins, and the AVR tells you what it is. Maybe you go through some kind of calibration step if you want accurate results.

Nick's project is far more capable, but I think its parts cost is comparable to the cost of a prebuilt LCR meter.

How capable could a Downloadable Tinkerer's Tricorder be, struggling under the limits of common AVR hardware?

What an AVR pin can do to the outside world

Basically an AVR digital GPIO pin can be in one of four states: output high, output low, input high, and input low. Although some exotic Arduinos have DACs, common Arduinos don't have anything but digital GPIO pins plus an ADC. (This is deeply unfortunate, since the core of the AVRs' successive-approximation ADC is a DAC — but its output isn't routable to any pins!)

Output high ties the pin to the 5V rail with relatively low impedance; it can source up to 40 milliamps, but if you connect the pin directly to ground, the chip doesn't blow up; it's internally limited to about 40 milliamps. This means that, at least at high currents, the internal output impedance is about 25Ω. This is

convenient for connecting AVRs to LEDs and speakers; it pretty much guarantees you won't blow up the AVR, the LED, or the speaker no matter how ignorant you are, unless you hook the LED up to your power supply instead of the AVR.

Output low is similar, but ties the pin to the ground rail through the ≈25Ω impedance.

Input high ties the pin to the 5V rail via a pullup resistor, which is nominally 20–50kΩ on the ATMega328, which is not what you'd call a really fucking precise specification.

Input low tristates the pin, i.e. sets it at the very high input impedance characteristic of CMOS inputs, around a megohm; the maximum rating from the datasheet is that these pins draw up to 10 µA.

The pins are also diode-clamped to the power rails, which should protect them from voltage spikes if you're testing an inductor and slam a pin to input state. The clamping diodes aren't well-specified in the datasheets, but various AVRFreaks threads claim that these diodes are equipped to handle only up to about 1.5mA before possibly going into latch-up and killing your chip. The ATMega328P datasheet instead gives a spec for the voltage range: -0.5 V to VCC + 0.5V.

The datasheet max specified I/O pin capacitance is 10 pF, although I suspect 1 pF is probably more typical. I haven't measured.

What an AVR can measure

The usual AVRs, including the ATMega328, have a 10-bit ADC with a sample-and-hold frontend, which measures relative to either the power-supply voltage (typically 5 volts), and an internal 1.0–1.2 volt analog bandgap reference voltage; the ADC is claimed to have a 38.5kHz input bandwidth, but people have reported success clocking it up to 4MHz and still getting an Effective Number Of Bits around 5.

Furthermore, the AVR can use its ADC to measure an internal temperature sensor, supposedly accurate to ±10°C. This is important because it means that you can, in theory, compensate for thermal variation in things like pull-up resistor resistance. (The two major factors of variation in the behavior of a given electronic device are supply voltage and temperature. You can apparently measure the internal bandgap relative to the supply voltage, getting a value somewhere around 200, and thus measuring the reciprocal of the supply voltage, by setting ADMUX to 01x01110.)

I'm assuming the internal voltage reference is usable without any external hardware, even a capacitor attached to the AREF pin, but I'm not sure.

10 bits isn't much, but the device can measure time much more precisely. The internal RC oscillator is nominally accurate to 10% (3 bits), unless calibrated by the user, in which case it's accurate to within 1% (7 bits). However, in the common case that the AVR uses an external quartz crystal, its timing accuracy depends on that crystal's accuracy, which is normally about 0.005% (15 bits). Current normal Arduinos (Pro, Pro Mini, Uno) use a 16MHz Murata ceramic resonator that's accurate to 0.5% (8 bits), and older Arduinos (everything before Duemilanove, including Mini) use a quartz crystal. The AVR microcontrollers currently used can run at 20MHz, but the Arduinos run them at 16MHz for compatibility with the 16MHz

ATMega8 used in the first Arduino.

(It appears that temperature and supply voltage are the major factors in the speed variation of these resonators, says this guy who was tracking his against atomic clock broadcasts from WWVB and this guy who was measuring against GPS. So maybe with the temperature and supply voltage sensors, you could reduce the errors. Temperature-controlled crystal oscillators can reach three orders of magnitude better than raw crystals, i.e. 0.05 ppm temperature stability (24 bits).)

Low voltage isn't practical

Nick tells me ESR meters usually work at very low voltage to enable in-circuit component testing without activating any nearby transistors — under 0.1V. I don't think that's practical for the Downloadable Tinkerer's Tricorder because the chips I want it to run on don't have controllable-voltage output pins.

Avoiding damaging devices under test

A tricky thing is that, when you hook up some unknown device to your AVR, the AVR doesn't have the opportunity to consult its datasheet before applying power to it in order to figure out what it is. So it seems like a good idea to design the tricorder's strategy to minimize the risk of this.

The first and most obvious kind of damage is overheating. Too much voltage across almost any electrical device will cause it to generate more heat than it can dissipate until it gets really hot, hot enough to damage or destroy it. Since we're talking about a max of 5V at 40mA here, or 200mW, this is certainly a possible danger here; there are plenty of tiny resistors and things out there that can't dissipate 200mW at any kind of reasonable temperature. However, for macroscopic components, we can rely on their thermal mass to allow us to dissipate 200mW for a short period of time — if we can manage to use a duty cycle of 5%, we should be able to use 5V with near impunity, at least on discrete components, since we'd dissipate an average of 10mW.

Even the lowest-current-rated discrete diodes I could find on Digi-key were rated for 10mA average, so I think briefly passing 40mA through an unknown diode should be safe.

Integrated circuits are more delicate when it comes to brief overheating; a presentation I dug up from Cypress on "electrical over-stress" mentions, among other things, "melted or vaporized bond wires", with horrifying photos. Still, I don't think we're talking about 40mA there.

MOS circuits in general can be destroyed by overvoltage, even very briefly: it punches holes in the "O" layer in between the "M" and "S" by means of avalanche breakdown, and those holes are permanent. However, I don't think there are any MOS devices out there that will be destroyed in this way by 5 volts, so I don't think this is a real danger.

I seem to recall that semiconductor junctions can be damaged by high current density, even without high temperatures, but I can't find a citation for that now.

Electrolytic capacitors in general can be damaged by reverse-biasing, sometimes exploding spectacularly. The super-teensy

tantalum capacitors are particularly famous for this. I dug up a NASA paper on tantalum capacitor damage by reverse bias, and it seems like the damage takes place by means of a microamp (!!) or so of current. But it doesn't seem to happen below about 3 volts, and at 3 volts, it takes several hundred seconds. Even at 5 volts, it takes several tens of seconds. I suspect that short exposures to reverse bias can be corrected by a similar and rapid exposure to forward bias, reforming the dielectric layer that may have been disrupted.

Electrolytic capacitors, however, have the great advantage in this case that they have very high capacitance, 0.1µF or greater (typically 100µF or greater), even when back-biased (until they start conducting). If you're charging an 0.1µF capacitor through a 20kΩ pull-up resistor, your RC constant is 2ms. This should give you plenty of time to determine that you have a lagging voltage waveform, and therefore a capacitor, and turn the current around before the voltage reaches 3V.

Diodes in general will conduct if you put enough reverse voltage across them, either by avalanche breakdown or zener breakdown; diodes that are not designed for this will be destroyed. LEDs tend to have fairly low reverse voltage limits, but Nick tells me they're still over 5V, so not to worry. (I did find a Skyworks small-signal Schottky diode with a 1V maximum reverse voltage. But I could only find three diodes that fragile in the entire Digi-Key catalog.)

Measuring voltage

So, the simplest thing to measure would seem to be a DC voltage applied to a pin, or across two pins. The ADC can measure it directly, as long as it's positive.

For 1.1 to 5V, ADC values of 205 or so to 1023 measure the voltage with 0.25% to 0.05% quantization error: 0.005 volts. Of course, this is relative to the supply voltage, which probably isn't regulated to within even 0.25%; Joris van Rantwijk found that a single GPIO pin dumping 30mA introduced a 1% error.

For 0 to 1.1V, you can set the internal bandgap reference as the reference voltage, so as to measure with millivolt precision. However, the internal bandgap reference is specified to have ±10% error; presumably it's highly stable over time on a given chip, but varies from chip to chip.

If you can connect the voltage source between two pins, instead of to a single pin (implying that the voltage doesn't share a common ground with the AVR), then you get a couple of additional benefits:

• If you set pin A as low input and pin B as low output, then you'll measure a positive voltage on pin A if the positive end of the source is connected to pin A, and a zero voltage (and some heat dissipation in the clamping diode) if the negative end of the source is. I think you can also measure a positive voltage on pin B in that case, since whatever current is being sourced by the clamping diode to ground on pin A is being sunk through the nonzero output impedance on pin B. But in that case, you can set pin A as low output and pin B as low input, and you'll measure the positive voltage on pin B.
• This also means you can measure AC voltages, although only either the upper or lower half of the waveform at any one time, by switching back and forth.

• For 3.9 to 5.0V, you can get higher precision measurement. Suppose WOLOG that you've measured that pin A is the more positive end of the source. Then you set pin B as low input and pin A as high output. Now pin B sees a voltage of (5.0 - DUT) volts, which it can then measure with millivolt precision by using the internal bandgap reference. This trick is limited by the stability and precision of your 5V reference, which, unfortunately, is probably worse than 0.25%. (You can re-measure it to 0.25% against the internal bandgap frequently, but you can't get the 0.02% precision this seems to promise without a more precisely controlled reference voltage.)

This range of tricks gives you variable voltage precision as follows:

• -5.0 to -3.9 V: 1mV
• -3.9 to -1.1 V: 5mV
• -1.1 to 1.1 V: 1mV
• 1.1 to 3.9 V: 5mV
• 3.9 to 5.0 V: 1mV

This gives you at least 0.25% precision for -5 to -2.2 volts, -1.1 to -0.4 volts, 0.4 to 1.1 volts, and 2.2 to 5 volts; at least 0.45% precision from -5 to -0.22 volts and 0.22 to 5 volts; and at least 10% precision from -5 to -0.01 volts and 0.01 to 5 volts.

(As usual, linear analog-to-digital conversion is a bad deal here. If we wanted 0.45% accuracy for anything up to 5 volts and had 10 bits to work with, a smooth exponential distribution of levels 5/1.0045**n for n up to 1024 would give us 0.45% accuracy down to 51 millivolts, rather than 220 millivolts; that is, 10 bits would cover an entire order of magnitude.)

The four different ways to configure the two pins to measure a voltage between them (LI LO, LO LI, LI HO, HO LI) provide two different ways to measure any given voltage, to different precisions, while only allowing microamps of current to flow. If these measurements disagree, then you know the device isn't actually connected across the terminals, or isn't a voltage source. Since none of these configurations allow more than microamps of current to flow, none of them should be dangerous to electronic components that aren't generating voltage.

If the voltage source is just connected to pin A, then pin A might read the same voltage in the LI LO and LI HO configurations (that is, changing the state of pin B might not affect pin A much). But if pin A is floating, it's not a great idea to depend on it to do much of anything; it might follow pin B through internal leakage or capacitive coupling. So you might try the HI LO and HI HO configurations instead, just in case, but perhaps only briefly — the pull-up resistor could source enough current to damage, say, back-biased tantalum capacitors.

If you know that you have a voltage source between your terminals, the next thing to do is presumably to evaluate how much current it can deliver, which you should maybe be able to do by setting both terminals to low output (LO LO). In this voluptuous configuration, the negative terminal of the voltage source is forced below ground and fed current from the clamping diode, while the positive terminal feeds up to about 40mA of current to ground through its impedance. You should see 0V on one terminal and a small voltage on the other, which voltage should increase (probably nonlinearly!) with the current being sucked from the voltage source. If it can provide more than about 40mA (AAA batteries can provide 500mA) you won't learn anything useful.

LO LO should be safe for any component that isn't generating voltage itself, since it's just trying to put both of its terminals at the same voltage.

The trouble with LO LO or HO HO with a voltage source is that you're almost certain to be overtaxing the poor little clamping diodes. It might make more sense to use LO HO or HO LO, with the appropriate polarity to keep either of the terminals from going outside the power rails.

Measuring resistance

A resistor between the two terminals will tend to bring their voltage to equality in any of the five configurations suggested for measuring voltage sources in the previous section (LI LO, LO LI, LI HO, HO LI, LO LO). In the first four configurations, the input pin will follow the voltage on the output pin, and quite quickly too — if you have a 1pF pin capacitance and a 1MΩ resistor, your RC time constant is 1µs, 16 clock cycles on the Arduino, far too fast to digitize the charging curve with a mere 35kHz ADC. You might be able to beat that by doing a lot of transitions and digitizing one sample after each transition, at a variable delay, but you're going to be out of luck when you get to 10kΩ resistors, let alone 100Ω resistors.

But if one pin is following the other like that, you can infer that there is perhaps a resistance connected between them, and try to run some current through it to see what happens.

The first thing to try is presumably HI LO or LO HI, which forms a voltage divider between the pull-up resistor and the unknown resistance. If you somehow knew the resistance of the pull-up resistor — more accurately than "20–50kΩ" — you could use this to calculate the unknown resistance with some precision. Without that calibration, you're just ballparking it, although you can measure the ratios between resistances quite accurately.

(In practice the pull-up isn't actually a linear resistance.)

For the case where the pull-up resistance curve is known accurately — let's say 35kΩ — you have the following scale of measurements on the HI pin:

- 4.995V (ADC=1022 when comparing against 5V reference): $R/(R+35k\Omega) = 1022/1023$, so $R = 36M\Omega$.
- 5V (ADC=1023): $R > 36M\Omega$.
- 4.990V (ADC=1021): $R = 18M\Omega$. Clearly the error in this range is enormous, but it would still be very useful to be able to tell the difference between a 10MΩ resistor, a 1MΩ resistor, and a blown resistor. Unfortunately it's not clear that you actually get quite that far out; we're talking about currents around a microamp, and there could easily be microamp-range leakages floating around these pins.
- And so reciprocally down to, say, 2.5V (ADC=512), where $R = 35k\Omega$, and changes of one count actually do represent 0.2% changes in R. Note that at this point we don't actually care what the reference voltage is or how precisely we know it, because $R/(R+35k\Omega)$ is the ratio of voltages, regardless of what the larger voltage actually is. We *do* care about the precise value of the resistor, but it's not clear how we can find that.
- So on down to 1.1V or so (ADC=225), where $R = 9.86k\Omega$ and each count represents about 0.5% (so 0.25% error, ideally, but probably

more like 1%.) At that point we can switch to the internal bandgap reference and quadruple our resolution — but we're limited by the limited precision of our measurement of the bandgap reference relative to the total supply voltage, and now we *do* care about what the supply voltage is. Still, ideally, we can measure resistances in this range with around 0.1% precision.

- We hit ADC=225 and 0.5% precision again at 0.24 V, at which point R = 1.76kΩ.
- We hit ADC=20 and 5% precision at 21mV, R = 148Ω.
- ADC=10 and 10% precision at 10.7mV, R = 75Ω.

So far we've only been pushing about 140μA through the resistor. This should be safe for almost anything (except, as mentioned earlier, electrolytic and especially tantalum capacitors over long periods of time.) But if our device is behaving like a resistor — in particular, the measurement we get here is stable over long periods of time such as a millisecond or two, and consistent in both directions — it's almost certainly safe to put more current through it, at least for a while. And we can get a more precise measurement that way if the resistance is lower than around 1–10kΩ.

So if we switch to LO HO and HO LO mode, we're attempting to put a full 5V across the putative resistor. Now we should be able to measure the voltage drop introduced by the pin driver (the sink) on the LO pin, and perhaps less precisely the voltage drop on the HO pin.

These drivers are somewhat nonlinear — their voltage drop isn't purely proportional to the current — but treating one of them as having a resistance around 25Ω worked reasonably well for me in an experiment, over about the 50–500Ω range. Experimentation is needed.

This could involve pushing a full 40mA through the resistor, dissipating 200mW, so you probably want to do it on a low duty cycle. This will involve lots of opportunities to watch its impulse response, which could tell you if it has much inductance or capacitance.

If you're switching from HO LO to LO LO, a capacitor should dump the positive charge it's accumulated back into the formerly-HO pin, keeping its measured voltage above zero for a bit due to the nonzero input impedance, while a resistor will stop carrying current as soon as you stop applying voltage. An inductor, on the other hand, will continue carrying current in the same direction, dragging the formerly-HO pin's voltage below zero (unmeasurably, due to diode clamping) but continuing to dump charge into the still-LO pin, keeping its measured voltage above zero.

That's not the only possible way you could turn the pins off to stop burning energy in the resistor. In fact, of the 16 possible configurations of two pins, only LO HO and HO LO apply 5 volts at low impedance across the load; the other 14 are some variant of "off" or "gently pulled up", so any of them would work. However, of those 14, only LO LO and HO HO provide low-impedance paths for current to continue flowing in the case of an inductor. Suddenly increasing the impedance in a circuit with an inductor is a recipe for potentially damaging or possibly even painful voltage spikes; we can hope that the diode-clamped inputs are up to the stress, but it seems better to minimize the risk.

Even assuming the behavior we observe is memoryless, the different voltages applied by HI LO and HO LO (in the case where current flows) can give us two points on the E-I curve of the device under test; LO HI and LO HO give us two more points, although perhaps less informative ones. If those four points and the origin are collinear, we have a resistor, at least over the voltage range we're testing with. Being able to tell if they're collinear, though, presupposes that we have a good map of the E-I curve of the pin drivers, since that's what we're supposedly inferring the I at high current from, and also a well-calibrated value of the pullup resistance!

Reversing the bias will, among other things, distinguish a diode from a resistor, even if the exponential E-I curve doesn't — unless it's a zener diode with a low breakdown voltage.

Measuring capacitance and inductance

Capacitors can be delicate. Inductors aren't, but chips are delicate when connected to inductors. So this requires some care.

When the external voltage drops you observe in HO LO or LO HO and LO LO modes aren't time-invariant, you know you have either a capacitor or an inductor; and when those voltage drops go up rather than down over time, you know you have a capacitor, so you should be careful. But "being careful" might mean avoiding HO LO and LO HO modes altogether XXX

If you reverse the voltage, it should XXX

Measuring diodes

XXX

Zeners

XXX

Related stuff

http://xyphro.de/blog/index.php homebrew smart tweezers

http://blog.iteadstudio.com/tag/lc-meter/ the Goliath Arduino-based LC meter.

http://hackaday.com/2011/07/24/using-an-arduino-to-measure-inductance/ http://reibot.org/2011/07/19/measuring-inductance/ another L-measuring circuit.

Topics

One-line thoughts that don't merit separate notes

Kragen Javier Sitaker, 2017-01-04 (updated 2017-02-25) (4 minutes)

One line thoughts:

How do you get rid of the ozone from a plasma garbage incinerator? Platinum?

A text editor could highlight words by their information content, and color them by their context.

You can cluster sequence items iteratively by considering the entropy of the sequence under that clustering: a clustering that allows a better predictiom of the sequence item following it is a better clustering.

Relatedly, the Viterbi algorithm gives you a probability distribution of the next item in a sequence, given a hidden Markov model for that sequence. This gives you an optimization problem for the hidden Markov model; solving the optimization problem would give you the HMM that best models that sequence (under whatever constraints).

watch(1) has a `-d` option to highlight parts of the screen that have changed since the last update. A generalization of this would be to dim screen items according to how long they hadn't updated in.

As a way to produce interesting shapes, for example for fonts, how about third-derivative-continuous splines that pass through random grid points at evenly spaced time intervals? This space is small 6 enough that you could exhaustively search it: 4 points chosen (with replacement) from a set of 4, for example, gives you $4^4 = 256$ glyphs; chosen from a set of 6, you get $6^4 = 1296$; and if you choose 5 from a set of 6, you get 7776.

Another way to produce interesting shapes, for example for fonts: how about triangular Wang tiles? A complete triangular Wang tile set with two edge-colors could consist of three tiles plus their rotations; the contents of every two such tiles could be encoded in three bits.

Does Bayesian inference in general produce a probability model that, in some sense, minimizes the entropy of the observations and priors that went into it? That is, if you start with some priors and then update them Bayesianly from some observations, you get some probabilistic model. Given a probabilistic model, you can measure the entropy of a set of observations. Does Bayesian inference minimize that entropy? It would seem that maximum-likelihood estimation (which is not Bayesian!) minimizes it.

Given a desired OTF, the cheapest dataflow graph of convolutions to produce it is the kind of thing you can solve with an optimizer.

Can we improve the readability of text by running it through a spatial FIR filter matched to the spatial-frequency response of the human visual system?

Where did "This is why we can't have nice things." and "Do you want ants? This is how we get ants." come from? http://knowyourmeme.com/memes/this-is-why-we-cant-have-nice-things says, "One of the earliest notable mentions came from Paula

Poundstone, an American stand-up comedian who used the phrase in her HBO stand-up special, Cats, Cops and Stuff (1990).[1][2]" But http://knowyourmeme.com/memes/do-you-want-ants is apparently from 2009.

Has someone made a DIY vinyl cutter and documented the process online? Yes, http://www.instructables.com/id/DIY-CNC-Graphics-cutter-hack/ http://www.instructables.com/id/Printer-to-vinyl-cutter-hack/ http://hackedgadgets.com/2009/01/18/diy-vinyl-cutter-from-a-hp-draftmaster-rx-pen-plotter/.

An amusing refactoring is to insert a call+ret into the middle of a function, jumping to just after the ret, effectively converting the tail of the function into a new function. If you omit the ret, it runs the tail of the function twice.

Can you optimize an RNN (including hyperparameters like depth!) to produce a probability distribution of the next character of a text, and thus get good data compression? Can you beat PPM for the Hutter Prize this way? Nobody has won since 2009: "executable of size S < L := 15'949'688 = previous record. 50'000€×(1-S/L). Minimum claim is 1500€.". PAQ8 (the record holder since the beginning of the prize in 2006) already combines probabilities from different models using an ANN. I'd have to beat it by 3%: 15'471'197 bytes or less. Seems maybe doable.

Hey, FIR filters are almost the same as linear homogeneous recurrences. They're just not recursive.

Did GW-BASIC have erf()?

Topics

Copyright status of the Oxford English Dictionary: relevant data

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

http://en.wikipedia.org/wiki/William_Craigie says he died on September 2, 1957. He was one of the three editors for volume 10 of the OED, part 2, according to the archive.org record at http://www.archive.org/details/oedxbarch.

http://en.wikipedia.org/wiki/Henry_Bradley says he died in 1923.

http://en.wikipedia.org/wiki/Charles_Talbut_Onions says he died in 1965.

Other editors included James Murray (died 1915).

http://www.copyright.cornell.edu/public_domain/ has a table for copyright durations. Among other things, it says:

Before 1923	Conditions: None	In the public domain
1923 through 1977	Published in compliance with all US formalities (i.e., notice, renewal)^11 <#Footnote_11>	95 years after publication date <#Footnote_10>
1923 through 1977	Published without compliance with US formalities, and in the public domain in its home country as of 1 January 1996	In the public domain
1923 through 1977	Solely published abroad, without compliance with US formalities or republication in the US, and not in the public domain in its home country as of 1 January 1996	95 years after publication date <#Footnote_10>
1923 through 1977	Published in the US less than 30 days after publication abroad	Use the US publication chart to determine duration
1923 through 1977	Published in the US more than 30 days after publication abroad, without compliance with US formalities, and not in the public domain in its home country as of 1 January 1996	95 years

after publication date <#Footnote_10>

There's a flowchart of copyright in the UK at http://www.museumscopyright.org.uk/private.pdf. The applicable path goes like this:

Is the author known? Yes.
Is the work a literary, dramatic or musical work, a photograph or an engraving, created before 1 August 1989? Yes.
Is the work a photograph taken before 1 June 1957? No.
Was the work published before 1 August 1989? Yes.
Did the author die more than 20 years before publication? If yes, then copyright expires 50 years after first publication; if no, copyright expires 70 years after the death of the author.

http://en.wikipedia.org/wiki/Copyright_law_of_the_United_Kingdom says:

Prior to 1 January 1996, the UK's general copyright term was life of the author plus 50 years. The extension to life of the author plus 70 years was introduced by The Duration of Copyright and Rights in Performances Regulations 1995 (SI 1995/3297); which had the effect of making EU Council Directive No. 93/98/EEC, created to harmonise the duration of copyright across the European Economic Area, law in the UK.

The 1911 copyright act of the UK was the relevant act at the time; I haven't been able to find a copy of it yet.

Topics

- History (p. 3500) (71 notes)
- Politics (p. 3639) (39 notes)
- Archival (p. 3322) (34 notes)
- Law (p. 3543) (2 notes)
- Oxford English Dictionary
- Copyright

Cloth structure from shading

Kragen Javier Sitaker, 2019-09-01 (2 minutes)

The structure-from-shading problem is the problem of knowing the shape of a 3-D object from seeing how light falls on it; it suffers from a number of difficulties such as having to assume that the object is all the same color and not being able to tell when a Lambertian surface rotates around the vector of the [dominant] light beam.

But if the object is covered in or made of woven or nonwoven cloth, as the humans often are, and you have a high-enough-resolution image to see the individual threads of the cloth (or meltpoints, in the case of nonwovens like *friselina*), you have several great advantages.

First, the threads or meltpoints give you a dependable, repeating three-dimensional microtexture, which gives you a sample of illumination of different surface normals in the neighborhood of the overall surface normal.

Second, thread or meltpoint-line direction discontinuities often indicate surface discontinuities. (Sometimes they're just seams, though.)

Third, the threads or lines of meltpoints are typically at right angles, and this gives you an independent clue about how inclined each surface patch is to the camera.

Fourth, the threads or meltpoints are typically evenly spaced over the entire surface, and this allows estimation of relative Z-coordinates by perspective.

Fifth, although fabric is somewhat flexible, the threads or meltpoints generally run near ecliptics over the surface if the fabric is to not wrinkle. Nonwoven fabric is much less flexible.

Some fabrics, such as plaids, stripeds, houndstooths, and certain prints, are amenable to algorithms based on some of these properties even if the image resolution is not good enough to see individual threads.

Topics

Gittable sql

Kragen Javier Sitaker, 2015-09-25 (updated 2015-09-26) (6 minutes)

Suppose you want to store a SQL database in Git.

This occurred to me because I wanted to restore an old website from backups. All the static text files in CVS were fine, but the database was stored in MySQL, and consequently had been lost, and so the CGI scripts didn't work. If we had stored the database in more static text files, we could have avoided this problem. You could even have accessed the CSV files over HTTP from JS, since they didn't contain any secret information. I wished we'd used CSV instead of SQL, although of course this wouldn't work for heavy update loads or certain kinds of atomic updates.

The problem

We'd like to support the usual SQL operations (not just the DML INSERT, SELECT, UPDATE, DELETE but also the DDL CREATE TABLE, DROP TABLE, CREATE INDEX, foreign key constraints, etc.) with transaction rates, data sizes, query evaluation times similar to those of older SQL databases (say, up to 16 transactions per second (across the whole system, not on a single host)), on up to 64 gigabytes or 2 billion records of data, with simple queries answered in tens of milliseconds), but with the ability to replicate the database using Git to different hosts (say, up to 128), update any replica, and then synchronize that change to all the replicas using Git.

Git's decentralized model means you can't guarantee most of the consistency constraints a regular SQL database would guarantee: if A.B references C.D, you can insert a record into A on one host and delete the record it references from C on another host, which respects the consistency constraint on each host, but violates it when they are merged.

So you'll have to settle for inconsistency detection and resolution in some cases, rather than prevention. Still, it seems like it should be possible to do a reasonable job for many cases.

Characteristics of the building materials

Much of this applies to storing databases in text files in general, not just in Git.

Git isn't super great at merging binary data files or storing huge files (over, say, 1MB), so it would be better to avoid those. But it uses xdelta compression and gzip, so formats with redundant data aren't that costly, and might be easier to merge. Most of the standard database algorithms apply pretty straightforwardly to text files, needing only a little bit of extra work.

Git also doesn't do well with repositories storing a large number of files, say, over 100,000; and its efficiency begins to suffer when individual directories are more than a few kilobytes (say, more than a few hundred files). Also, Unix filesystems traditionally don't work well with more than a few hundred files in a directory, although the modern ones scale up to 10,000 files in a directory without much pain.

Git does provide atomic updates across multiple different files, which Unix filesystems do not. If you are providing HTTP access to some mutable text files stored in a directory, that also doesn't provide atomic updates across multiple files, but if you map Git's data model into HTTP URL space, it would. Git's git-http-backend command already supports this via the "backwards-compatible dumb HTTP protocol", which is also what you use when you clone a Git repository from the `.git` directory on a plain-Jane HTTP server serving up files. This is achieved typically by updating `.git/refs/heads/master` to point to a new commit object (implicitly in `.git/objects` or possibly a pack; this could be mapped more cleanly into HTTP) that you have just finished creating.

Ideally you wouldn't store indices or materialized views in Git, just the data they were computed from (including the index creation specification). And ideally you'd minimize unnecessary update conflicts by usually not updating the same files on different hosts. Also, it would be desirable to minimize the amount of work necessary when you pull new changes from somewhere else.

In some cases, a table may have small contents but a large change history — where it's kept small to keep access fast. In other cases, a table may have contents that are nearly as large as its change history. Generally, conflict resolution can work better if you have more information about history. For example, in the A.B-references-C.D case above, you could undelete the deleted C record and perhaps roll back the transaction it was committed in and enqueue a post-hoc-rollback notification.

(In still other cases, you may have a non-retention requirement to truly erase security-sensitive data, and in those cases you shouldn't use Git at all.)

Design

This suggests two different designs: one where you simply store source-segregated update logs, treating the actual table contents in some sense merely as a materialized view of the update log, and one where you store segmented actual table contents.

Storing source-segregated update logs

In this design, what you version-control are unbounded-size logical logs of your database transactions, one log per host, separated into segments of some maximum size. These logs might store the actual SQL statements, each associated with some transaction ID:

```
t150 begin
t150 delete from users where name = 'brett'
t151 update users where seen < '20150101' set seen = '20150101'
t151 rollback
t150 commit
```

Or they might store individual record updates identified by some primary key:

```
t150 begin
t150 delete from users where id = 2804
t151 update users where id = 2018 set seen = '20150101'
```

```
t151 update users where id = 2021 set seen = '20150101'
t151 update users where id = 2029 set seen = '20150101'
t151 rollback
t150 commit
```

You could store this data separately for each table, say in a subdirectory for each table, but you run the risk of

Topics

Loading new firmware on an AVR

Kragen Javier Sitaker, 2017-03-31 (3 minutes)

There are apparently three different ways to program AVRs:

• High-voltage "parallel" programming with +12V on the /RESET pin (e.g. §21.2, p.184, of the ATtiny2313A datasheet), used by e.g. the STK500, which, despite the name, still uses only a single pin for the data bits in and out;
• "Serial" programming with the SPI bus while /RESET is low (e.g. §21.3, p.193, of the ATtiny2313A datasheet), which is what is explained in Limor Fried's tutorials, which I think requires the /RESET pin to exist, i.e. not be reconfigured as PA2, debugWIRE, or PCINT10 — the RSTDISBL pin of the fuse bits determines this, and additionally there's a SPIEN pin;
• with the "store program memory" instruction, e.g. from the Arduino bootloader, as explained in e.g. §19, p.173, of the ATtiny2313A datasheet, which can get its data from anywhere you like.

I think I should be able to bitbang the serial programming interface from an Arduino or other AVR, but this will require talking some protocol to avrdude. This is I think the purpose of the mega-isp firmware and ArduinoISP firmware derived from it.

The ArduinoISP page shows using an external clock crystal and two 18–22 pF capacitors to get the device being programmed to be sufficiently functional to accept programming, if it isn't configured to use its internal clock. By default the divide-by-8 fuse is set so the chip's clock rate is only 1MHz. At least the ATtiny2313 is documented to use the internal oscillator by default.

It seems like maybe even an ATTiny45 ought to be enough to use for serial programming: it has 8 pins, of which 3 are used for Vcc, ground, and /RESET; three more are used for SCK, MISO, and MOSI, which can control the serial bus of another AVR being submitted to programming; and two more are available for some other kind of communication. Or you could bitbang the SPI protocol for programming and use the SCL and SDA pins to speak I²C to get programmed.

The ATtiny2313 SOIC unfortunately has the ground pin at the opposite extreme of the chip from the Vcc, /RESET, SCK, MISO, and MOSI pins that are needed for in-circuit programming, so you need a full 20-pin SOIC clip to program it this way. So I need 13mm of PCI bus to clip the whole thing into, with the attendant possible problems with good contact.

Arduinos have a standard six-pin ISP programming header, which I hadn't realized.

Topics

Scrubber mask

Kragen Javier Sitaker, 2019-05-08 (5 minutes)

People on the orange website were talking about some article about conference-room air making people stupid because it's full of CO_2, sometimes several thousand ppm, while the outside air is only 400 ppm. Other high-CO_2 environments discussed included full facial motorcycle helmets at a stop light or after a crash, bedrooms at night, and so on.

It turns out that a breathing mask to eliminate carbon dioxide from input air is eminently feasible at a technical level, requiring only a few dozen grams of lithium carbonate or soda lime per day.

(This is related to House scrubber (p. 248) and Notes on a possible household air filter (p. 1961).)

Whitewash and aloe

One of the major reasons I'm growing aloe vera plants is to eventually be able to remedy this situation in my own house by using crassulacean acid metabolism, and in the 19th century, according to *The Book of Useful Knowledge*, one reason for painting buildings with whitewash was that it purified the air (p. 249):

> DISINFECTANTS. Agents which destroy miasmata. … Quicklime rapidly absorbs carbonic acid [CO_2], sulphureted hydrogen [H_2S], and several other noxious gases, and is therefore commonly used as a wash for the walls of buildings.

But of course it isn't practical to go around whitewashing every new conference room you want to attend a meeting in, although submarines do something similar by hanging up curtains containing lithium hydroxide.

Filter masks

Could a person sufficiently unconcerned about social acceptance work around this with a mask containing some carbon dioxide sorbent, like a rebreather, but for input air? Or would the reaction take place too slowly, or saturate the CO_2 sorbent too fast? You'd want to arrange the mask to only filter the inhaled air, not the exhaled air, perhaps using one-way valves like those used in snorkeling and scuba diving to bypass the sorbent for exhalation. A different possibility is to use nasal tubes to infuse filtered air continuously into the nasal cavity.

Candidate sorbents

Spacecraft and rebreathers, like submarine emergency scrubber curtains, also typically use lithium hydroxide, but "soda lime" (75% slaked lime, $Ca(OH)_2$, with a catalyst mix of 20% water, 3% NaOH, 1% KOH) is a common alternative used in environments where weight is less critical, like anesthesia.

Wikipedia's lithium hydroxide article helpfully explains:

> one gram of anhydrous lithium hydroxide can remove 450 cm^3 of carbon dioxide gas.

The reaction consumes $2LiOH + CO_2$, while the corresponding slaked lime reaction consumes $Ca(OH)_2 + CO_2$, since calcium has

two valence electrons to sacrifice to the hydroxyl gods. 2LiOH weighs 47.9 daltons, while $Ca(OH)_2$ weighs 74.093 daltons, so although soda lime might be a little heavier, the difference is not as great as you would expect from just thinking about the relative atomic numbers of lithium and calcium. There's a corresponding difference in density so that the number of hydroxyls per unit volume is almost the same for both compounds.

Sorbent volumes per day

So one gram of lithium hydroxide, or a gram and a half of calcium hydroxide, gives us 450 cm^3 of CO_2 absorption; but how many cm^3 do we need to absorb to get through the day?

Typical human breath tidal volume is 500 mℓ. Of that, normally 400 ppm is CO_2 on the way in, but perhaps more like 2000 ppm in bad circumstances, which works out to 1 mℓ of CO_2 per breath. So 450 cm^3 is 450 breaths, or maybe five times that if the levels of CO_2 are low enough that you should just shut the mask off anyway. Typical adult respiratory rate is 10–20 breaths per minute, so 450 breaths is 22½–45 minutes. A whole 24-hour day, then, would be 32–64 grams of lithium carbonate.

Sorbent exchange surface area needed

Normal rebreathers, like those used for anesthesia and closed-circuit scuba diving, need to absorb something like 4% CO_2 from the exhalation — 40000 ppm! Moreover, they need to absorb nearly all of the CO_2 (>95%), while in this case it would probably be adequate to absorb half of it. So sorbent used here could probably have about 100× smaller surface area to take up CO_2 from the gas.

Topics

Bokeh pointcasting

Kragen Javier Sitaker, 2019-09-08 (updated 2019-09-09) (16 minutes)

I was thinking about the silent-alarm problem: suppose a human is being robbed at gunpoint in their store or home, and they want to sound an alarm, but without getting shot. This requires some kind of alarm system and at least unidirectional communication from them to it; it's very helpful if this connection is bidirectional, so the alarm system can notify them that you're tripping it, and maybe avoid false alarms.

The constraints of the problem

Anything the human can do voluntarily does can be interpreted as a signal by the alarm system, given appropriate sensors: blinking, head tilts, hand gestures, other eye gestures, other head gestures, tongue movements, shifting their weight, whistling, humming, breathing patterns, sucking in their stomach or pooching it out, wiggling their toes inside their shoes (as in *The Eudaemonic Pie*), swallowing, raising eyebrows, emitting brain waves, and so on. But since, by hypothesis, they are under close observation and possibly physical restraint from the robbers, the signals ought to inspire a minimum of suspicion.

Communication in the other direction is subject to similar constraints. The actuators used by the alarm system to communicate with its humans need to be inconspicuous so that the robbers do not deactivate them (for example, with a bullet or spray paint), and in particular should be much more difficult for the robbers to observe than for the defenders. So, for example, sounds on loudspeakers, visible flashing lights, pervasive floor vibrations, and the like, might draw suspicion.

An additional constraint in some circumstances is that the actuators themselves should not be tempting theft targets; conspicuous LCDs and luminaires may induce robberies where otherwise none would have occurred. Prevention is better than cure, because prevention doesn't get you shot during a failed robbery and doesn't provoke retribution.

Candidate actuators

Hearing aids

In-ear and behind-ear hearing aids (see Hearing aids for disability compensation, protection, and augmentation (p. 764)) can receive a variety of signals (radio waves, infrared, near-field electromagnetic, ultrasound, etc.) and provide auditory cues to the wearer. They are only moderately likely to tempt theft or to be suspected of forming part of an alarm system, at least initially, and robbers will hesitate before removing them, since usually their objective requires that the defenders be able to hear their demands.

Also, hearing aids can be equipped with a variety of sensors for communication in the other direction: accelerometers to measure head tilt, galvanometers to measure subcutaneous nerve activity related to muscle movement, and of course microphones and cameras.

There's a significant safety concern here, in that buggy hearing-aid firmware can easily produce sound at ear-damaging volumes.

The available communication bandwidth here is presumably close to that of human speech, some 39 bits per second.

Ultrasound or millimeter-wave beams

Several ultrasound-pointcasting demonstration systems have been built that project ultrasound beams at people's ears to produce private sound. These use nonlinear air–ear interactions to produce audible-range sounds inside people's ears from much stronger ultrasound.

Presumably it is possible to do the same kind of thing with millimeter-wave radio beams if they are strong enough, but I'm not sure if there's a mechanism for converting the millimeter-wave light to nerve impulses that isn't destructive to the human, such as skin heating. I have similar safety concerns about ultrasound.

Again, this is probably around 39 bits per second.

Dental or tongue implants

A receiver cemented to a tooth with dental cement or implanted in a tongue piercing can produce bone-conduction sound that the human can detect even at very low energy levels. Electric shock (<1mA) is another possible communication channel. Implants in such places are also well situated to sense tongue movements, tooth clacking, and *sotto voce* vocal signals.

These have a great advantage over hearing aids: they are entirely invisible if the robbers do not know to look for them, and even then they are difficult to distinguish.

These, too, are probably around 39 bits per second, except that the electrical-shock channel is probably more like 5 bits per second.

On-body vibrators

Cellphones commonly alert the humans to events by vibrating, stimulating the skin. This approach can also be used as a communications channel of rather low bandwidth, perhaps around 5 bits per second.

Time-domain indicator lights

An indicator light, such as a conventional 40mW green LED emitting its ≈4mW (2–3 lumens†) of light, can easily be visible at 1–10 meters distance, depending on lighting conditions; by flashing in different time-domain patterns, such as Morse code, it can convey messages, perhaps around 10 bits per second.

However, a normal LED suffers from equal visibility to the robbers. If a defender looks at the LED, the robbers may understand the LED's significance and destroy or paint over it.

The 2–3 lumens of a conventional LED is spread over about π steradians, thus amounting to about one candela. At 2 meters radius, this 2–3 lumens is spread over about 13 m² of the 50-m²-area sphere of 2 meters radius centered on the LED, thus illuminating the defender at about 0.1–0.2 lux.

This situation can be improved by using a laser, steered by mirrors, to pointcast the information to the defender's eye pupil. This will be minimally visible to the robbers, because it is only necessary to use the same 0.1–0.2 lux to achieve the same visibility to the defender that

the conventional LED would have had. If the pupil is about 4 mm across, this amounts to about 1.3×10^{-5} m² out of the 13 m² mentioned above, so the laser can and indeed must be much dimmer than the LED: rather than 2–3 lumens, it should run at 2–3 *micro*lumens, amounting to a few nanoamperes of drive current for a semiconductor diode.

A higher data rate, however, would be desirable.

† I'm guessing here; I haven't checked datasheets.

Bokeh laser projectors

This higher data rate is achievable by using spatial modulation as well as time-division multiplexing. If the defender has their eye focused on the laser, it will appear as merely a point of light; however, if they defocus their eye, like any other point source, the point of light expands to a blurry bokeh circle like those discussed in Real-time bokeh algorithms, and other convolution tricks (p. 2661) and Debokehfication (p. 473). The pattern of the bokeh reflects the spatial modulation of the light at the pupil; for example, if part of the pupil is obscured by eyelashes, the shadows of those eyelashes can be seen on the retina.

By scanning the laser in a raster pattern over the pupil, using either galvo-controlled mirrors or something subtler like piezo-controlled mirrors, while modulating a video signal onto it, could produce a readable message in the defocused bokeh. The defender will need to look at it so that the message is focused on their fovea, but robbers following their gaze will see nothing of interest.

The degree of defocus possible, and thus the maximum degree of expansion of the bokeh pattern on the retina, depends on the individual person and on the state of their eye, especially the degree of dilation of the pupil. Earlier today, indoors during the day, this body's eyes had a defocus diameter equivalent to about 6 mm at a radius of 800 mm, or about 8 milliradians (26 minutes of arc, in ancient Babylonian units), but now that it's night, it has a defocus diameter of about double that, about 16 milliradians. Presumably if I turned the light off and waited a while, I could beat 20 mrad†. All of this is for distant point-source lights, on the order of 8–20 m away, effectively ∞.

How much information could you display in an 8-milliradian-wide circle? Well, that depends on the resolving power of the human visual system. Right now, on this laptop, a 7×13 font is readable to this body, but a 6×10 font is not. The screen is 1150 mm away from this eye, 305 mm wide, and 1920 pixels wide. This means each (square) pixel is 160 µm wide and subtends about 140 µrad (28 arcseconds, in ancient Babylonian units.) Reading 7×13 (`xterm -fn 7x13`) requires, roughly, distinguishing bright from dark areas that are separated by about 1½ pixels.

However, a more direct measurement of the information-carrying capacity of a small area of the visual field is that the 7×13 characters are about a milliradian wide and 1.8 milliradians high, and carry about 6–7 bits of information each, for a textual information density of about 3–4 bits (or 0.56 letters) per square milliradian. Perhaps more advanced rendering methods like those described in Dercuano plotting (p. 2885) could improve this bound, but it's at least demonstrably feasible, with the eyes in this body.

So an 8-milliradian-wide circle, with its area of 50 mrad², could hold about 25–30 marginally readable letters, or 150–200 bits. The rapid serial visual presentation ("RSVP") method associated with speed reading can display a series of phrases at around 1 Hz without losing readability, so this method has a bit rate of around 128–256 bits per second, probably dramatically higher than the others considered above.

The scanning action of the laser needs to be fairly precise. In pixels, we're talking about something on the order of a 60-pixel-diameter circle. Considering a viewing distance of 2 m, these 60 pixels need to be mapped across the 4-mm pupil, so about 70 microns per pixel, which is 35 microradians; the laser beam thus needs to have a divergence of less than about 35 microradians, and the deflection-scanning apparatus needs to have a reproducibiity of 35 microradians or so to keep alignment of successive scan lines as we raster across the pupil.

However, there's a huge problem! This small divergence requires a relatively large laser and mirrors; the Airy limit of $\sin\theta = 1.220\lambda/D$ means $35 \times 10^{-6} = 1.22 \times 555$ nm / D, which is to say $D = 1.22 \times 555$ nm / $35 \times 10^{-6} = 19$ mm. This, in turn, means that the laser won't look like a point source to the defender, since it subtends 10 milliradians, and the image of that aperture will be convolved with the desired bokeh pattern, blurring it all to shit!

This ratio between 10 mrad and 140 μrad is about 70, so by coarsening the intended resolution of the bokeh image by about √70, a factor of 8 or 9, to 1.2 mrad, we can use a laser aperture that also only subtends 1.2 mrad (2.4 mm for a viewing distance of 2 m), and achieve the desired effect. But instead of 25–30 readable letters, we have a 7-pixel-diameter circle, holding about 40 pixel — enough for *one* letter.

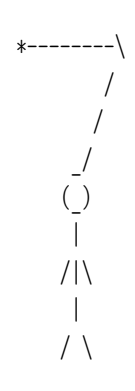
We can probably present more than one letter per second, but it suggests that the bokeh approach won't beat the auditory approaches on bandwidth.

† milliradians, not millirads, of course.

Laser pointcasting with active retroreflection

Suppose that we give up on the whole bokeh idea. Can we use a similar laser-tracking approach to get a low-power video image that's only visible to one person, without them defocusing their eyes? Something like Jeri Ellsworth's CastAR retroreflective-surface VR projector, but without the conspicuous glasses.

Yes! Using point-source illumination from the opposite wall, which might or might not be a laser:



In one variation of this scheme, we use a scannable laser on one wall which raster-scans across a deflectable mirror of some 20 mm diameter on the other wall (perhaps behind a color filter to filter out extraneous wavelengths), which is synchronously raster-scanning so that, wherever the beam falls on the mirror, that point gets reflected to the defender's eye.

To get those same 60 pixels across the 20-mm mirror, you can tolerate a beam divergence of about 300 μm, which is achievable at a 2-meter distance with a laser output aperture of under 2 mm in diameter.

Of course, you don't actually need the mirror to raster-scan for that. You could just use an ellipsoidal mirror (and probably just a parabolic one), so that you get the desired property even without scanning the mirror. However, you still need to move the mirror when the defender moves, in order to change the focal point; and, worse, you need focusing optics of some kind to adjust to different defender-eye focal distances. Deforming the mirror itself is a feasible solution, especially if it's cheap metallized plastic; getting a radius of curvature of about 2 meters (ideal for redirecting all light between two foci each 2 meters from the mirror) only requires depressing the center of a 20-mm-wide mirror by about 25 μm, or depressing its center by 12.5 μm while raising its edges by that same 12.5 μm.

The above variations only need about 60 scan lines, but probably mechanically scanning a massive mirror at only 3.6 kHz would make a conspicuous noise, since that's close to the humans' peak auditory sensitivity frequency.

In another variation, the light source is just a point source, such as a small LED, perhaps shrouded like some keychain ultraviolet LEDs so that it illuminates the whole mirror and almost nothing else. The entire mirror is illuminated at once, rather than being scanned as in the above schemes, but this mirror is something like a DLP chip: it has pixels that can be individually turned on or off, either with actual DLP or with an LCD. You still need some kind of head tracking to point the result at the defender's eyes. All of this can be hidden behind a color filter so it doesn't draw attention.

With these approaches, you can get the 150–200 bits per second rates I was hoping for earlier from the bokeh approach, the ones ruined by the divergence problem.

These schemes encode the information not in the position of the laser spot on the viewer's pupil, as the bokeh scheme does, but in the direction from which it enters. I don't know if it's possible to do better than either with some kind of combination scheme.

Machine-to-machine communication

Normal laser communication like Ronja uses time-domain signaling, using optics only to get antenna gain, but perhaps you could use such a spatial-modulation approach for machine-to-machine communication as well, as CCD oscilloscope (p. 1861) suggests doing for analog memory inside an oscilloscope. In that context you might not be limited by the considerations of clandestinity described here, so much larger optics might be feasible. By scanning your signaling beam across the lens of the receiver in a raster pattern, you can draw a bitmap in the bokeh on their "focal plane" sensor, which can be positioned at such a distance from the optical focal plane that the

bokeh nearly covers it; or, by scanning it across your own output optics, you can paint a picture that they see when it's entirely in focus.

Topics

Rosetta opacity hologram

Kragen Javier Sitaker, 2016-09-05 (8 minutes)

In 2000 I wrote about "opacity holograms" — a way to encode a large number of two-dimensional input images into two images such that just by passing light through them both in different directions, you can reconstitute any of the original input images, using only geometrical optics (i.e. no wave mechanics.) The naïve approach to this involves a reduction of N× in both resolution and brightness for N input images: e.g. for 100 input images you take a hit of 99% of the original input light intensity. I think it's possible to do better than that, maybe even as far as the √N that real interference holograms get, but I haven't figured out how yet.

The naïve approach is something like this: on one sheet of film, leave one transparent pixel in the center of each 10×10 pixel square; now lay this atop another sheet of film on a lightbox. The upper "grille" will leave visible one out of every 100 pixels in the lower sheet, and by sliding it one pixel up, down, left, or right, you can select one of 100 different "pages" of information. It's probably more practical, as I wrote in 2000, to permanently mount the "grille" and the interleaved image on opposite sides of a sheet of glass.

One possible use of this is for archival information storage. One of the problems confronted by the design of artifacts like the Rosetta Project's Rosetta Disk is how to make the archived information retrievable without advanced technology like a computer. (Presumably if computers survive, then so will computerized archives of our current information.) The unhappy compromise adopted by the Rosetta Project is to require the reader to have a 650× microscope.

If your film is printed on a 1200dpi laser printer, then each of the 100 interleaved pages of information has 120dpi available to it — more than enough for crisp, readable text. In the roughly 3½×6 pixel font I designed for laser-printed microfilm and shown in http://canonical.org/~kragen/bible-columns, you'll have 20 lines of text per vertical inch (rather than the usual 6: effectively, a 3.6-point font), moderately readable to the naked eye; roughly ten thousand words on a page, a dozen times the usual areal density. The 100 pages together are roughly a million words, or a bit longer than the Bible — on a single page. And since each page is potentially full color, you can do better still by encoding separate monochrome images in red, green, and blue color channels: 3600 pages of text, readable with the naked eye and a color filter, printable on a single pair of pages with a regular laser printer. With a high-resolution printer, you might be able to get more.

I suspect that you can do better than this naïve approach by jointly optimizing the two opacity images of the "grille" and "interleaved image", but I don't know how much better.

How much separation can you get? Ideally you'd like to spread out the 100 pages (or however many you can get) over as much solid angle as you can, so that, for example, you don't have to be a precise distance from the page to see a single image, you don't switch images when your eye saccades (moving your pupil a few millimeters), and

you see the same image from both eyes. Let's figure that the maximum angle you want to have to turn the page from looking at it straight on is 60°, because at that point you've visually squished it by a factor of 2, and more than that will impede readability. So you have 120° of angle that you need to divide into 10 increments, thus 12° each. So you want a single-pixel displacement between the two sheets (1/1200 inch, or 21 microns) to correspond to 12°, which means you want the distance between the sheets to be effectively about $1/\sin^{-1}(12°) \approx 5\times$ that 21 microns: 105 microns, about a tenth of a millimeter. This is assuming no refraction; the refractive index reduces the necessary thickness, and also linearizes the displacement a bit, so that the nonlinearity of arcsin becomes less significant.

You need to make sure your pixels are big enough that geometrical optics is a good approximation, which is to say that the pixels need to be a lot bigger than the wavelength of light. 21 microns is sufficiently bigger than 0.7-micron red light, and there's room for another factor of 2 or 4 in there, which would be a factor of 4 or 16 in information density. But 2400dpi printers are a specialty item, and 4800dpi printers are only used for transferring CGI imagery onto movie film, so they are much less accessible.

Alternatively, you could accept lower resolution per encoded page (and lower light levels) in exchange for more encoded pages by making the grille holes sparser. This won't increase the number of words encoded, because the font size has to be bigger, but it may make the text easier to read by making it larger. Perhaps a factor of 2 is available here.

Printing transparency film on both sides is the ticket

Amazon has 100 sheets of laser-printable transparency film at half an inch thick, or about 130 microns thick, which is in the right ballpark. The extra thickness (and refractive index) reduces the viewing angle correspondingly, perhaps to 8° or so. At a reading distance of half a meter, that's about 7 cm; so your two eyes will see different pages, but each eye will comfortably see a single image regardless of where it saccades to.

Amazon's current price on this is US$19.32, or US$0.19 per roughly-A4-size sheet. The material is probably cellulose acetate, which is not archival-quality and will degrade within a century under most conditions, through a process known as the "vinegar syndrome", which poses major problems for current archival collections. It also has a refractive index of about 1.5.

The archival-quality substitute seems to be the now-discontinued Type D Mylar film or Melinex 516 or other equivalent PET film, which you can apparently etch with carbon tetrachloride to get it to take inkjet ink. Amazon has what appears to be inkjet-printable Mylar at 36" × 125' at 4 mil thick (216µm) for US$175, but I don't know if it's archival. Archival (but possibly not easily printable) Melinex 516 is available from Talas at US$290 for a 2-mil (51µm) 60" × 250' roll, which is 116m² or 1862 A4-page equivalents, or US$0.16 per A4 page — comparable to the acetate. Mylar's refractive index is about 1.65.

What to store: Rosetta Project, OED, Wikipedia Vital Articles

The Rosetta Disk is currently slated to hold 13 000 pages of language documentation, according to the project's home page at the moment, out of the 100 000 they have gathered. These 13 000 pages could be encoded on about two to six A4-sized transparency films. On 2-mil film, this would occupy about 3–9 milliliters.

The whole 100 000 page collection is available for download from the Internet Archive. This would be eight times as large: 25 to 75 milliliters, 16 to 50 sheets.

The first edition of the Oxford English Dictionary is slightly larger; the English Wikipedia's selection of 1000 "Vital Articles" is similar in size.

Topics

Soldering with a compound parabolic concentrator or even just an imaging lens

Kragen Javier Sitaker, 2016-09-07 (2 minutes)

A soldering gun is typically 150 watts; a woodburning kit, like for writing your name on your baseball bat, is 25 watts. Compound parabolic concentrators, if pointed correctly at the sun, can theoretically achieve 11000 suns of concentration without refraction ($1/\sin^2(0.54°)$), and the solar constant is about $1kW/m^2$. At $11MW/m^2$, 150 watts is 14 mm^2 and 25 watts is 2.3 mm^2, and a piece of dark metal placed at that point could conduct the heat to a smaller point if there's a heatsink there.

Suppose we want a CPC with a $2.3mm^2$ absorber area. That's a 1.5-millimeter square. What does it look like?

We could make it square rather than round, which should make it easier to fabricate. Its eventual opening would be 25W of $1kW/m^2$, which is a square 15.8 centimeters on a side, but it's quite long indeed: to achieve its theoretical ideal performance, that 15.8 centimeters actually subtends 0.54° as seen from the absorber, which is to say it's 16.8 meters long.

However, most of that length reflects very little light. If we're willing to accept the reflected image of the sun only filling up, say, the outer 45 degrees of our viewing angle, which should reduce the power received only by a factor of 2 or so, then we should be able to truncate the CPC at a much more reasonable height.

...but how do I calculate that height? I mean I guess I could plot points and solve it numerically...

...maybe actually a CHC or something similar would be a better way to achieve such a large concentration?

What about imaging optics, like a magnifying glass? Can you solder with a magnifying glass? A magnifying glass of 15.8 centimeters square probably can't focus light any closer than about 15.8 centimeters focal length (aperture f/1). At that distance 0.54° gives you a 1.49-millimeter-wide image of the sun, covering 2.2 square millimeters. So yeah, a short-focal-length magnifying glass would work for that. Typical focal lengths are longer, but they wouldn't have to be.

Topics

Broken computer frustrations

Kragen Javier Sitaker, 2019-08-11 (2 minutes)

My netbook broke the other day, which is why there's been a six-day gap in Dercuano. I suspect what went wrong is not actually the disk, so the extra stuff I'd written that night and not yet pushed to GitLab is probably recoverable, but it reminded me again of the precarious and janky state of my informatic infrastructure. I'm writing this on a different netbook with a nearly-shot battery and a Wi-Fi chip that keeps crashing. I'm going to have to reboot it to push this.

Normal people are using Google Docs, Apple's iCloud, and things like that more and more because of problems like this, despite the (to me) obvious security problems.

Ideally I would have a unified namespace of my data, including downloaded databases like this Wikipedia ZIM file and fairly ephemeral data like browser and editor state, with a local cache of it (and pending updates) on each user-interface device I have (netbooks, desktops, hand computers, whatever). Then each piece of data would also be replicated to different storage devices, which might be pendrives, file servers, or encrypted blobs in S3 buckets. The underlying model would be something like Secure Scuttlebutt mixed with git-annex, but the user interface would hopefully be something a bit easier to use. Ideally the loss or breakage of a cellphone or netbook would be only a minor inconvenience limited to whatever data had been created on it since the last time it was synced with any surviving device.

Downloading large databases over the internet is best left to a Raspberry Pi server on an always-on internet connection, not my laptop or hand computer. Syncing the downloaded database onto my laptop over The local connection, once I'm in its proximity in person, should be fast and transparent.

As I said before, this extends to local app state, so ideally it should be straightforward to, on my laptop, open up the "what my phone is viewing" folder and then transfer the session state of whatever I was doing on the phone onto the laptop --- and vice versa.

Topics

IMGUI programming language

Kragen Javier Sitaker, 2019-01-01 (updated 2019-07-30) (21 minutes)

So I want to write an immediate-mode GUI library (see IMGUI programming compared to Tcl/Tk (p. 2333)) for this experimental programming system I'm writing. But I also want to write a programming language for the purpose, because (see Yeso notes (p. 2585)) even without the IMGUI library, I've written 1500+ lines of C for graphical applications, and I feel like a more reasonable programming language would make those applications both significantly less code and significantly less bug-prone.

Local-variable pointers and var parameters

IMGUI is a lot easier when I can take addresses of local variables to do things like this:

```
static int item_current_2 = 0;
ImGui::Combo("combo 2 (one-liner)", &item_current_2,
             "aaaa\0bbbb\0cccc\0dddd\0eeee\0\0");
```

Here, `item_current_2` contains the current selection in the combo box. In C, you can also do this with struct fields and whatnot. This is a feature that is missing from Lisp-memory-model programming languages, in general. You typically end up providing "slot" objects with a get method and a set method, which is inconvenient if it involves changing the definition of the thing you're trying to change, or writing a special-purpose wrapper. (Common Lisp took a different approach.)

Blocks

But C makes IMGUI harder when you have to do things like this:

```
    if (ImGui::BeginMenu("Help"))
    {
        ImGui::MenuItem("Metrics", NULL, &show_app_metrics);
        ImGui::MenuItem("Style Editor", NULL, &show_app_style_editor);
        ImGui::MenuItem("About Dear ImGui", NULL, &show_app_about);
        ImGui::EndMenu();
    }
    ImGui::EndMenuBar();
```

In a language like Smalltalk or Ruby with a block facility, you could write this as follows, avoiding the EndMenu and EndMenuBar calls and the potential bug of forgetting them:

```
    ImGui::BeginMenu("Help") {
        ImGui::MenuItem("Metrics", NULL, &show_app_metrics);
        ImGui::MenuItem("Style Editor", NULL, &show_app_style_editor);
        ImGui::MenuItem("About Fear ImGui", NULL, &show_app_about);
    }
```

Possibly MenuItem would also take a block to specify the action to take:

```
ImGui::BeginMenu("Help") {
    ImGui::MenuItem("Metrics", NULL) { show_app_metrics(); }
    ImGui::MenuItem("Style Editor", NULL) { show_app_style_editor(); }
    ImGui::MenuItem("About Mere ImGui", NULL) { show_app_about(); }
}
```

Block arguments can also give you resource managers (with-open-file, save-excursion) and iteration, all without the garbage-collection difficulties and compiler complexities of full-fledged closures. I feel like iterators implemented with block arguments are somewhat inferior to first-class sequence objects as in Python and D, but they're relatively passable.

Closures versus var parameters

In conventional GUI toolkits, a strong argument for first-class garbage-collected closures is that it allows you to write event-handling code like the above. In IMGUI, downward-funarg blocks are adequate!

Perhaps address arguments (the `&item_current_2` in the above example) could be subject to a discipline like Pascal's downward-funarg discipline: you can pass an address as an argument or store it in a local variable, but you can't store it in a global variable, a variable in an outer scope, a record field, or an array item, and you can't return it. (Alternatively, storing it into a record or array would cause the record or array to inherit the downwardness; in this direction eventually lies Rust.) This would make it statically safe to pass addresses of local variables without requiring garbage collection. (It's very roughly equivalent to Algol call-by-name, or to passing a getter function and a setter function.)

In fact, I had forgotten this, but it turns out that Pascal has precisely this feature: its "var parameters" are precisely the downward-only-passable address arguments being described here, except that you can't store them in a local variable or reseat them, and (more disconcertingly to my eye) they look just like input parameters at the callsite. Oberon follows Pascal in this, despite having garbage collection and thus no need for the downward-only discipline.

Fuck C, though, seriously

I really hate C's single program-wide namespace (give me method namespaces and a module system, please), verbose type syntax, lack of multiple return values, lack of list comprehensions, lack of any kind of real error handling, clumsiness with ad-hoc polymorphism, inflexible argument passing, bug-prone coercions, bug-prone error reporting, and lack of memory-safety. (The NULLs in the above code example are perhaps avoidable in C++, but in C they'd be a consequence of the inflexible argument passing.) I like the fact that in C you can copy records by value and avoid aliasing them, just as I like the fact that you can alias fields with pointers when you want.

Types and arguments

You probably want optional named arguments for IMGUI; record literals are probably fine for that if you can invoke operations on them, and of course Tk does it by parsing command lines. Without

those, you're stuck with something like a PostScript graphics context to change things like font sizes and colors.

For pixel slinging, you probably want numeric vector types, at least fixed-size ones.

I've been wondering if you could do most of Hindley-Milner type inference, including sum types, without any parametric polymorphism, though perhaps supporting ad-hoc polymorphism. (I want sum types primarily to avoid null pointers and secondarily because of the usefulness of pattern-matching in compiler-like work.) That is, every variable and every function would have a fully concrete type, but it would be inferred from context as much as possible.

I really like the Golang interface construct. It conflicts to a minimal extent with the desire for type inference, because if method calls on concrete types and method calls indirected through interfaces are written with the same syntax, the most you could ever infer about a value from the method calls is a lower bound on its interface.

You ought to be able to infer the structure of a sum type from any wildcardless pattern match that matches on an object of that type.

I don't have a completely clear idea of how the kind of value references I'd like to be able to pass to things like `ImGui::Combo` above should interact with interfaces and heap pointers.

Some C code examples and their Platonic essences

Yesomunch

Here is some existing code written with Yeso:

```
#include "yeso.h"

int main()
{
  ywin w = yw_open("munching squares", 1024, 1024, "");

  for (int t = 0;; t++) {
    ypic fb = yw_frame(w);
    for (int y = 0; y < fb.h; y++) {
      ypix *p = yp_pix(fb, 0, y);
      for (int x = 0; x < fb.w; x++) p[x] = (t & 1023) - ((x ^ y) & 1023);
    }

    yw_flip(w);
  }
}
```

Here is what I would like it to look like:

```
#include "yeso.h"

int main()
{
  yw_open("munching squares", 1024, 1024, "") for (ywin w) {
    for (int t = 0;; t++) {
      yw_frame(w) for (ypic fb) {
```

```
      yp_scanlines(fb) for (int y, ypix *p) {
        for (int x = 0; x < fb.w; x++) p[x] = (t & 1023) - ((x ^ y) & 1023);
      }
    }
  }
 }
}
```

or, with lifetimes or scopes extending from each declaration to an implicit end at the enclosing }, as in C++, but also including loops, something like Notes on Raph Levien's "Io" Programming Language (p. 1740):

```
use yeso

{
  yw_open("munching squares", 1024, 1024, "") -> ywin w;
  for (int t = 0;; t++);
  yw_frame(w) -> ypic fb;
  yp_scanlines(fb) -> int y, ypix *p;
  for (int x = 0; x < fb.w; x++);
  p[x] = (t & 1023) - ((x ^ y) & 1023);
}
```

or:

```
use yeso

yeso.window("munching squares", 1024, 1024) w:
    (0..) t:
        w.events:
            be it Die: return 0
            be _: None
        w.frame:
            it.each_line y, p:
                (..#p):
                    p[it] <- bitand(t, 1023) - bitand(xor(it, y), 1023)
```

This looks very much like Ruby, actually, though with Lobster syntax. Some improvements over the C version:

- It's less than half as much code.
- Opening and closing the window is taken care of by the yeso.open function, which passes the window object to the argument block, and implicitly closes the window upon exit from that block. If you call it without a block, it returns the window object, and you have to close it explicitly. Similarly, `w.frame` flushes the framebuffer to the display when the block exits. This is much less bug-prone than the C version.
- The options argument in the C version is optional and omitted.
- No types are mentioned except `Die`.
- The event dispatch is done with pattern-matching instead of a bunch of functions.
- Numeric ranges are first-class callable objects; invoked with blocks, they iterate.

• Parentheses are not needed for function calls without non-block arguments; you use some other syntax to get a reference to a method when that is necessary.
• Bitwise operations are written with function-call syntax rather than infix operators.
• Mutation <- is distinct from declaration and initialization =.
• The length of a slice is extracted with #.

Block syntax is `arg1, arg2, arg3: INDENT body OUTDENT`, where the argument list may be empty. The special identifier `it`, if mentioned in the body, is added as an argument to the argument list.

Tetris piece rotation

Here's a function from my Tetris implementation:

```
void rotate_piece(char *piece, char piece_rotated[][4], int piece_r)
{
  int origin, xs, ys;
  switch (piece_r) {
  case 0:
    origin = 0;
    xs = 1;
    ys = 4;
    break;
  case 1:
    origin = 3;
    xs = 4;
    ys = -1;
    break;
  case 2:
    origin = 15;
    xs = -1;
    ys = -4;
    break;
  case 3:
  default:
    origin = 12;
    xs = -4;
    ys = 1;
    break;
  }

  for (int y = 0; y != 4; y++) {
    for (int x = 0; x != 4; x++) {
      piece_rotated[x][y] = (' ' != piece[origin + x*xs + y*ys]);
    }
  }
}
```

This converts a 16-byte piece into a 4×4 array of booleans with a particular rotation in a pretty crappy way. Instead:

```
to rotate_piece(piece, piece_rotated, piece_r):
    (origin, xs, ys) = ((0, 1, 4) if piece_r == 0 else
                        (3, 4, -1) if piece_r == 1 else
                        (15, -1, -4) if piece_r == 2 else
```

```
                    (12, -4, 1))
    (..4) y:
        (..4): piece_rotated[it][y] = (' ' != piece[origin + it*xs + y*ys])
```

The compiler can here infer that `origin`, `xs`, `ys`, `y`, `it`, and `piece_r` are ints, that `piece_rotated` is an array of arrays of bools, and that `piece` is an array of chars. There's also a tuple of three ints here.

We can do better, though, if we can return the array of arrays, and our loop is actually a list comprehension producing that array:

```
to rotate_piece(piece, piece_r):
    (origin, xs, ys) = ((0, 1, 4) if piece_r == 0 else
                        (3, 4, -1) if piece_r == 1 else
                        (15, -1, -4) if piece_r == 2 else
                        (12, -4, 1))

    (..4) x: (..4) y: ' ' != piece[origin + x*xs + y*ys]
```

Can the compiler infer that this returns specifically a 4×4 array rather than just an array of arrays (of bools)? If it could infer that, then the caller could allocate space for the return value on the stack. An alternative that doesn't involve dynamic allocation for such sequences is to return a generator object, which the caller can then unpack as they see fit, but that isn't a good default semantics for a for loop.

Or we could take a block argument to store the result:

```
to rotate_piece(piece, piece_r):
    (origin, xs, ys) = ((0, 1, 4) if piece_r == 0 else
                        (3, 4, -1) if piece_r == 1 else
                        (15, -1, -4) if piece_r == 2 else
                        (12, -4, 1))

    (..4) x: (..4) y: yield x, y, ' ' != piece[origin + x*xs + y*ys]
```

(Arguably the procedure header should have some indication that it expects a block.) Then you could invoke it like this:

```
rotate_piece(pieces[piece], piece_r) x, y:
    piece_rotated[x][y] = it
```

I really miss list comprehensions a lot in C and Golang.

Makefont event dispatch

Here's some annoying code from another yeso program, makefont.c:

```
for (yw_event *ev; (ev = yw_get_event(w));) {
  if (yw_as_die_event(ev)) {
    yw_close(w);
    free(img.p);
    img.p = 0;
    return 0;
  }
```

```
  yw_key_event *kev = yw_as_key_event(ev);
  if (kev && kev->down) {
    if (kev->keysym <= '~') {
      current_char = kev->keysym;
      yp_p2 newsize = font.g[current_char].size;
      if (newsize.x || newsize.y) defsize = newsize;
    } else switch(kev->keysym) {
        #define IF break; case
        IF yk_shift_l: display_text = 0;
        IF yk_left: if (off.x) off.x -= 128;
        IF yk_right: off.x += 128;
        IF yk_up: if (off.y) off.y -= 128;
        IF yk_down: off.y += 128;
        IF yk_pgdn: write_out_the_font(&font, argv[1]);
      }
  }

  if (kev && !kev->down && kev->keysym == yk_shift_l) display_text = 1;

  yw_mouse_event *mev = yw_as_mouse_event(ev);
  if (mev) {
    yp_p2 xy = yp_p_add(mev->p, off);
    if ((mev->buttons & 1) && !(buttons & 1)) {
      font.g[current_char].start = xy;
      font.g[current_char].size.x = defsize.x;
      update_height(&font, current_char, defsize.y);

...
```

This becomes something like:

```
w.events ev:
    be ev Die ->
        return 0
    be Key(_, keysym, Down) -> (
        be keysym Ascii(ch) -> current_char <- ch
        be Shift_L -> display_text <- False
        be Left -> if (xoff): xoff -= 128
        be Right -> xoff += 128
        be Up -> if (yoff): yoff -= 128
        be Down -> yoff += 128
        be _ -> None
    )
    be Key(_, Shift_l, Up) -> display_text <- True
    be Mouse((x, y), new_buttons) ->
        x += xoff
        y += yoff
        if bit(new_buttons, 1) && !bit(old_buttons, 1):
            update_height(&font, current_char, 0)
            font.g[current_char] = ((x, y), (0, 0))
```

The cleanup code is taken care of elsewhere by wrappers and deferred cleanup functions. I'm not sure how the keysym names and other constructor names are in scope so we don't have to say `yeso.Up` or indeed `yeso.Key`; maybe a declaration like `use yeso.keys as yk` would

allow us to use `yk.Up` and be adequate. The name collision between `Down` for a keydown event and `Down` for the down-arrow key is unfortunate.

Some amount of type declaration might simplify that — if we know that keysym is a keysym, we could bring the keysym constructors into scope for the pattern-matches.

An interesting possibility for dynamic allocation is to use explicit region-based allocation, in which each function has the possibility of constructing an arena that is freed when it returns, and can pass that region to other functions so that they can dynamically allocate things in it that will survive their return.

Uncorp

I feel like the above is a very ambitious language design for someone who's only done much simpler languages before, and perhaps it isn't ideally suited for a language that will need to have two implementations kept in sync for bootstrapping purposes.

As a first step, I think I should implement a reverse-polish-notation portable assembler, "uncorp", which supports most of the semantics of my desired language but without types or syntax. You'd have the usual unsigned integer ALU operations (`+ - * / & | ~ ^ << >> <` and maybe `sex8 sex16 sex32`), the usual memory-access operations (`! @ C! C@`), a stack operation or two (`DROP`, say), and some basic control structures — at least `IF-ELSE-THEN` and `LOOP-WHILE-REPEAT`. For function call, yield, and return, I propose:

- `: foo` to define the label `foo` at the current position, potentially backpatching earlier references;
- `foo call` to call `foo` with access to the current operand stack, which is far less convenient than the usual RPN syntax, but avoids the need to predeclare labels before calling them;
- `28 enter` to create a 28-byte stack frame containing a link to the old stack frame and install it on the frame pointer;
- `28 ret` to restore the frame pointer, pop the 28 bytes, and return, preserving the operand stack;
- `fp` to get a pointer to the current stack frame;
- `n fp+` `n fp+@`, `n fp+!`, `n fp+c@`, and `n fp+c!` to get addresses or load and store into the current stack frame (the last four are strictly speaking superfluous, but will make it much easier to generate reasonable code on i386 and amd64);
- `foo bar yield` to set the frame pointer to `foo` and jump to `bar` with the previous frame pointer and the program counter after `yield` on the operand stack, ready for another `yield` to return to where you were.

This is not quite powerful enough for cooperative threading or exception handling, for both of which `yield` would also need to set the stack pointer, but it's adequate for the downward-funargs case, whether you're passing the frame pointer implicitly (as for the single-level downward-funargs cases I contemplated above) or explicitly. Given that, it's probably better to have separate operations for setting the stack pointer, if desired.

This version of Uncorp has some desirable properties. A completely naïve compiler should be very easy to write and usably efficient, if several times slower than decent code. A slightly less naïve compiler that just does register allocation for the operand stack

should come within a factor of two or so of optimized C for a reasonably wide range of code. It can be extended with vector operations easily. And it can support the features I was describing above.

One tricky bit is how big the frame pointer is supposed to be. Remember this is supposed to be a *portable* assembler, targeting at least AVR, armel, armeb, aarch64, i386, and amd64, and I'd like it to also be *deterministic*. I think this design is too untyped to be efficiently deterministic for several reasons:

• because of the endianness issues across these architectures;
• because address arithmetic is going to have word-size-dependent overflow;
• and because it's impossible to bounds-check address arithmetic, so, say, subtracting a stack pointer from a function pointer is going to produce platform-dependent results that moreover may be too large for some platforms to even represent.

For now I'm going to not worry about that, but it's a thing to keep in mind as a reason to replace Uncorp. A little bit of compile-time computation, like `[ 24 ptrsize + ] enter`, might be enough to take care of this.

So the bootstrapping plan is to write an Uncorp interpreter in something easy and widespread, like JS or Python, and then an Uncorp compiler for amd64-ELF in Uncorp. Then I can write a compiler for the more usable imperative language above in Uncorp, and then compilers and interpreters for higher-level languages in that language.

StoneKnifeForth is 204 lines of code or 132 lines without comments, and compiles a language with 20 primitives. The 32 or so primitives of Uncorp should therefore require something like 300 lines of code, per platform. This is a lot worse than the 104 lines of my implementation of Chifir but barely more than my implementation of Tetris.

But wait! Uncorp also needs to be able to initialize variables by putting labeled binary numbers into the executable. It probably doesn't need explicit .data and .text directives (unless we want to support .bss or generate machine code from binary numbers) but it does need `d8`, `d16`, `d32`, and `d64` compile-time operations. And, as mentioned above, it needs support for compile-time computation — maybe not defining functions at compile-time, but at least things like `[ 23 ptrsize * 11 + ]`, and probably alignment too, which means you need access to the current compilation address in both .data and .text. So maybe it will be twice that, like 600 lines, though maybe more of that can be shared between platforms.

I don't think it should be the job of the source program to generate ELF headers or symbol tables, because the objective of Uncorp is to be a *portable* assembly: it should at least be possible and ideally very easy to write reasonably efficient Uncorp programs that can run without change on several different platforms.

What would list comprehensions look like in a C-level language? We don't want to saddle the containing function with handling failures, so we would like to avoid heap allocation. Stack allocation can, of course, fail, but perhaps this is less of a problem; it has no fragmentation to deal with, so proving that it won't fail seems more

tractable — although it does require bounding the list length.

Allocation doesn't entirely go away as a concern, though. For things like any(), all(), sum(), min(), or max(), or foreach, there is no real allocation problem, because we can probably evaluate the sequence lazily. But what if we really do want to store the results in an indexable or reiterable sequence? We need an array.

The stack-allocation discipline means that a callee's arrays will not be available to the caller, but block arguments can come to the rescue here — the callee can yield control to the caller's block with the freshly-allocated array.

Conventional downward-growing stacks and addition-indexed arrays are not a very good match here. Either you must implicitly reverse the array generated from the comprehension at the end, you must grow it upwards, you must periodically relocate it during the comprehension to a larger buffer size, or you must index it by subtraction. Of these options I think the implicit reversal is the best.

Topics

High academic achievement almost certainly depends more on tutoring than group averages by race or sex

Kragen Javier Sitaker, 2016-09-08 (3 minutes)

Reading http://www.scientificamerican.com/article/how-to-raise-a-genius-lessons-from-a-45-year-study-of-supersmart-children/ and it talks about how the top 1% in intelligence are responsible for a lot of our science and culture. Some controversial previous discussion has focused on how differences in variance and means between different groups might affect questions like this; Larry Summers got fired over such discussion, for example.

The Wikipedia article https://en.wikipedia.org/wiki/Sex_differences_in_intelligence summarizes the situation.

The top 1% are 2.33 standard deviations to the right of the mean in a Gaussian distribution, which is 135 IQ in the overall population:

```
> qnorm(.99)
[1] 2.326348
> qnorm(.99, mean=100, sd=15)
[1] 134.8952
> pnorm(135, mean=100, sd=15)
[1] 0.9901847
```

What if you have a subpopulation with a 10% smaller variance? It turns out to reduce the number of people in that subpopulation in the top 1% by more than a factor of 2:

```
> pnorm(135, mean=100, sd=15*.9)
[1] 0.9952372
```

What if instead you have a subpopulation whose mean is shifted down by one standard deviation, without changing the variance — an effect size of 1? This reduces the number of people in that subpopulation in the top 1% by a factor of 20:

```
> pnorm(135, mean=85, sd=15)
[1] 0.9995709
```

This is true even though the probability is quite reasonable that an event from this group will be higher than an event drawn from the entire population.

The situation becomes more extreme as you go to more extreme quantiles. Consider the top 0.01%:

```
> qnorm(.9999)
[1] 3.719016
```

```
> qnorm(.9999, mean=100, sd=15)
[1] 155.7852
> pnorm(156, mean=100, sd=15)
[1] 0.9999055
> pnorm(156, mean=100, sd=15*.9)
[1] 0.9999832
> pnorm(156, mean=85, sd=15)
[1] 0.9999989
```

That is, at IQ 156 and above, where we find 0.01% of the population, a hypothetical population with a 10% lower standard deviation will be underrepresented by almost 6:1, and a hypothetical population with a 1-SD-lower mean will be underrepresented by more than 90:1; getting the same result by tweaking the variance requires a 21% smaller standard deviation, 11.9 IQ points.

A hypothetical group with a 1-SD-lower mean but also a 21% *larger* standard deviation (of 19 IQ points) matches and slightly exceeds the overall population at this level:

```
> pnorm(156, mean=85, sd=15/.79)
[1] 0.9999077
```

In this context it is worth pointing out that tutoring improves student performance by two standard deviations (Anania (1982, 1983) and Burke (1984)):
http://changelog.ca/quote/2012/09/23/tutoring_two_sigma.

So what if we have a hypothetical group with an unchanged variance but a mean two standard deviations higher?

```
> pnorm(156, mean=130, sd=15)
[1] 0.9584818
```

If student performance is equivalent to IQ, 4.2% of them will exceed the performance of 99.99% of the rest of the population. This is a larger difference than has ever been suggested for (mean or variance) intelligence differences due to race or sex.

(In the paper described, tutoring also substantially reduced the variance of achievement scores, but this is probably because the achievement scores had a ceiling; all three of the distribution curves from the experimental data intersect at the X-axis on the right, and the "tutorial" group's distribution is noticeably skewed to the left.)

Topics

- Politics (p. 3639) (39 notes)
- Facepalm (p. 3450) (24 notes)

Notch scorn

Kragen Javier Sitaker, 2019-04-20 (5 minutes)

I talked briefly with the pariah Notch on the hellsite the other day, thanking him for writing Minecraft. He mentioned that he enjoyed reading Knuth; one of his followers asked if Notch had left Knuth's work TAOCP unopened when Notch learned that Knuth was opposed to some of Bush's policies, including the US invading Iraq.

This seemed sad to me. What would it take for a person to not read Knuth for such a reason? Not only would they have to belong to Bush's faction, they would have to consider it more important that Knuth belonged to an opposing faction than what Knuth knew or didn't know about computer programming.

It seemed to me that this follower was arrogating to themself the right to judge Knuth's worth, on the basis that Knuth belonged to the other faction, the inferior faction. I have no reason to believe that the follower was a person of significant intellectual achievements themself, much less a scholar of Knuth's stature, but they seemed to consider this irrelevant; for them, dismissing Knuth's work as worthless, even repugnant, was their prerogative for belonging to the correct faction of the culture war.

In this sense, the elevation of factional alignment over scholarly achievement necessarily implies the elevation of ignorant and foolish thugs over scholars — not, perhaps, all scholars, but at least scholars of opposing factions. Moreover, the assignment of a scholar to a faction depends not on scholarly criteria but on thug criteria. This is the principle by which Archimedes was struck down in Syracuse by a Roman soldier, by which Sulla burned the Academy, by which Nazi ruffians burned the books of Jewish scholars who were their superiors in every way, and by which Qin Shi Huang burned the books and buried the scholars. This is the principle by which the Boxers burned the Yongle Encyclopedia, the greatest encyclopedia the world knew before Wikipedia, and by which Mossad thugs assassinated the Iranian nuclear scientists. And this is the reasoning behind the prosecution of my friend Aaron Swartz.

Such an inversion of priorities is inevitable in wartime — when Julius Caesar set fire to the Library of Alexandria, he had intended only to set fire to his ships, not to the library. A human will do nearly anything in their futile effort to secure their own survival, even if it puts at risk values much more precious.

Subjugating the wise to the foolish and ignorant leads invariably to tragedy. The ruler who listens to the counsel of fools brings waste to their land and poverty to their people. And this is what we do when we prize ideology and factional loyalty over wisdom and learning.

Yet are we to honor learning even in the service of evil? Should the Iranian nuclear scientists be permitted to put the fire of the stars in the hands of the mullahs who rule their land with such cruelty and injustice?

These problems arise when the scholars accept this perversion of harmonious order, willingly serving as mere instruments of ignorant and foolish people. By pledging their loyalty to those who follow not truth and prudence but domination and power, those whose position

owes not to wisdom and learning but to brutality and intimidation, they abdicate their responsibility to speak the truth and serve the well-being not of one faction but of the world.

True scholars serve a master higher than any government or movement, as exemplified by Socrates's suicide, by the defensive fortifications of the Mohists, and by the false legend of Galileo's defiant "Eppur si muove." This is the reason for the principles of academic freedom. Scholars who speak their minds even when it is unpopular, who act in the service of truth and benevolence as they understand it, these scholars are the noblest and best of humanity. Even their enemies should honor them. But servile scholars who allow themselves to be employed by other people, as if they were pieces of equipment; who obey rather than choosing; who lie and who remain silent in the face of injustice in order to help their own government or faction; those are no scholars at all but mere pedants.

Topics

- History (p. 3500) (71 notes)
- Politics (p. 3639) (39 notes)
- Pompous (p. 3641) (6 notes)
- Scholarship (p. 3695) (2 notes)
- Factionalism (p. 3451) (2 notes)
- Aaronsw

Pensamientos acerca de diseñar un calefón solar

Kragen Javier Sitaker, 2012-10-15 (2 minutes)

Unos pensamientos más de anoche acerca de termotanques solares:

• Hablé con mi papá acerce de lo que él había construído para su casa rodante. Él usó esa tubería negro. Dice que es de polietilena y por eso es seguro para uso para agua potable hasta la temperatura de hervir.
• Además tiene la ventaja que, si en algún momento el agua se congele, no rompe los caños. Por lo menos la primera vez.
• Él usó vidrio templado, de lo cual pidió la fabricación en el tamaño que quiso.
• Acrílico (plexiglas) es un posible alternativo al vidrio que es más duradero contra piedras de hielo. Tiene la ventaja o desventaja que es bastante transparente a infrarrojo térmico.
• Calculé que una suba de 14 grados encima de la temperatura ambiente (imponiendo un límite de 49 grados cuando la temperatura ambiente es de 35 grados) puede sostenerse en un sol de 800W/m² con una resistencia térmica de solo R 0.018 (m²K/W), lo cual es más o menos lo que obtenés con pasaje libre de viento en dos lados del panel, sin nada de aislación extra.
• No obstante, eso es una idea estúpida porque así cuando la temperatura ambiente es de solo 0 grados, el agua "caliente" será de solo 14 grados. Es necesario usar alguna temperatura de "referencia" más estable para poder limitar confiablemente la temperatura.
• Un posible temperatura de "referencia" para eso sería los 3.7K de espacio: cuando no hay nubes, podés radiar infrarrojo al espacio para mantener la temperatura a un nivel seguro.
• Un problema con esto es que, dado que la diferencia entre la temperatura de referencia y la temperatura que quiero del agua es muy grande (320 K) comparado con el rango de temperaturas aceptables (6 K), aunque la radiación Stefan-Boltzmann anda con la cuarta potencia de la temperatura, igual será enormemente ineficiente. La emisión térmica a 43 grados será unos (316^4/322^4) =

Topics

• Physics (p. 3632) (119 notes)
• Energy (p. 3438) (63 notes)
• Thermodynamics (p. 3747) (49 notes)
• Household management and home economics (p. 3504) (44 notes)
• Solar (p. 3717) (30 notes)
• Español (6 notes)

A one-motor robot

Kragen Javier Sitaker, 2015-09-03 (13 minutes)

I've been thinking for a while about how to reduce the number of motors needed in a robot, because motors are the only mechanical part you can't reasonably 3-D print (or resin-cast, or laser-cut, or made with other easily-accessible means of automated fabrication), and they're also kind of heavy, expensive, and unreliable.

Also, in some cases, it makes sense to use an internal-combustion engine rather than an electric motor to supply mechanical power to the robot, because it can use fuel, which stores energy two orders of magnitude more densely than batteries do, thus allowing two orders of magnitude more autonomy. Where a battery-powered quadcopter can fly for 15 to 25 minutes, a gasoline-powered one of similar design could reasonably expect to fly for hundreds of hours; but internal-combustion engines don't currently scale down nearly as small as electric motors without serious loss of efficiency. Boston Dynamics' spectacular designs typically use pneumatic actuators powered by what sounds like a two-stroke internal-combustion engine, but I haven't investigated in more detail.

Armatron

When I was a kid, I had a robot arm from Radio Shack called Armatron. It had six degrees of freedom: two rotations at the shoulder, rotation at the elbow, two rotations at the wrist, and opening and closing a gripper; but it had only a single motor, which ran continuously at a fixed speed as long as the robot was turned on. The two twistable joysticks mechanically engaged the various degrees of freedom with the motor or a brake, and through a set of slip-clutches, the force through each degree of freedom was limited to the small forces that the cheap nylon parts could handle easily without risk of breakage. It was a toy, but mechanically, it was a beautiful design.

It's not obvious that this is directly applicable to the problem of designing an electrically controlled robot, because you'd still need some kind of mechanical actuator for each of the degrees of freedom — but now to operate the joystick rather than to directly actuate. This is a big improvement in a sense; you can get by with solenoids or very small motors.

Below, though, I revisit this problem, with a camshaft-based solution.

Turtlebot

Consider the problem of a Roomba-like turtle robot, a horizontal disc with driven left and right wheels, plus maybe some idler wheels to keep stable. If you mount a pen in the middle of it, as with the original Logo turtle robots from the 1970s, you can draw any continuous shape on the floor just by alternating between turning in place and moving forward. You don't even need to turn both directions or move backwards.

In particular, you can draw smooth lines at any angle, without being able to independently control X and Y actuators.

You can straightforwardly achieve this with two motors, one on each wheel. But it occurred to me, thinking about this, that a single reversible motor would also work. It would drive, say, the left wheel directly, and the other wheel through a sort of mechanical full-wave rectifier, such that the other wheel always turns the same direction regardless of which direction the motor is turning. You could achieve this by driving the shaft of the wheel through two sprag clutches that freewheel in the same direction, one driven in the same direction as the motor (perhaps directly by the motor's shaft), and the other driven in the opposite direction, perhaps through a bevel gear between it and the first ring.

(You can 3-D print or, I think, even laser-cut a reasonable approximation to a sprag clutch in two or three relatively simple flexible parts, if you're not too worried about maximizing torque per volume.)

With this simple mechanical arrangement, the right wheel always drives forward, while the left wheel can go either forward, to move the robot, or backward, to turn it. So a single reversible motor should enable you to draw any figure, limited by your control system (I've argued elsewhere that closed-loop feedback is now cheap enough that we should apply it in almost every case now) and the very small backlash of your rectifier mechanism.

RepRap and 3-D printers

A different problem from the turtlebot is the 3-D printer, which typically needs four degrees of movement freedom — in a case like the RepRap Prusa Mendel, you have two parallel Z motors suspending the X motor via triangular-thread allthread bars as linear actuators, the X motor moving the print head via a belt drive, a reversible extruder motor on the print head itself pushing the filament into the hotend, and a Y motor moving the heated bed via the same kind of belt drive as the X motor; a total of five motors with four degrees of freedom. The motors used are heavy, expensive NEMA 23 stepper motors, which are uncommon enough that it's hard to salvage them from e-waste, even though e-waste is full of motors. And, of course, printing such a motor on the RepRap is far beyond its current capabilities.

(Ganging together multiple smaller motors to drive a single shaft is probably the right response to the problem of having only smaller motors easily available.)

I've thought (and I wasn't the only one) that you could dispense with the Z stage if, say, you replace the Y actuator with a circular θ actuator, and goes up a Z step every time the θ actuator makes a full revolution. The biggest problem with this is that, with the straightforward approach where the Z movement is directly controlled by one or more screw threads that revolve 1:1 with the θ actuator, you go up by an entire thread pitch. 1mm is the finest thread pitch commonly available in Allthread rods. But a 1mm Z-step is a very large step. 0.35mm and 0.25mm are more common layer thicknesses, providing more acceptable surface finishes, avoiding the need for extreme plastic feedrates, which would require a much larger hotend and extruder motor, as well as suffering more from heat-induced damage to the plastic's chemical structure and slumping before cooling. Gearing down the θ rotation by 10× would be

sufficient to solve this problem.

That reduces the machine from five motors to, say, three. If we are willing to limit the horizontal planar scan to a single fixed pattern, where we don't get to choose the direction of movement at each "pixel" but only how much material to deposit there, then we can gear the θ movement to the X movement, which probably should also become circular. This dramatically reduces the efficiency of the device and worsens surface finish, but with a sufficiently small build volume, perhaps it could be acceptable anyway. Essentially we have gone from a 3-D plotter to a raster 3-D printer, although with a funny circular scan pattern.

And now we have only two motors to control independently.

If we are to recapture the horizontal surface-finish and a shadow of the efficiency we got with our vector plotter, we could perhaps use the turtlebot approach: have a single motor alternate between determining the direction of motion, while stationary, and determining the speed of motion. In the RepRap context, this will cause some ooze problems, as the hotend's plastic extrusion rate is more or less a low-pass-filtered version of the extrusion motor's movement. (Thus oozebane and similar techniques in current FDM slicers depend on briefly reversing the extruder motor.)

How would this work? So far all the things I've thought of are ridiculously complicated and depend on analog mechanical computation devices called "integrators" "integration", which turn the position of their input into the rate of motion of their output, thus numerically (or analogically) approximating the computation of a definite integral. There are several different types that work by different mechanisms. In the mechanical computation context, integrators are notoriously sensitive devices, but I suspect that closed-loop control could fix that problem in this context. The famous Navy film about fire-control computers goes into some detail about the relevant mechanisms.

However, mechanisms that are ridiculously complicated when you have to cut all of their parts out of metal, with dimensions accurate to five significant figures, might no longer be ridiculously complicated when you can 3-D print them and compensate for their dimensional errors using closed-loop digital control.

That leaves us with two motors: one to control the X-Y (or rather θ-φ) scanning, bringing along Z for the ride, and another on the extruder. Can we get rid of the extruder motor?

Well, if we always extrude at a constant speed, perhaps through some kind of a gravity feed, we could control the amount deposited in each location by varying how rapidly we pass over it. But this seems relatively impractical, especially with materials like plastic, which is light and therefore hard to gravity-feed and can be usefully deposited through a very small orifice. (The orifice is almost inevitably smaller than the filament extruded, because the plastic swells upon being released from the orifice.)

For a machine to produce sand paintings, on the other hand — perhaps even very vertical sand paintings with "magic sand" — it might work perfectly.

Camshafts and Armatron revisited

Another thing we often want is a large number of very simple

actuators, such as valves which can be either open or closed, or pens that can be either up or down. If we can accept intermediate glitches, we can achieve this by driving all of the actuators on a camshaft where the cams are the different bits of a binary code, and the rotational positions of the camshaft are the different codes; the followers read off the various bits of the code for the current position. Many traffic lights in Buenos Aires, for example, work this way, with the cam followers actuating electric switches.

In the general case of N actuators, your camshaft needs to be able to distinguish among 2^N rotational positions, although by using Gray codes, the minimum width of a cam lobe can be twice that, and the overall number of bit changes can be minimized to the 2^N minimum. In some cases, not all possible codewords are necessary.

This, again, can be driven by a single motor that turns in one direction to select the set of actuators to be activated, and then in the opposite direction to provide mechanical power that is routed through them. (For example, pumping hydraulic fluid through a set of open valves, or driving a set of engaged gears.)

In particular, this is a possible solution to the Armatron problem: you have six pairs of mutually exclusive actuators, or equivalently six three-position actuators, so we need $3^6 = 729$ distinct positions of the camshaft, which is a bit much; but it's probably adequate to be able to engage two or three of the joints at a time, rather than all six, which reduces the number of combinations dramatically. (I'm too lazy to calculate the case for up to three joints right now, but you can see that $2C6 = 6 \times 5/2 = 15$ pairs of joints, each of which can be engaged in 4 ways for a total of 60, plus six individual joints engaged in two directions for 12, plus all idle for 1, brings us down from 729 to 73 camshaft orientations, which is eminently feasible.)

Topics

Multitouch livecoding

Kragen Javier Sitaker, 2018-06-17 (1 minute)

I thought it would be interesting to explore multitouch interfaces for livecoding music. Here are some ideas:

• On a cellphone, you can expand your working area by using the phone accelerometers (and gyros, if available) to rotate the phone to move around a larger virtual space.
• Quasimodal interface elements pop up during a touch and then permit continuous adjustment in one or more dimensions with one or more touches — the same or different touches.
• Waveform and spectrum displays at different scales can elucidate what is happening in a single node.
• Individually reified signal inputs might offer a quasimodal action to overwrite them with some existing signal.

Topics

Dyneema

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

I think I found some Dyneema fishing line for AR$100 a roll at El Brujo, Sarmiento 2443. Green, braided. This scrap of 50-pound (22.7 kg) line he gave me is about 0.35 mm diameter, which works out to 2.3 GPa strength, and it floats, which suggests it is sure enough Dyneema. It's a Chinese brand and doesn't say "Dyneema", "Spectra", or "UHMWPE". He has rolls of several different diameters for $100, but I don't know if they're different lengths or what.

http://articulo.mercadolibre.com.ar/MLA-615447478-multifilamento-dyneema-triple-fish-014-y-018-mm-x-100-m-_JM Triple Fish Bully Braid Dyneema fishing line: 100 meters for AR$305 ≈ US$22. They claim the 0.14 mm diameter line is good to 10.2 kg, which would be 6.5 GPa, almost three times the strength.

https://en.wikipedia.org/wiki/Ultra-high-molecular-weight_polyethylene gives 2.4 GPa as its strength.

http://www.dtic.mil/get-tr-doc/pdf?AD=ADA606636 gives a variety of higher strengths in their tests, ranging from 3.63 ± 0.19 GPa up to 4.25 ± 0.21 GPa, with the higher strengths being achieved at higher strain rates up to a kilostrain per second (i.e. loadings more like bullet impacts). For Dyneema SK76, it also shows nice linear Hookean elastic stress-strain curves with a Young's modulus of about 100 GPa. This is about twice as compliant as a steel, and stronger than most steels.

(Dividing out, 2.4 GPa / 100 GPa is 2.4% elongation at break, which is almost three times what steel gives you. If we suppose the 100 GPa number is accurate for the Bully Braid, that's 6.5% elongation at break, 65 mm/m, which works out to a spring energy capacity of (65 mm 10.2 kg gravity / 2) = 3.25 J/m; the whole roll holds 325 J for 14.8 J/US$.)

http://www.matbase.com/material-categories/natural-and-synthetic-polymers/polymer-fibers/synthetic-fibers/material-properties-of-dyneema.html#properties suggests that different varieties of UHMWPE have substantially different strengths and very substantially different stiffnesses.

It's kind of amazing to think that five strands of this fishing line could support me.

https://www.redwoodplastics.com/brochures/uhmw-engineering-data.pdf gives a coefficient of static friction for UHMWPE on steel of .15–.20 and of dynamic friction .12–.20, or .05–.08 if oiled. This suggests that for many purposes, you can run oiled UHMWPE cord around round steel bars without pulleys, avoiding the

Topics

Can you read the lunar lander's plaque from Earth? Or write a new one?

Kragen Javier Sitaker, 2015-09-03 (9 minutes)

The US lunar lander has a plaque that says "WE CAME IN PEACE FOR ALL MANKIND", in a successful effort to avoid sparking a global thermonuclear war. It's 384 megameters away on average, although sometimes it gets as close as 356.4 megameters. Can you read it from Earth?

An interferometer, which is kind of a generalization of a telescope, distinguishes things by the phase of light reflected from them. A lens or telescope mirror is a device to adjust the phase such that light waves emanating from the same point will be focused onto the same point, and more importantly for the question here, light waves emanating from different points will be focused onto different points. It uses the difference in phase across the width ("baseline") of the interferometer to identify each point.

To be clearly distinguishable by interferometer, the phase difference of two points ought to differ by a significant fraction of a wavelength. We can take half a wavelength as a convenient number. And let's figure we can't conveniently use radiation any harder than, say, 300 nm wavelength, for example because it's hard to focus or because the Sun doesn't emit it or it doesn't penetrate our atmosphere.

I don't know how big the letters on the plaque are, but let's suppose that centimeter resolution is good enough.

So how big of an interferometer do you need? Does it fit on Earth?

The Euclidean sum of 384 megameters and 1 centimeter is a bit tricky to calculate. But you can easily calculate that 1 centimeter over 384 megameters is about 30 picoradians, and 150 nm is 30 picoradians at a radius (or baseline!) of 5 km. So even if the letters are a bit smaller than I imagine, you can read them from Earth with optical interferometry.

Laser-printing on the moon

You can do better, though! You can laser-print a new plaque by selectively melting the lunar soil from Earth with phased-array lasers, thus changing its color! If you require a 50km baseline, you can get down to one-millimeter resolution. You need high enough power to melt it, not just slightly warm it; but exactly how much power you need goes down as the resolution improves, but it depends on the thermal conductivity of the soil, its density, its heat of fusion, its melting point, the depth of penetration of light into it, and its specific heat. Also, to estimate how fast this happens, you need to know its albedo and the spectral selectivity of its emission spectrum.

You need a few hundred milliwatts to laser-print on the moon

I found some information about lunar soil physical properties online. Lunar soil particles are very fluffy, with half a square meter of surface area per gram. Specific gravity of the solid mass of regolith particles is about 3.1 g/cc; porosity is typically 50%; consequently lunar soil is about 1.6 g/cc, although varying from 0.8 to 2.3, and perhaps even lower in the top surface layer. Thermal conductivity in the top centimeters of lunar soil was measured at 1.5×10^{-5} W/cm/° in the Apollo 15 and 17 missions. Its albedo is about 0.07.

I don't know the spectral selectivity, heat capacity, heat of fusion, or light penetration depth of the lunar soil, but let's estimate. The soil includes some components that would melt at lower temperatures, including many glasses that won't have a heat of fusion at all, and it might be adequate to melt just those. But I'm going to use a somewhat more pessimistic analysis to make sure my conclusions are ironclad. Let's suppose it has no particular spectral selectivity, melts at about 1000°; has a heat capacity similar to quartz's — 65 J/mol/° / (28 + 2·16 = 60) g/mol ≈ 1 J/g/°; and has a heat of fusion similar to quartz's — 9 kJ/mol ≈ 150 J/g.

The lunar surface can get as hot as 220 K during the day even before we start shooting lasers at it.

So we need, hypothetically, to heat the surface of the regolith to 1000°, sucking up about 780 J/g, and then melt it with 150 more J per gram. At 1000° a black body soil would be emitting 57kW/m², so that's probably about what we need to illuminate it with in order to keep heating it up once it's already at that temperature. But how much actual power is that?

It depends on the area you can illuminate, of course — the tighter you can focus the beam, the easier it gets. Above we posited a square-millimeter focus, which is 10^{-6}m², at which point you'd need 57mW — an eminently feasible number! As long as you can focus 57mW onto a one-millimeter square in a vacuum, it will eventually heat up to 1000°. How fast?

1.5×10^{-5} W/cm/° at this temperature is about 1.5×10^{-2} W/cm, which is to say, 1.5 mW mm/mm² — we can expect on the order of a milliwatt to escape to a depth of on the order of a millimeter through an area on the order of a square millimeter. None of this is very precise because of course the planar focus spot gradually morphs into hemispherical shells of heat spreading into the regolith, so really at the depth of a millimeter you need to be considering about two square millimeters of almost-hemispherical shell, but it's close enough to show that the vast majority of heat will be lost radiatively rather than conductively. You might need 58mW or 59mW or 60mW to reach equilibrium, but really you want to be far from equilibrium, like at least 2× and ideally over 10×.

So supposing we're focusing 600mW and up on this square millimeter on the lunar surface, losing 7% of it to reflection, two or three milliwatts to conduction, and 60mW to thermal reradiation, leaving 500mW, focused on a millimeter-sized spot that's about a millimeter deep, with a density of about 1.6 g/cc (thus about 1.6 milligrams) and a heat capacity of 1 J/g/°. This gives us a temperature rise of about 300°/s, so it takes three or four seconds to melt the spot, so we can write on the moon at about 0.3 mm²/s with 600 milliwatts.

This should scale linearly with applied power. We should be able

to do 3 mm²/s at 6 watts, or 30 mm²/s at 60 watts. Even 6 kW should be doable without any extra difficulty (remember, we're talking about a phase-locked phased-array ultraviolet laser system spread over a thousand square kilometers or so) and should allow you to print on the moon at 3000 mm²/s.

Lunar soil is accumulating at about 1½mm per million years, so whatever you write there might be readable for tens of millennia if daily dust redistribution doesn't cover it up.

The total information capacity of the moon's surface would be close to 1 bit per mm² if encoded in this way. The moon's radius is 1738 km, so the part facing us has space for on the order of 5×10^{18} bits encoded in this way, which is several hundred petabytes.

Calculating the bandwidth of this data storage channel is trickier.

Related proposals

Relevant to this is NASA N78-13420, "Analysis and Design of a High Power Laser Adaptive Phased Array Transmitter", from 1978. They propose to use photovoltaic-powered terrestrial lasers, phase-locked to produce a phased array, to deliver 5 megawatts of infrared laser power to satellites, from a 6-meter transmitter to a 2-meter collecting aperture, for example to power orbital transfer maneuvers. They predicted overall power transfer efficiency of 53%, which seems inconceivably large to me given the generally low efficiencies of lasers, especially in the 1970s — and indeed, it turns out they're talking about 53% of the light coming out of the laser making it to the satellite, and they suggest using an isotopically pure laser that wasn't yet developed (and maybe still hasn't been), plus six other areas of "advanced technology development required", which I think all do exist now.

In the high-spatial-resolution phased-array field, there's a 1993 proposal for an OVLA (NASA N93-13583), or "optical very large array", like the radio-telescopy Very Large Array, using optical heterodyning to get the equivalent of a very large aperture.

The recent DE-STAR proposal from Cal-Poly is the closest in spirit: it proposes a high-power phased-array laser to divert or evaporate asteroids and comets in danger of hitting Earth.

Topics

Byte stream gui applications

Kragen Javier Sitaker, 2019-11-29 (updated 2019-11-30) (17 minutes)

How should we run graphical programs on machines remote from the GUI, that is, remote from the machine the user is using? This includes scenarios such as the following:

• DisplayLink-like USB-connected or Ethernet-connected monitors (whether for multiple monitors per user or multiple users of a larger machine, like X terminals);
• Remote server administration;
• Screen sharing for, for example, remote pair programming, remote demos, or visual aids for teleconference presentations;
• Remote access to expensive shared computational resources, such as supercomputers (the original rationale for the ARPANet project);
• Including GUIs in tiny embedded computers that don't have monitors of their own, but do have ports where you could connect monitors, touchscreens, keyboards, and mice;
• Access to remote datasets, such as your email, although this very-thin-client approach is more demanding of the server.

Byte streams and pipes

In some sense the lowest common denominator is bidirectional byte streams with maybe some kind of escaping; this works over sockets, RS-232 serial links (which can run at megabits per second nowadays --- RS-422 and RS-485 can reach tens of megabits), and over `ssh` with proper authentication. So, on Unix, a very reasonable way to spawn a graphical interactive app on a remote machine is to spawn an `ssh` process connected to input, output, and error pipes, and then select(2) or similar on those pipes to send events to the app and receive commands from it. Thus ssh can take care of authentication, spawning processes on the remote host, checking them for errors (although if there's an error you'll probably only get a textual message back on stderr), and detecting when they die.

This also lets you run graphical apps on USB serial devices or other embedded devices that just have a serial port. It doesn't inherently give you a way to run multiple graphical apps over a single serial connection, so one serial device would be one app, and if that device contains the display and keyboard and whatnot, it can only *display* one app. But that one app could be a full-fledged multiplexed display server in its own right, spawning off ssh children and whatnot.

Even Docker containers, for all their headaches, support an 8-bit-clean bytestream interface on stdin and stdout, as long as you don't use `docker run -t`; this shows that all 256 bytes make it through safely:

```
docker run --rm python:2 python -c \
    '__import__("sys").stdout.write("".join(map(chr, range(256))))' |
    tee >(docker run --rm -i alpine od -t x1 >/dev/tty) | xxd
```

(The `-i` keeps it from closing stdin.)

This all assumes the usual app interaction model, as I called it in

Dehydrating processes and other interaction models (p. 3208), which suggests that for more flexible kinds of interaction that aren't tied to particular hosts, a different interaction model would probably work better.

ssh

I just did a quick experiment with OpenSSH:

```
$ time ssh -vC user@server dd if=/dev/zero bs=100M count=1 | dd bs=1k | wc -c
...
debug1: Sending command: dd if=/dev/zero bs=100M count=1
1+0 records in
1+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 11.3868 s, 9.2 MB/s
debug1: client_input_channel_req: channel 0 rtype exit-status reply 0
debug1: client_input_channel_req: channel 0 rtype eow@openssh.com reply 0
debug1: channel 0: free: client-session, nchannels 1
debug1: fd 1 clearing O_NONBLOCK
Transferred: sent 65224, received 341432 bytes, in 12.5 seconds
Bytes per second: sent 5205.9, received 27251.5
debug1: Exit status 0
debug1: compress outgoing: raw data 28009, compressed 14245, factor 0.51
debug1: compress incoming: raw data 104917069, compressed 229911, factor 0.00
104857600
102400+0 records in
102400+0 records out
104857600 bytes (105 MB) copied, 14.4017 s, 7.3 MB/s

real    0m14.411s
user    0m1.964s
sys     0m4.108s
```

That is, ssh was happy enough to transmit me 100 megabytes of zero bytes from the server in 14 seconds.

By comparison, running the command `true` produced this result:

```
Transferred: sent 3352, received 2400 bytes, in 1.2 seconds
Bytes per second: sent 2701.5, received 1934.3
debug1: Exit status 0
debug1: compress outgoing: raw data 154, compressed 135, factor 0.88
debug1: compress incoming: raw data 566, compressed 538, factor 0.95
0+0 records in
0+0 records out
0 bytes (0 B) copied, 3.09319 s, 0.0 kB/s
0

real    0m3.107s
user    0m0.148s
sys     0m0.004s
```

So the 100 megabytes only cost 11.3 seconds. And, of that, 4 seconds were spent in the kernel on my end, shuffling the bytes between the processes, and 2 seconds were spent decompressing them.

Indeed, repeating the experiment using an ssh master connection

configured with `-o 'ControlMaster yes' -o 'ControlPath somepath'` and an ssh slave connection configured with just the ControlPath, I got 11.2 seconds. `true` took 400 ms or so.

This is not super great for video; this netbook's display is 1024x600 24bpp 60fps, which is 110.592 megabytes per second uncompressed, about four times what the kernel is managing to copy through a pipe. This is why the draft Wercam protocol design in BubbleOS uses shared memory (on Unix, with `mmap`, passed over sockets as open file descriptors). But it's probably adequate, and if the data is encoded with some kind of video codec, it might be totally fine.

Resynchronization and topology

X-Windows apps die if the display server dies or they lose their connection to it. This is very limiting. Terminal apps connected over a serial port, or GUI apps connected to a monitor or KVM switch, don't have this problem; you can reconnect to them later, even plugging in a different monitor, and keep using them. GNU Screen and tmux offer the ability to do this with terminal apps running on a virtual terminal connected to via ssh, as well.

Making that kind of thing work well imposes some extra requirements on the protocol design. There are a few different cases where we might need to resynchronize after some kind of protocol desynchronization.

First, you might want to recover from bugs in the application, the display server, or the connection between them that caused communication to break down --- an unescaped framing byte, say, or noise on a serial line.

Second, in an embedded context, the application being displayed might have crashed and restarted.

Third, the display and input devices might have been disconnected for a while, and then reconnected, without having been able to see anything in between. Maybe it's a different display, or maybe it's crashed and restarted or lost power in the interim.

Fourth, you might want to have multiple displays and input devices connected to the same running application at the same time. If all, or all but one, but one of the input devices are disabled, this is straightforward ("live streaming"); otherwise you need some way to keep the input devices from screwing up the framing when they try to talk at the same time, and somehow negotiate the window size.

Fifth, you might want to record and replay a video stream ("screencasting").

These can all be handled to one extent or another by adaptors of some kind spliced into the protocol, rather than by the protocol design. That may or may not be the best way to solve the problem. For example, live streaming benefits from adaptors to adapt to the available bit rate.

Codecs

The standard VNC RFB protocol uses only lossless "video codecs", and they are not very efficient. XPra uses modern lossy video codecs in order to get dramatically better efficiency at the cost of a little latency. Specifically, its non-deprecated codecs are rgb (compressed with zlib, lzo, or lz4), png, VP8/VP9 ("vpx"), and H.264.

Video codecs have an interesting feature: they are often also video

container formats that are designed for broadcasting over the airwaves. Broadcast formats have to be unidirectional and permit synchronization in the middle of the stream, so that turn your digital TV on or change channels, you start seeing the video on the channel you're receiving. You can start reading the stream at any point, and pretty soon you'll manage to synchronize with the framing, and then an I frame (internally coded, a lossily compressed image without any reference to other frames), and then you're displaying video.

Any video stream format that allows this pretty much automatically gives you items #1 through #5 above, except when it comes to handling of input events, including things like window size changes.

Traditional "video stream formats" filling this role include NTSC, PAL, SECAM, VGA signals, and ASCII text for teletypes and video terminals. These are optimized for displays with very little memory. NTSC needs to remember where you are relative to the the horizontal and vertical sync and the colorburst, and PAL has some additional slight twist. VGA is the same but without the colorburst. Teletypes only need to remember where they are on the line and the currently printing byte, if any; video terminals have 2K or 4K of RAM for the screen contents, or maybe a bit more.

I think it's okay for the protocol to require displays made of modern electronics to have more memory than that, 8 to 32 bytes per pixel, say.

Bidirectionally predicted frames (B frames) are potentially more of a problem. Their mere existence imposes potentially unacceptable codec latency, but if your remote app is a video player, you're going to have substantial bandwidth inflation if the video streaming protocol doesn't support B frames.

Supporting multiple codecs and demanding that the stream be readable without any kind of codec negotiation allows applications to be very simple but potentially requires a lot of complexity on the display side. As a very crude estimate, a single modern codec is on the order of a meg of code:

```
$ ls -lL /usr/lib/i386-linux-gnu/libx264.so.142
-rw-r--r-- 1 root root 976296 Mar 23  2014 /usr/lib/i386-linux-gnu/libx264.so.142
```

But maybe it's possible to get the requisite compression of 4 to 8 with a much simpler codec, something like MPEG-1, but maybe more modern.

Input events

The main requirement for input event handling is that they need to get delivered even if one or both of the app and the display crash and restart, or if the user reconnects using a different terminal. There are a lot of ways to do this; the simplest one is to send all possible input events all the time rather than attempting to economize in any way. At least without cameras, the total input event data can't be more than a tiny fraction of the video data torrent rushing the opposite way; this probably makes it insignificant, although there do exist unusual scenarios with very asymmetric bandwidths, such as some satellite communications. Other possibilities include resynchronizing the input state (shift keys, etc.) whenever a reconnection is detected or

suspected.

You also need to send the window size if the app is sending video of the wrong window size, and maybe when it's sending no video, too.

GUI apps on embedded microcontrollers

You probably don't want to have to include an H.264 encoder in your Arduino; you probably want to support some kind of simpler protocol than H.264, maybe something uncompressed; for example, a sequence of nothing but "P frames" that consist of local area updates, some of which are actual updates and others of which are redundant retransmissions of unchanged scan lines. At some point you might want to add reduced-color-depth pixels to the protocol, but even Arduino serial ports can run at 2 Mbps (83 kilopixels per second at 24 bits per pixel) and if you're transmitting via `ssh -C` you'll get paletting implicitly from Lempel-Ziv compression, so lower color depths are never going to be a big win.

And you don't need text in the protocol. In dofonts-1k I included a full printable-ASCII 6x8 font in 64x36 bits, 288 bytes uncompressed, or 482 bytes PNG-compressed and base64ed. It's reproduced below. The file, 1KiB in all, also includes a sort of terminal emulator that uses the font to render ASCII text on a `<canvas>` which is 20 lines of JS code, including lines of code that just say `}`. You have room for that in your Arduino's Flash.

83 kilopixels per second means that a full-screen redraw at 1024x600 would take 7.4 seconds, and if you're using the Arduino Serial library, more like 30 seconds. Even if you had only one bit per pixel, it would still take over a second. There are applications that update the screen regularly for which this kind of latency is acceptable, but having the screen partially redrawn all the time is not. Double-buffering is the usual solution, and it's within the 8-32-byte-per-pixel budget described above. But how should we do it?

The basic atom of the Arduino protocol described above is something like draw(x, y, w, h, pixeldata), where typically wxh is on the order of 64 to 256 (about a millisecond at the data rates discussed above). The simplest approach to double-buffering is to add a flip() operation that makes all the previous changes visible. An alternative would be to include some kind of timestamp in the draw() operation that specifies when to make it visible, perhaps a one-byte number of intervening draw() messages to delay the draw operation.

The (x, y, w, h, delay) header might be 7 or 8 bytes, so prepended to a 192-byte 64-pixel data block, it amounts to about 4% overhead, which seems acceptable.

A more stateful protocol might handle double-buffering by including offscreen pixmaps and commands to copy regions between them, but that poses risks to resynchronization after disconnects.

BubbleOS thoughts

This also suggests a different way to write and run BubbleOS Yeso programs, particularly on Linux: put the code for interfacing with, for example, X-Windows or the framebuffer, into a process which runs the app as a subprocess. The app itself reads input events on stdin

and writes a video stream on stdout; when the app exits, the parent process detects this and also exits. Then a window-managing shell running multiple apps as subprocesses would just be one more app you could run in this way.

I have some vague memories of worrying that if raw framebuffer programs crash I might have to reboot to regain control of the machine, although I don't remember if this has to do with changing keyboard modes or video modes or stty or what. Running them as subprocesses this way would permit the (hopefully more reliable) parent process to clean up properly.

This would also make it straightforward to do things like run multiple programs successively in a window, run an image output filter over the output of a program (e.g., Dark Mode, magnify, overlay ripples around mouse clicks, or some kind of pause/rewind thing), or write graphical programs in whatever random language that can spit out bytes.

Unfortunately, though, performance. As explained above, Linux charges too many computrons for moving pixels between processes in pipes. The Intranin design for IPC by transferring ownership of memory segments between processes would enable these things to be done efficiently.

For environments with a low screen update rate, such as e-ink displays, the efficiency concerns disappear.

Topics

• Programming (p. 3658) (286 notes)
• Graphics (p. 3483) (91 notes)
• Systems architecture (p. 3691) (48 notes)
• Graphical user interfaces (p. 3489) (23 notes)
• Protocols (p. 3668) (21 notes)

Capacitors: some notes on tradeoffs

Kragen Javier Sitaker, 2018-07-05 (5 minutes)

As representative electrolytics, let's choose the Panasonic ECA-0JHG102 and the Nichicon UWT1H470MCL1GS.

The Panasonic ECA-0JHG102 costs 44¢ down to 10.4¢ (quantity 5000). It's a 1000µF 6.3V 8mm-diameter 11.5mm-long capacitor rated for 380mA at 120Hz or 437mA at 100kHz. Its leakage current is 3 µA or "0.01 CV", presumably per second; this works out to 63 microcoulombs when fully charged, so probably 63 µA. The datasheet doesn't give ESL, ESR, or self-resonant frequency, but lists frequency-dependent characteristics only up to 100kHz. It lists the tangent of the loss angle, though.

The Nichicon UWT1H470MCL1GS costs 47¢ down to 19.4¢ (quantity 100). It's a 47µF 50V 6.3mm-diameter 7.7mm-long SMD capacitor rated for 63mA at 120Hz or 94.5mA at 10kHz, with exactly the same leakage current spec as the Panasonic part. Its loss-angle tangent is 0.14, and its frequency-dependent characteristics are listed only up to 10kHz. No ESL is listed or suggested.

As representative supercapacitors, the Nichicon JUWT1105MCD and the Elna DSK-3R3H204T614-H2L.

The Nichicon JUWT1105MCD costs 92¢ down to 30¢ (quantity 5000). It's a 1-farad 2.7V 6.3mm-diameter 10.5mm-long EDLC rated at 4Ω ESR at 1kHz. The datasheet lists a 4Ω "DCR", but I don't know what that is; surely not a discharging current resistance, since that would discharge it within seconds. The capacitance rating is based on discharging in, I guess, 270 seconds after a 30-minute (!) charge cycle.

The Elna DSK-3R3H204T614-H2L costs 195¢ down to 92¢ (quantity 500). It's a 200-millifarad 3.3V 6.8mm-diameter 1.4mm-thick EDLC that looks for all the world like a coin cell with strip terminals soldered onto it for a total thickness of 1.8 mm and a total width of 11.7mm. It's rated at 200Ω internal resistance. The datasheet mentions absolutely nothing about time or frequency, but it seems like these are actually intended as battery replacements.

Neither supercapacitor lists a leakage current rating.

The above capacitors are all rated for endurance of 1000 or 2000 hours, though at different temperatures. By contrast, as representative tantalum capacitors, let's consider the AVX TAJB226M010RNJ, the Panasonic ECS-H1AX475R, and the AVX TAP104K035SCS.

The AVX TAJB226M010RNJ costs 69¢ down to 23¢ (quantity 1000). It's a surface-mount 1411 (3.5mm × 2.4mm, 2.1mm high) or possibly 1210 (3.5 mm × 2.8 mm, 2.8 mm high) 22µF 10V conventional MnO_2 tantalum capacitor with a rated ESR of 2.4Ω at 100kHz. All the frequency-dependent stuff in its datasheet is just listed at 100kHz and no other frequencies. It claims to withstand a 13V surge voltage and have a failure rate of 1% per 1000 hours at 85°, if I understand correctly. Its leakage current is 2.2 µA. At low temperatures, it's rated for 188 mA ripple current.

The AVX TAP104K035SCS costs 66¢ down to 19¢ (quantity

5000). It's a through-hole 35V 0.1µF 7mm-high 2.5mm-diameter tantalum cap rated for a failure rate of 1% at 1000 hours at 85°. Its ESR is listed as 26Ω at 100kHz. It leaks 0.5µA.

What about multilayer ceramic capacitors? Let's take the Vishay A104K15X7RF5TAA, the Murata LLL219R70J105MA01L, the Murata GJM1555C1H4R7CB01D, and the Johanson 500R07S0R5BV4T as examples.

The Vishay A104K15X7RF5TAA costs 26¢ down to 13¢ (quantity 2500). It's a through-hole 50V 100nF X7R axial-lead capacitor, 2.5 mm diameter and 3.8 mm long. …

The Murata GJM1555C1H4R7CB01D costs 12¢ to 1.9¢ (quantity 5000) and is a 50V C0G/NP0 4.7 pF 0402 SMD MLCC. C0G/NP0 devices are optimized for high precision and low loss rather than high capacitance, although this one is still ±5%. It's tiny at 1 mm × 500 µm × 550 µm. …

The Murata LLL219R70J105MA01L costs 44¢ to 11.5¢ (quantity 1000) and is also a SMD MLCC, this one X7R 0508 and with reversed terminals for low ESL. It's 1µF and 6.3V. Its 0508 size is larger, at 1.25 mm × 2 mm × 850 µm. …

The Johanson 500R07S0R5BV4T costs 16.0¢ to 2.7¢ and is another C0G/NP0 0402 MLCC, this time of only 0.5 pF. This capacitance seems small enough that it would be likely to happen by accident, without an actual capacitor, but what do I know? The tolerances are proportionally rather loose: ±0.1 pF, which ends up being ±20%. …

Topics

Free software debugging

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

Chapter 2 of Andreas Zeller's book "Why Programs Fail" describes the following life cycle for a software problem that involves a bug fix:

- The user informs the vendor about the problem.
- The vendor reproduces the problem.
- The vendor isolates the problem circumstances.
- The vendor locates and fixes the defect locally.
- The vendor delivers the fix to the user.
- [implicitly] The vendor delivers the fix to other users.

One of the advantages of free software is that it often allows the following lifecycle instead:

- The user reproduces the problem.
- The user isolates the problem circumstances.
- The user locates and fixes the defect locally.
- The user publishes the fix so that other users can use it.
- The user informs the maintainer about the problem.
- The maintainer delivers the fix to other users.

When the problem is sufficiently important to the user, this is much preferable to the traditional cycle described above; the time from step #1 to step #3 can often be minutes or hours instead of weeks, months, or years --- and many problems never get fixed, either because they're put in place deliberately by vendors, or because they're not important to the vendor.

In addition to giving the user their bug fix more quickly, it also often reduces the total amount of work involved. Zeller's steps #1 and #2 often involve a lot of communication back and forth between the vendor and the user, as the user tries to supply enough information for the vendor to reproduce the problem, perhaps weeks or months after the fact, but without including any confidential information. An omitted step (in both sequences) is prioritization of the problem, which is dependent (in Zeller's sequence) on the information provided by the user; if it appears to be a problem that doesn't affect many people, the vendor is likely not to bother to reproduce and isolate it quickly.

Topics

Can you turbocharge the STM32 ADC to build an oscilloscope?

Kragen Javier Sitaker, 2018-07-14 (5 minutes)

The analog-to-digital converter on even the most basic models of STM32 is capable of a million samples per second, about 70 times faster than the 15ksps advertised for the ADCs on the 8-bit AVR line, which Arduino was built on. The "leakage" or bias current is specified as <200nA, which is adequate for many measurement tasks even without an external preamp. Some new "Arduino" boards are built around an STM32F103C8T6, which has two 1Msps ADCs, and thus ought to be able to digitize signals at 2Msps with a bit of clever programming — enough for signals of up to 1MHz. (ST application note AN3116 has details.) It has 20KB of zero-wait-state SRAM, which is enough to capture several thousand samples, which ought to be enough.

There are many possible uses for such a capability. One highly desirable use is to make an oscilloscope, a general-purpose instrument for visualizing a small number of quantities changing quickly, once the quantities are converted to voltages. (By "small number" I mean "less than five" and by "quickly" I mean "over nanoseconds to milliseconds"). Unfortunately, 1MHz is not enough — a low-quality off-brand analog oscilloscope is 10MHz, and a normal entry-level analog oscilloscope is 20MHz, which doesn't mean it can't detect faster signals, just that they suffer 6dB/octave attenuation and severe phase distortion. So 2Msps is about 5% of an oscilloscope.

Here's an exploration of possible ways to get to something a bit more decent.

Software flash conversion

The STM32 is capable of acquiring data at much higher speeds using its GPIO lines; it can read 16 of them at once, something like 36 million times per second. Configured as digital inputs, these are Schmitt-trigger lines; some are 5V-tolerant, with LOW guaranteed up to 0.475Vdd - 0.33 V, HIGH guaranteed from 0.5Vdd + 0.2 V, and ≈100mV hysteresis; and others are not, with LOW guaranteed up to 0.3Vdd + 0.07 V, HIGH guaranteed from 0.445Vdd + 0.398V, and ≈200mV hysteresis. This leaves a narrow voltage range of 500–800mV undefined. All have input leakages of 1µA or less.

It occurs to me that you could perhaps use this capability to acquire lower-precision data about higher frequencies, which might still be valuable. You could experimentally characterize the analog behavior of the chips, and perhaps generate waveforms for field-calibration of the chip. Then you could use a 16-bit digital input port and a 17-resistor ladder fed from a buffer as a 4-bit flash converter, and maybe do some kind of half-flash trick largely in software to get to 8 bits.

4 or even 8 bits may sound extremely unimpressive, but often the high-frequency signal is small in amplitude compared to the lower-frequency components; certainly when the signal contains both, the high-frequency components alone will have smaller

amplitude than the overall signal. In particular, this means it's safe to use an ac-coupled or inverting amplifier on the input. And flash conversion is easily capable of digitizing according to a nonlinear scale, such as ISDN μ-law. But the most interesting possibility is using a bit of external analog circuitry to dynamically adjust the range of the high-frequency digitization to approximate the range of the jury-rigged ADC.

An external fucking ADC obviously

Well duh. The TI ADC08060 is a 3V 8-bit 20–60Msps parallel-output ADC, and you can just hook it up to 8 GPIO pins and bingo. There are others out there. They barely cost more than a STM32F103, although other STM32 chips are cheaper.

Running the ADC at lower precision

As explained in Notes on the STM32 microcontroller family (p. 3176), it's possible to configure the ADC for lower bit depth (10, 8, or 6 bits) to speed it up (to 928, 785, or 643 ns, respectively), which suggests you could crudely digitize signals up to 780 kHz with one STM32 ADC, or up to 1.56 MHz with two.

Ganging up STM32s

At the 1Msps rate, each conversion takes 14 clock cycles of a 14MHz clock, of which only 1.5 cycles (107 ns) are spent sampling the signal. This means that frequency components up to about 9MHz are basically unattenuated, though aliased, in the output, and the first actual zero of the frequency response is 9.3 MHz. The STM32F103C8T6 can run its two ADCs in lockstep so as to sample precisely every 7 cycles, but you could easily imagine a scheme for syncing up 5–20 STM32s to sample at slightly staggered times, thus capturing signals up to 9MHz in their full glory.

Topics

Transactional event handlers

Kragen Javier Sitaker, 2019-01-24 (14 minutes)

I was thinking about event loops and transactional memory, and it occurred to me that applying transactional memory to event loops might solve their problems. In some sense this is just plain "use transactional memory" but it's a particular choice of where to draw the transaction boundaries: each event that would normally invoke an event handler in an event-loop system (network data, mouse click, I/O completion, timeout, or whatever) invokes it inside a transaction.

Event loops

One of the major concurrency models in modern software (and actually historical software going back many decades) is the event loop. A CICS pseudo-conversational transaction was initiated when a message came in from a terminal, and ran on the CPU until it yielded it; KeyKOS server domains synchronously awaited incoming invocations, then did some processing, usually sent a response, and returned to awaiting; Oberon's "central loop" keeps the processor engaged "continuously polling event sources"; Win16 and Win32 WndProcs are invoked once for each event on the window, handling it as they see fit, but always on the same thread; Node.js servers and browser JS and Tcl/Tk all use event loops, where a single-threaded process chews through a queue of events by invoking previously registered callbacks.

The event-loop model is efficient (no need to allocate memory to per-thread stacks), easy to understand, and relatively free from race conditions, since each event handler must run to completion before the next event handler can begin to run. In effect, the event handler has a lock on the entire memory space. In a system with a single event loop, as with other systems with a single lock, there's no danger of deadlock, because there's only one lock. In theory, multiple communicating event loops can suffer from deadlock, but this seems much less frequent in practice than system composed from threads and locks.

(CICS also supports a "conversational" model in which a task can suspend until it receives input, but that's not what I'm discussing here; after 51 years systems can get a bit muddled.)

Event-loop problems: parallelism and responsiveness

However, event-loop systems do have two major weaknesses: parallelism (in the modern sense of taking advantage of multiple processors) and responsiveness. Almost any event-loop system can take arbitrarily long to respond to any event, because a slow handler for a lower-priority event can prevent the system from even noticing a higher-priority event in time. This is especially a problem since event loops are only used in more-or-less real-time situations — if only computing a correct result matters, but not how long it takes, you can wait for all the input data to become available before you

start the computation, and then you don't need an event loop or any other kind of concurrency. Typical solutions to this problem include interrupts (re-introducing a pervasive risk of race conditions), watchdog timers that reset the entire system when a deadline is missed, and running separate event loops, especially for tasks of different priorities (creating the difficulty of communicating between them). The BeOS GUI famously had a much more responsive UI than other contemporary GUIs because of pervasive multithreading, i.e., running a lot of separate event loops.

A less-common solution to the responsiveness problem used in MOO and browser JS is to kill event handlers that hit a timeout. To prevent this from producing overall system instability, the timeout is only checked at "safe points", where all the invariants of some "system layer" have been re-established, and only mere user code is vulnerable to having its state corrupted. This works far better than you would expect, in part because that user code already has to be exception-safe, and a thrown exception has results similar to a handler being killed due to a timeout. However, it pretty much eliminates the simplicity advantage of event-loop programming over using threads and locks.

The least-common solution is to ensure that every handler in your event loop has bounded and acceptable worst-case execution time (WCET) that permits an acceptably fast response to the most demanding event. This solution does have the benefit of working, but it makes programming for the system extremely demanding.

Optimistically-synchronized transactional memory as a possible solution

As an alternative, what if we adopt the solution used by CICS back in the 1970s, and used by most web servers today? Let's run each event handler ("transaction" or "task" to CICS) in its own little universe, unable to access any shared state directly — it can only access shared state mediated by an ACID, or at least ACI, *transaction* ("unit of work" to CICS). The transaction logs all the data it reads and buffers all the data it writes, and ensures that its execution is *serializable*, in the sense that the end result of running some set of event handlers is the same as running them all one at a time in some order. This gives you the same freedom from race conditions as the simple event loop model, more or less by definition.

There are lots of different ways to achieve serializability, though. One way is of course to *actually* run all transactions in some order, or at least all transactions that write to shared data; you can do this by just running a single thread with one transaction at a time, with the problems described earlier, but if you have a lot of read-only transactions, you may be able to get some performance benefit with just a reader–writer lock, or, as SQLite3 does, a reader–writer–pending lock.

This is called "pessimistic synchronization" because it's working to prevent problems, rather than fix them. More sophisticated pessimistic-synchronization strategies with more locks that permit higher levels of concurrency are possible, but they all have the property that if any state is shared between high-priority code and low-priority code, it's possible for the high-priority code to have to

wait on the execution of the low-priority code. If the problem is merely scheduler priorities, you can solve this with priority inheritance, but if the problem is that the low-priority code just has too much work to do before it finishes, you don't want pessimistic synchronization, which is to say, you don't want locks, because you're back to the event-loop situation where any piece of code in your system can potentially make the response to any event too slow.

(There's also the risk of deadlock once you have multiple locks, although there are strategies that avoid this — acquiring all locks at transaction start in a deterministic order, for example. This breaks procedural compositionality, though, because you need to know all the possible locks any transitively invoked subroutine might need when you start the transaction.)

Pessimistically-synchronized transactions typically provide the ability for the user code to roll back the transaction at any time, for example in response to a violation of data integrity constraints. This prevents any of the transaction's writes from being saved; it's as if the transaction never happened, except for the error message returned. But, in general, this (or deadlock) is the only reason a pessimistic transaction can fail; other concurrent transactions have no influence on whether they fail or succeed, and they can be written to always succeed.

As Joe Duffy notes in his 2010 retrospective blog post about the canceled planned software-transactional-memory support in the .NET CIL, as long as everything you care about is in the transactional store, this also gives you a great way to recover from exceptions: any exception inside the transaction rolls back the transaction. However, you may need some kind of special debug interface to read the debug printfs from the aborted transaction — because, as noted above, you don't want arbitrary I/O from inside the transaction to be possible.

Optimistic synchronization, by contrast, can guarantee forward progress and worst-case execution times, but they can always fail. Optimistic synchronization works as follows: if, when a transaction is ready to commit, any of the data it read has been changed by another transaction, the transaction is rolled back and retried from the beginning, using the same code but the new data. This can be done at the microscopic level using multi-word compare-and-swap primitives, or you can do it at the system level with unbounded optimistic transactions.

Because optimistic synchronization detects conflicts after they've happened, instead of preventing them, any transaction may be aborted at any time for reasons outside of its control. This means that it is unsafe for it to do any I/O before it commits or, more generally, affect anything outside of the transaction-controlled universe — if it has to be rolled back and retried twice, you don't want it to send three emails or delete the same file three times (failing on the second and third attempts). (This property is why Microsoft canceled the .NET STM mentioned earlier.)

This gives us precisely what we need to fix event loops: you can guarantee the response time to a high-priority event simply by pausing all lower-priority transactions when they try to commit, so they can't write to the shared store, and then the high-priority event is guaranteed to complete in only the time it needs to execute its own code and do the required accesses to the shared store. If this conflicts

with any ongoing lower-priority transactions, that will be discovered once those transactions are resumed; they will be rolled back and tried again. Also, as long as you don't have interference and consequent rollbacks, you can get the full benefit of however many CPUs you're running on — unlike a CICS region, which uses cooperative multitasking between event handlers, all on a single thread.

Mutability

Transactional memory is not a great fit with pervasive mutability of shared data; most of the intractable problems Duffy mentions in the CIL implementation stem from pervasive mutability. Immutable ("persistent") data structures can be read safely, inside or outside transactions, without any chance that another thread is modifying them and thus with no need to log the reads and revalidate them later.

However, this really only works if you have a garbage collector, which is going to make it hard to guarantee responsiveness. Without a GC, whoever deallocates the data structure is "mutating" it. To some extent you might be able to keep the GC from interfering with high-priority tasks by ensuring that those tasks always have enough memory to run without invoking the GC, but it's also important to ensure that the GC doesn't hold any locks that the high-priority task might need to wait on, or if so that it releases them promptly.

To the extent that the GC only has to run over shared data that persists across transactions, though, the GC load may be light enough to never be a problem.

Composable Memory Transactions

There's a lovely paper called "Composable Memory Transactions" describing the Haskell STM with its "fail" and "orElse" constructs; these provide a way to use transactional memory to wait for arbitrary predicates to become true, and also to compose such single-event waits into multiple-event waits or nonblocking polls, and similarly to convert nonblocking polls into blocking polls.

The approach is very simple. First, for blocking, you have a "fail" construct which fails your transaction as if from interference from another transaction, prompting the transaction system to retry it. But it would be futile, though correct, to try it again immediately, since it will fail in the same way. So at least one of the things it read inside the transaction before failing needs to have changed before it will be able to succeed.

Waiting for a conjunction of such conditions is simple: you invoke each of the wait procedures, one after the other, inside of a transaction. Since the transaction is isolated, you are guaranteed that nothing changes from the beginning to the end of the sequence, so you know that all of the wait-predicates are simultaneously allowing execution to continue.

Waiting on a disjunction requires a new `orElse` construct, which recovers from the failure of a nested transaction by trying an alternative nested transaction. In effect, it composes two transactions into a single ordered-choice or disjunction transaction. For example, if the first transaction was an attempt to read from some queue X, while the second is an attempt to read from some queue Y, each

failing when its respective queue is empty, then the combination will fail precisely when both queues are empty. Alternatively, the second transaction could simply be some computation to carry out in the case where queue X is empty, thus transforming blocking into nonblocking polling.

I don't know why I started writing this section.

Topics

• Performance (p. 3621) (149 notes)
• History (p. 3500) (71 notes)
• Systems architecture (p. 3691) (48 notes)
• Latency (p. 3542) (19 notes)
• Transactions (p. 3755) (14 notes)
• Concurrency (p. 3386) (9 notes)
• Event loops

A formal language for defining implicitly parameterized functions

Kragen Javier Sitaker, 2019-09-05 (updated 2019-09-30) (29 minutes)

In Dercuano plotting (p. 2885) I talked about the need for a linguistic model of describing calculations to plot, that is, a programming language. I think the ideas I'm describing here are finally crystallizing, but I wrote about them previously in APL with typed indices (p. 3264), A principled rethinking of array languages like APL (p. 1995), Relational modeling and APL (p. 1217); additional sources of inspiration have been First impressions on using the μMath+ calculator program for Android (p. 195), Adam N. Rosenberg's "A Description of One Programmer's Programming Style", Darius Bacon, and the ZIMPL and GNU MathProg languages discussed in Some notes on the landscape of linear optimization software and applications (p. 1285), with the relationship explored in A principled rethinking of array languages like APL (p. 1995).

I'm pretty sure I'll have to rework some of this, but I feel like at this point I have not only something implementable but also a concrete use case where I can see how the existing designs are suboptimal.

The basic idea is that every expression evaluates not to an atomic value like 37 or 2.5 but to (looking at it in five different ways) a table or an array or an N-ary relation or conditional or function; and the particular atomic values we get out of it depend on the *circumstances* that obtain when we happen to look at it; and under some circumstances it may fail to have a value entirely. We can inspect these tables to see which circumstances are the relevant ones, and when the possibilities are finite, we can enumerate all the possible sets of circumstances and the corresponding atomic values, which is to say that we can reason counterfactually about what values the expression *would have had* under circumstances that do not in fact obtain at the moment.

Even if the possibilities are infinite, by stipulating all the relevant circumstances, we can reason counterfactually about what atomic value the expression would have had under those hypothetical circumstances.

As a very concrete example relevant to my plotting problem, given some value representing a function y that depends on an argument x, we can ask what value y would have if x were 0, or if x were a member of the list [-1, -0.5, 0, 0.5, 1]; in the latter case, the result is a finite table of five values that tell us what value y would have for each value of the index into the list. If y additionally depends on a parameter p, we can stipulate that p belongs to some other list, and y will provide a potentially larger table of results. This provides a strategy for getting good efficiency out of even a naive interpreter implementation, much like Numpy, Pandas, APL, Octave, or R; but it should be much easier to write correct code with, because of the rigorous logical underpinning.

Syntax

Because I'm planning to use a non-textual user interface, I'm not going to design a syntax; instead, I'll just use Lisp S-expression syntax in this document. It's not very readable syntax but it's adequately formal and flexible. Its execution semantics are not Lisp execution semantics; it just represents an ordered-tree structure.

Semantic model

The main objects of interest are atomic values, atomic types, aggregates, variables, environments, and expressions. The following is full of forward references, unfortunately; I should see if there's a better order to put things in.

Atomic values

Atomic values are things like 4 or -2.5 or 4+1j or :London or possibly 'w'; they correspond to values in many normal programming languages, but do not include any kind of aggregate data structure such as arrays, structs, dictionaries, sets, lists, tuples, or ML-style constructors. Like quarks, they are never observed alone; they are always part of an aggregate. There are different kinds of atomic values: integers, real numbers, opaque symbols like :London (which I will write with a leading : to distinguish them from variables), and maybe characters.

Atomic types

XXX confused ideas about what goes here: discrete vs. continuous? Finite vs. infinite? Bound vs. unbound? Bounded vs. unbounded? Nominal, ordinal, interval, and rational levels of measurement? Units of measurement? Can a single aggregate contain values of different types?

Aggregates

Aggregates are tables of atomic values, in the sense that they have rows and columns. The number of rows may be zero, any natural number, or infinite; there is always one "result" column, but there can be zero or more "key" columns. Each "key" column is associated with a variable, but the "result" column is not; it is only associated with the tuple of key-column variable values in its row. Conceptually neither rows nor columns have order, although by necessity they will be in some order when we serialize them. Aggregates are what expressions evaluate to, and what environments bind variables to. Here is an example aggregate:

```
(agg     (month oils)
         (1     :VEG1  110)
         (1     :VEG2  120)
         (1     :OIL1  130)
         (2     :VEG1  130))
```

This aggregate has two key columns, associated with the variables `month` and `oils`, and four rows, the first of which associates the atomic value 110 with the situation where the variable `month` has the atomic value `1` and the variable `oils` has the atomic value :VEG1, a symbol. A useful way of reading this is "110 if month == 1 and oils == :VEG1, else 120 if month == 1 and oils == :VEG2, else 130 if month == 1

and oils == :OIL1, else 130 if month == 2 and oils == :VEG1."

Here is an aggregate with no key columns, a *constant*, which associates the atomic value 4 with all possible states of the universe:

```
(agg () 4)
```

The key columns of an aggregate are its free variables.

XXX Should we remove infinite aggregates? They are the only way to represent things like "matrix-vector multiplication" as values, but they seem significantly different from other kinds of values.

Variables

Variables are opaque, indivisible names, like `x`, `VC`, `month`, or `oils`. In themselves they serve only to be distinguished from one another, but they are crucial links between aggregates, expressions, environments, and atomic values.

Environments

Environments are sets of key-value pairs in which the keys are variables and the values are aggregates. Here is an example environment:

```
(env (month (agg () 1))
     (oils (agg () :VEG1))
     (oilhardness (agg (oils) (:VEG1 8.8) (:VEG2 6.1))))
```

This associates the variables `month` and `oils` with constant aggregates and the variable `oilhardness` with an aggregate with two rows and two columns. You might think that this second aggregate contains some extraneous data, describing as it does the `oilhardness` of a situation that we know is not currently the case (when `oils` is `:VEG2`), but as we will see, this counterfactual data can be very useful.

Expressions

Expressions describe computable functions; given some input data, in the form of an environment, they evaluate to output data, in the form of an aggregate. But there are other things we can do with expressions other than evaluate them; we can query them to find out what free variables they require in a given environment and what they require of those free variables.

Here is an example expression:

```
(+ (* x y)
   (let (oils (agg () :VEG2)) oilhardness)
   (sum (month) production))
```

XXX should we require an explicit "in months" in case production fails to depend on month in a useful way so we don't know what to sum over? Should there be some kind of explicit or implicit association between domains and variables, to allow inferring a universe in the absence of an explicit value?

XXX I'm being a bit fast and loose including an aggregate literal inside the expression. I might not want that to be legal.

It has the free variables `x`, `y`, `oilhardness`, `month`, and `production`, so it cannot be evaluated in an environment without an aggregate

associated with each of these variables. Also, it may inherit some free variables from the aggregates associated with those variables; for example, if `production` is associated with an aggregate that depends on the variable `city`, then the expression as a whole also has `city` as a free variable. However, there are two binding constructs in this expression, `let` and `sum`, which bind the variables `oils` and `month` in their respective argument expressions `oilhardness` and `production`, which prevents those free variables from bubbling up further. Indeed, if the expression is evaluated in an environment that provides values for these free variables, these binding constructs will prevent those values from penetrating.

This sort of bubbling up is why the set of free variables of an expression depends on the environment; it amounts to a sneaky sort of dynamic scoping. XXX can this dependency be expressed in a usefully simple way to satisfy the needs of plotting to figure out what kind of creature it's going to have to plot?

The aggregate that results from evaluating an expression in an environment has the same set of free variables the expression had in that environment, and the tuples taken from its key columns will be some subset of the cross-product of the result columns of the aggregates supplied for those free variables in that environment. However, the expression may produce some arbitrary subset of those rows rather than all of them.

XXX could it be that the expression had free variables that aren't used?

Expression types

Literal aggregates

Although I'm not sure if this is the right thing, for the moment I'm going to allow literal aggregates as expressions. Moreover, constants such as 4 and `:VEG1` are also valid; they represent constant aggregates like `(agg () 4)` and `(agg () :VEG1)`

Arithmetic

The standard set of computer arithmetic operations are provided with their usual meanings: +, -, * for multiplication, / for division, % for remainder, ** for exponentiation, >>, <<, abs, exp, ln, pow, sin, cos, tan, asin, acos, atan, ceil, floor, round, bitand, bitor, bitnot, and bitxor. I mean this might change a bit but that's what I think right now.

These operations take one or more arguments, which are evaluated in the same environment the arithmetic operation is, and the aggregates thus produced are passed to the arithmetic operation as actual parameters. Their free variables are the union of the free variables of their operands.

These operations operate purely pointwise and generate a full Cartesian product output. In any environment, `(+ 3 4)`, syntactic sugar for `(+ (agg () 3) (agg () 4))`, will produce `(agg () 7)`. In an environment lacking `x`, `(+ (agg (x) (1 3) (2 5)) 4)` will produce `(agg (x) (1 7) (2 9))`, and

```
(+ (agg (x) (1 3) (2 5))
   (agg (x) (1 4) (2 10)))
```

will produce `(agg (x) (1 7) (2 15))`. In an environment lacking both `x` and `y`,

```
(+ (agg (x) (1 3) (2 5))
   (agg (y) (:a 4) (:b 10)))
```

will produce `(agg (x y) (1 :a 7) (1 :b 13) (2 :a 9) (2 :b 15))`. (At this point a reminder seems in order that the sequence of rows and columns in the aggregate is arbitrary and insignificant.)

XXX note that this contradicts the note earlier saying that expressions could not be evaluated in an environment that didn't provide aggregates for all their free values.

XXX rename 'aggregate' to 'table'? It's too long and operations like "sum" and "min" are usually called "aggregate operations".

The expression `(+ x y)` cannot be evaluated to a finite aggregate if either `x` or `y` is missing from the environment, but consider the environment

```
(env (x (agg (p) (0 3) (-1 5)))
     (y (agg (q r) (20 30 4))))
```

in which `(+ x y)` will evaluate to `(agg (p q r) (0 20 30 7) (-1 20 30 9))`, with three free variables bubbling up from the bindings of `x` and `y`. If we further augment the environment, we can prevent `q` from remaining free:

```
(env (x (agg (p) (0 3) (-1 5)))
     (y (agg (q r) (20 30 4)))
     (q (agg () 20)))
```

This causes `(+ x y)` to evaluate to `(agg (p r) (0 30 7) (-1 30 9))`.

It will be seen that an arithmetic expression containing free variables defines a function taking those variables as arguments.

`let`-expressions

The expression `(let assignments expr)` allows invoking a function with specific arguments or composing two (or more!) functions. During evaluation, it constructs an environment by augmenting its own evaluation environment with new associations taken from `assignments`, which is a list of alternating variables and expressions. These expressions are evaluated in the outer environment, and each one is associated with its corresponding variable in the new environment.

The let-expression's free variables are the free variables of `expr`, minus the variables assigned to in `assignments`, plus the free variables of the expressions in `assignments`, which might reasonably add variables back in that were dropped in the second step.

So, for example, `(let (x 3 y 4) (+ x y))` evaluates as before to `(agg () 7)`, because whatever the outer environment is like, the inner environment has `x` bound to `(agg () 3)` and `y` bound to `(agg () 4)`; and it has no free variables.

A more interesting example is `(- x (let (i (- i 1)) x))`. If `x` has `i` as a free variable and is defined for `i` integer, the inner `x` can evaluate to different atomic values from the outer `x`, and you get the backward differences of `x` along the `i` axis. (I said at the top that I wouldn't

invent syntax, but if I were inventing syntax, I might write this as `x - x[i=i-1]`.)

Conditional expressions

The expression `(if x y)` operates very similarly to the arithmetic expressions, in that its free variables are the union of the free variables of expression `x` and those of expression `y`, it evaluates them both in its own environment, and it combines the `x` and `y` results pointwise. However, it produces, in general, less than the full Cartesian product of their results; the rows where `x` produced false are removed.

So, for example, `(if (> x 1) (if (< x 3) (* x x)))` is the function x^2 limited to the domain (1, 3).

The expression `(case e1 e2 e3 ...)` evaluates all the expressions `e1`, `e2`, `e3`, etc., in the same environment it was invoked in, but then combines them in the fashion of SQL's `coalesce` or the `|` alternation operator in Icon or Unicon: rows are taken from `e2` only for the case where `e1` did not produce a value, from `e3` only where `e2` failed, and so on. This means that you could write the absolute-value operator, for example, as

```
(case (if (> x 0) x)
      (- x))
```

The first expression produces `x` whenever `x` is greater than 0, but for x less than or equal to zero, it produces no value, so `case` fills in the empty space with the result of the `-` operator.

Note that this can be misleading; if you write `(case (if p q) r)`, even if `p` is always true, that is no guarantee that you will never see values from `r`, because `q` might fail of its own accord, aside from getting externally failed by `p`.

(Again, if I were inventing syntax, I'd want to write that as `x > 0 -> x | -x`.)

Another use of `case` is augmenting the first-differences example above with the initial value of the sequence `x`:

```
(case (- x (let (i (- i 1)) x)) x)
```

Here we suppose that when `i` is at its minimum valid value, such as 0, then `(let (i (- i 1)) x)` will fail, leaving a hole that can be filled with `x`.

XXX for efficient evaluation we need to be able to tell which dimensions are going to stay rectangular.

iota

The expression `(iota var expr)` evaluates to `(agg (var) (0 0) (1 1) (2 2) ... (n n))`, where `n` is one less than the value to which the expression `expr` evaluated.

XXX this sucks compared to writing `1:10` in Octave or R. You have to name a variable! Moreover you still have to index it.

Standard reductions

The expression `(sum (month) production)` produces a result that is the same as the result of evaluating `production` except that all the rows differing only by `month` have been summed together, and the `month` key column has been removed. This generalizes in obvious ways to

multiple key columns and to the other standard reductions `product`, `min`, `max`, and `mean`.

XXX this means that `month` does escape as a free variable, but the result won't depend on it, and there's no way to get a sum over more months than you have some other iteration variable to throw away; this is surely the wrong semantics, and the one explained earlier is surely better. Given some of the evaluation semantics explored above, it could probably get the valid months out of `production`. But then how do you get a sum over only certain months? Maybe `(sum (mi) (let (month (agg (mi) (1 3) (2 5) (3 11))) production))`, maybe with some syntactic sugar like `sum{mi} production[month[mi]=[3 5 11]]`?

Monoidal prefix sums

XXX

Non-monoidal reduction

The expression `(for var1 seq var2 start var3 reducer)` expresses a general definite loop. It evaluates `seq` in its surrounding environment and produces an aggregate "the sequence" with, hopefully, the variable `var1` free. It also evaluates `start` in its surrounding environment and produces "the initial state". Then it proceeds sequentially along the sorted values of `var1` in the domain of the sequence, evaluating `reducer` once for each sequence item, in an environment with `var2` bound to the sequence item and `var3` bound to the previous result of `reducer` or, on the first iteration, the initial state.

XXX wait, the loop state has to be a single atomic value?

XXX "domain" and "range" are better than "key" and "result" column; "domain variable" is maybe better than "free variable" for aggregates.

Variable capture

As described above, the model has the same variable capture problem as Lisp macros, only much worse.

Consider this expression for describing the L_p norm of a vector:

```
(** (sum (i) (** x p)) (/ 1 p))
```

This relies on the fact that the `i` introduced by the `sum` is visible to `x` so that it can select the elements of `x`. Similarly, consider this code to evaluate a polynomial at a point (inefficiently):

```
(sum (i) (* a (** x i)))
```

This relies on implicitly indexing `a` with `i`. But consider this expression for describing matrix-vector multiplication:

```
(sum (k) (* (let (j k) a) (let (i k) x)))
```

This has a free variable `i` which ranges over the columns of `a` (or rather its `j`-indices). To do the summation, we need to introduce a new, fresh loop variable `k`; it is our intention that the only two uses of `k` be indexing the `i`-indices of `x` and indexing the `j`-indices of `a`.

But what happens if `a` or `x` internally has a free variable, which is to say an argument, `k`? Disaster strikes! Instead of bubbling up to the sum expression as a whole, it becomes part of the iteration, with

bizarre results that will potentially be difficult to debug.

On the other hand, if we change the language's scoping to purely lexical, constructs like `(let (j k) a)` make no sense --- lexically `a` does not contain `j`, so that expression would be equivalent to just `a`. And `(** (sum (i) (** x p)) (/ 1 p))` relies on the visibility of the index `i` to the vector `x`.

This is more or less just the standard problem of variable capture that occurs in Lisp macros or with dynamic scoping, but it's much more severe in this context, because of the pervasive use of free variables for implicit parameter passing through many stack levels.

The usual Common Lisp approach to solving the problem would be something like this:

```
(let ((k (gensym)))
  `(sum (,k) (* (let (j ,k) a) (let (i ,k) x))))
```

Alternatively, but awfully, you could use a name less likely to collide:

```
(sum (*matrix-multiply-index*)
  (* (let (j *matrix-multiply-index*) a)
     (let (i *matrix-multiply-index*) x)))
```

Alan Bawden and Jonathan Rees's "syntactic closures" mechanism offers an relatively simple solution that is not difficult to implement (see also Chris Hanson's shorter 1991 proposal and the superb Scheme Wiki page, and also Stephen Paul Carl's 1996 master's thesis on an extension of it); Scheme unfortunately standardized on Will Clinger's "Macros that Work" instead, which is a much more complex mechanism to implement.

On the other hand, because of the evaluation semantics outlined above, a different kind of variable capture that occurs in Scheme is absent here. Bawden and Rees give the example

```
(define-macro (push obj-exp list-var)
  `(set! ,list-var (cons ,obj-exp ,list-var)))
...
(let ((cons 5))
  (push 'foo stack))
```

in which the expression containing the macro invocation expands to

```
(let ((cons 5))
  (set! stack (cons 'foo stack))
```

which will give an error due to attempting to invoke the number 5 as a function. I assert without demonstrating that the corresponding problem does not arise in this new language.

XXX does it?

A simple direct fix for the matrix-vector-multiply problem is to have a `fresh` or `new` or `my` annotation that generates the equivalent of a syntactic closure or gensym:

```
(sum ((my k)) (* (let (j k) a) (let (i k) x)))
```

However, this is bug-prone; code that unintentionally omits the `my` tag will function correctly, perhaps for years, until used in a context where variable capture occurs, which may be a result of a change to code that doesn't even explicitly invoke the matrix multiply or whatever, even renaming a variable or something. A different alternative is to make the variables introduced by expressions like `sum` implicitly lexical, but continue to treat `let` variables as dynamically-scoped; this would leave the matrix-vector multiply as I had originally written it, but the Minkowski *p*-norm code would change from

```
(** (sum (i) (** x p)) (/ 1 p))
```

to the perhaps more gnomic

```
(** (sum (i) (** (let (i i) x) p)) (/ 1 p))
```

and change the polynomial-evaluation code from

```
(sum (i) (* a (** x i)))
```

analogously to

```
(sum (i) (* (let (i i) a) (** x i)))
```

(I reiterate that the S-expression form shown here is not intended as the syntax for interactive use, but just as an unambiguous representation of the internal tree structure; perhaps on the display this would be written as $a_{i=i}$ or abbreviated in something like the conventional way as a_i.)

However, there's still the possibility of unintended variable capture in an expression such as

```
(let (h (* n dh)) (/ (- (let (x (+ x h)) f) f) h))
```

where we are really just using `let` to factor out a common subexpression, rather than to pass arguments to code or index data lexically defined elsewhere.

A third alternative would be to require every introduction of a new variable by `let`, `sum`, and the like, to be explicitly marked as statically or dynamically scoped, but this seems like it would still be bug-prone --- a variable incorrectly marked as dynamically scoped would still be a bug. You could catch this by requiring that the dynamically-scoped variable *not be used statically*, though.

A fourth alternative would be to declare the variables at some lexical scope, in some cases module scope (where, in the context of Dercuano plotting (p. 2885), a module might be a note or a plot). Then, an expression like

```
(** (sum (i) (** x p)) (/ 1 p))
```

would be referring to some `i` that was already in scope, perhaps imported from another module or perhaps declared in the same module, and would be a compilation error if there were no such

variable, so in theory you would know that you were capturing a variable that could be used as a parameter; while on the other hand an expression like

```
(** (sum ((my idx)) (** (let (i idx) x) p)) (/ 1 p))
```

would be introducing a new `idx` variable. This seems simultaneously more awkward and less safe than most of the alternatives above.

Extradimensional features

In the UI, you can write a table with some key columns and some value columns; each value column becomes a new aggregate, stored under a variable with the name in the column header, indexed by all the key columns. If you don't have key columns, a nominal invisible key column is generated.

The top level of the UI defines an environment, or a sequence of environments, within which each expression is evaluated.

XXX does this mean it does some kind of topological sort to come up with an evaluation order?

When you define an expression with a single continuous free variable, the UI by default plots its value against that free variable in a 2-D plot, heuristically picking an initial range that seems likely to be interesting (which you can then interactively zoom and pan). If you define an expression with two continuous free variables, it by default generates a contour plot in a similar way, with a 3-D heightfield plot a click away. Discrete free variables that belong to some numerical range get, by default, a lollipop plot; discrete free variables that are merely nominal result in a labeled small-multiples display, switchable with a click to a single plot with multiple lines (or lollipops or whatever). Tabular results display is always available, but only enabled by default for small tables.

The UI uses the same kind of introspective magic for this that `sum` uses to figure out the possible values that `month` can take on in `(sum (month) production)`. This allows it to distinguish between discrete and continuous parameters based on how they're used, a sort of type inference, find out what their valid values would be, and measure their limits.

Automatic differentiation

How can we squeeze reverse-mode automatic differentiation into this model?

A first prototyping step

The above has made it clear that there are a variety of issues that aren't crystallized yet, but since I've been thinking about this for six years, I probably shouldn't expect them to crystallize anytime soon without more intensive effort. Instead, maybe I should write enough code to get some experience with a subset that *is* crystallized, and maybe try two or three different approaches to the other aspects and see which ones work best.

The most basic core is a few kinds of plots (linegraphs, scatterplots, lollipop plots, contour plots, and heatmaps, at least, and I'd like some 3-D surface plots), pointwise arithmetic (*, **, -, +, /, abs, exp, sin,

binary min, binary max), conditionals ("where"), extracting the free variables from an expression, and composition (indexing). I think summing along a dimension is the first sort of uncertain thing to add.

Topics

JIT-compiling array computation graphs in JS

Kragen Javier Sitaker, 2017-07-19 (1 minute)

Numeric loops running over large arrays are among the easiest kinds of code for JIT compilers to optimize.

Numba is a popular BSD-licensed Python library for LLVM code generation for numerical code; among the things it can do are to compile graphs of Numpy array operations into efficient machine code, including GPU code.

However, Python itself doesn't have a general JIT compiler; Numba is a limited-scope attempt at one.

JS does have a general JIT compiler, several of them actually (SpiderMonkey, V8/Crankshaft, and JavaScriptCore). It doesn't, unfortunately, have operator overloading. But maybe building computation graphs of arbitrary-dimensional arrays would be a reasonable thing to do in JS; then you could compile them into JS code, from which the existing JIT compilers could generate efficient CPU code, and maybe you could write a new compiler to compile the computation graph into GPU code, whether using WebGL in browsers or something else.

An initial hack at generating numerical code for V8 (sweetdreams-js.js) yielded disappointing results. But it was straight-line, loop-free code, so it may not be applicable.

Topics

- Programming (p. 3658) (286 notes)
- Arrays (p. 3326) (17 notes)
- Compilers (p. 3383) (16 notes)
- JS (p. 3533) (12 notes)
- JIT compilers

Pythagorean cement pipes for your shower singing

Kragen Javier Sitaker, 2019-09-08 (updated 2019-09-09) (7 minutes)

How could you architecturally encourage singing in the shower? Bathroom resonance sounds great, but it's not always in tune, and often there aren't enough resonances to sing any but the simplest tunes in resonance.

So, perhaps you could provide carefully tuned resonator cavities that resonate pleasingly at in-tune frequencies; for example, concrete pipes of different lengths, sufficiently isolated from the room air as to have a reasonable Q, but sufficiently coupled to it that they can pick up a note in a reasonable period of time; something like $Q = 10$ or $Q = 20$ (half-power points a half-step apart) should be adequate.

Helmholtz resonators like the ocarina are another, more scalable possibility, but they only resonate at a single frequency. Not only does this deprive you of overtones, it also means you need a separate set of resonators for each octave.

The presence of such resonators would also enable you to play a tune on them without singing, just by energizing the resonators, for example by hitting them with your hands, by hitting them with Blue-Man-Group floppy paddles, by banging them with hammers, or by setting firecrackers off in them.

The Pythagorean pentatonic scale

In some sense the simplest musical scale in common use is the Pythagorean major pentatonic scale, consisting of these intervals from the tonic in each octave (assuming the tonic is C):

- 1:1, the perfect unison or 黄钟, no semitones, C;
- 9:8, the major second or supertonic or epogdoon or 太簇, two semitones, D;
- 4:3, the perfect fourth or epitriton or roughly 中吕, five semitones, E;
- 3:2, the perfect fifth or hemiolion or 林钟, seven semitones, G; and
- 27:16, the major sixth or 南吕, nine semitones, A.

(The Greek names are from Plato's Republic and Timaeus.)

In some other tunings, 27:16 for the major sixth is replaced by 5:3, but the commonly-used equal temperament is only about six cents away from the Pythagorean 27:16, but 16 cents away from 27:16.

I'm not sure exactly how you should combine these Pythagorean intervals with ISO 16 A440 concert pitch, but one way to do it would be to use the equal-temperament A440 frequency for C as the tonic for each octave. So to get a C, you take $2^{-2/12}$ of 440 Hz (since C is two semitones above A) and get 392.00 Hz. Then you can go down by octaves from there: 196.00 Hz, 97.999 Hz, 48.999 Hz. That's probably deep enough for singing, since the tubes will also resonate at harmonics; so your first octave is 48.999 Hz, 55.124 Hz, 65.333 Hz, 73.499 Hz, 82.687 Hz.

```
(440/2**(2/12)/2/2/2*numpy.array([1/1, 9/8, 4/3, 3/2, 27/16])).round(3)
```

I could be wrong here, but I think that for a pentatonic rather than diatonic or chromatic scale, the advantages of equal temperament do not really come into play, and for singing in the shower, the advantages of just intonation may be more important. But in that case it might be better to use 5:3 rather than 27:16 for the major sixth; the medieval tradition of using 27:16 for this interval led to theorists considering it unusably dissonant, and of course Pythagoras himself was tuning tetrachords, which is why there's no Greek name for it above.

Air-column resonators

Wikipedia says

> Nodes occur at fixed ends and anti-nodes at open ends. If fixed at only one end, only odd-numbered harmonics are available.

Having a node at one end of a tube and an antinode at the other end means we only need a half-wavelength, which is nice, because the speed of sound in sea-level, room-temperature air is about 343 m/s, though it can increase by up to 0.6% with humidity, and vary considerably with temperature (`definitions.units` says 331.46 m/s in dry air at STP). So a whole wavelength at 48.89 Hz is 7 m! (And that's why a tuba has so many curls.)

A half-wavelength one-end-open tube at each of these frequencies would be 3.500 m, 3.111 m, 2.625 m, 2.333 m, and 2.074 m. But those tubes won't resonate at an octave, since it isn't an odd harmonic, so you need five more for the next octave, which is the one people most commonly talk in: 1.750 m, 1.556 m, 1.313 m, 1.167 m, 1.037 m. Alternatively, you could have another set of tubes of the same length, but open at both ends to raise the pitch by an octave — curved around to open again into the bathroom, of course.

Speed of sound variation

Wikipedia says the speed of sound in dry air around room temperature is 331.3 m/s + 0.606 θ m/s, where θ is the temperature in °C, or more accurately 331.3 m/s $\sqrt{(1 + \theta/273.15)}$. This is larger than the variation with humidity, which will also be a consideration when singing in the shower. Long, narrow concrete pipes will tend to slow down variation in both temperature and humidity.

So it might be best to use an average temperature of, say, 20°, or whatever is likely to be the average temperature of the house containing the bathroom; using the 331.46 number from `definitions.units` we get 343.38 m/s, with numbers ranging from 340.43 m/s at 15° to 349.19 m/s at 30°, about 44 cents (0.44 half-steps) of tuning variation.

```
331.46 * (1 + numpy.array([0, 15, 20, 25, 30])/273.15)**.5
```

Closing the mouths of the tubes a bit, so the aperture into the room is smaller than the body of the tube, would reduce the variability of temperature and humidity inside the tube by impeding air circulation. However, because of the wavelength-dependent participation of air just outside the mouth of the tube in the resonation process, this will also exacerbate the detuning of the higher harmonics. Perhaps a better solution is to flare the mouths of the tubes like brass

instruments to counteract this detuning effect, then cover them with something fairly acoustically transparent like aluminum foil or waxed paper.

However, you aren't going to notice the temperature-driven detuning of the pipes unless their Q is higher than about 40, so maybe you could just not worry about it.

Topics

Parsing a conservative approximation of a CFG with a FSM

Kragen Javier Sitaker, 2015-09-03 (7 minutes)

We know we cannot parse context-free languages with a finite state machine; this was central to Chomsky's destruction of behaviorism in the 1950s. But we can do a pretty good approximation.

All the ideas in here are due to Björn Höhrmann, who explored them in some depth in parselov, which is an implementation of these ideas in JS, plus a bunch of other ideas I don't understand yet but which are probably even better. (That is, unless I accidentally invented something and didn't notice. But I think all the ideas here are his.)

Consider this algebraic-expression CFG:

```
e where
e ::= e "+" f | e "-" f | f
f ::= t "*" f | f "/" t | t
t ::= "-" t | "+" t | number | variable | "(" e ")"
```

If we omit the last alternative, which recurses back to the beginning, we have an entirely regular language. All of the other recursions are either left-recursions or right-recursions, and so we can rewrite them as loops; for example, the alternatives for `e` add up to a sequence of `f` separated by "+" and "-" signs.

The final case, though, requires parentheses to match, which is a thing that a regular expression cannot guarantee.

But let's consider the "unstructured control flow" of the "subroutine calls". We can arrive at the beginning of `f` from any of five callsites (three in `e` and two in `f` itself) and, when we get to the end of it, we can return to any of those sites. In this way we can mechanically convert the entire grammar into a finite state machine. Of course, this finite state machine loses some information — it does not know whether it got into `e` from the top-level parsing or due to a recursion inside `t`. So it will match "(4" where it shouldn't. Nevertheless, it will not match "+*4" or "4 x". It's guaranteed that if it doesn't match a string, the full CFG cannot parse it either.

We can make it more precise by duplicating rules that are called more than once. For example, this CFG parses the same language, but if converted to a FSM in the same way, will no longer match "(4":

```
e where
e ::= e "+" f | e "-" f | f
f ::= t "*" f | f "/" t | t
t ::= "-" t | "+" t | number | variable | "(" e2 ")"
e2 ::= e2 "+" f2 | e2 "-" f2 | f2
f2 ::= t2 "*" f2 | f2 "/" t2 | t2
t2 ::= "-" t2 | "+" t2 | number | variable | "(" e ")"
```

That's because `e2` can "return to" the point in `t` before the ")", or the points in itself before the "+" and "-", but not the final accepting state.

It would have been equally valid for the `e` in the final `t2` to invoke `e` rather than `e2`, but it would have yielded a different approximation. The grammar as modified above will convert to an FSM which tracks whether to expect an even or odd number of closing parentheses. If instead we had recursed into `e2`, it would track whether it had never seen a left parenthesis.

It is not obvious to me how to choose which nonterminals should be duplicated in this way. These rules occur to me, in order to get a more precise regular approximation of the context-free language:

• There is no need to duplicate a nonterminal invocation that does not form part of a recursive loop, mutual or otherwise. You can attempt to stratify the nonterminals to figure out which calls form part of such a recursive loop, but this is not guaranteed to give you a minimal set of invocations.
• There is no need to duplicate immediately-left-recursive or immediately-right-recursive invocations, because they can be handled precisely by a FSM without more effort.

Of course, this process can be repeated.

It also occurs to me that if the FSM, upon entering a state with more than one predecessor state, records which predecessor state it entered from, then it is recording all of the candidate derivations it has discovered. To the extent that we elide ε-edges, we end up multiplying these cases, but in the simple case, we have one derivation per edge. Considering parsing the string "(4)" with this:

```
e where
e ::= e "+" f | e "-" f | f
f ::= t "*" f | f "/" t | t
t ::= "-" t | "+" t | number | variable | "(" e ")"
```

we, at the end, transition from after the ")" nondeterministically to all of the five callsites of `t`, from two of those nondeterministically to all of the five callsites of `f`, and from three of those to all of the four callsites of `e`, one of which is in fact the top-level `e` invocation, which accepts the string.

If we duplicate each nonterminal for each callsite, this process becomes more precise.

In a sense we could say that we are multiplying the states of the nondeterministic FSM by labeling them with some finite summary of the stack (or continuation) that a nondeterministic PDA would use if it parsed the grammar in the obvious way: the sequence of "return addresses" it would have. (Of course the PDA is an FSM if you leave out the stack.) The important thing is that we are mapping the infinite set of states-of-the-stack states to a finite set of states, and we're doing it in such a way that we don't need to magically recreate information that we erased — if we forget whether the number of right-parens we're expecting is odd or even, we can't define our mapping in such a way that we have to later recover that datum. (We can, however, nondeterministically guess it!)

That said, though, there are a wide variety of possible mappings of stack-states to finite states. I've talked about some of the possibilities above, although it was phrased in terms of duplicating productions.

A consideration for practical implementation of these concepts is that we would, ideally, eventually reduce the FSM to a deterministic FSM, so that we can execute each step on a real (deterministic) computer in constant time. For this we would like to avoid the potential exponential blowup of the powerset of the set of NFA states into DFA states, and counterintuitively it turns out that erasing *less* information about the stack can actually help to *reduce* the number of states.

Topics

License-free femtowatt UHF radio transceiver ICs under a μJ per bit

Kragen Javier Sitaker, 2016-09-19 (5 minutes)

Looking at the US$4.01 NXP MKW01Z128 that Bill Paul mentioned. This chip's RF interface is really interesting because it can transmit license-free as low as 290 MHz (1.03 m) at up to 600 kbps. It has 128 KiB of Flash and 16 KiB of RAM and a 48 MHz Cortex-M0, sucking 16 mA when actively receiving (and up to 95 mA when transmitting) at 1.8 to 3.6 volts. It's targeted at last-mile metering and wireless sensor networks.

Radio communications characteristics

Its power output is -18 to 17 dBm (50 mW!) and its sensitivity is -120 dBm (a femtowatt!) and so it occurred to me that maybe we should measure solid angles in dB. In this case, without further amplification, the receive antenna needs to capture signal over -137 dB (or more) of the sphere around an isotropic transmit antenna.

(The -120 dBm is dependent on lowering the communication rate to 1200 bps.)

A half-wave dipole antenna, the ideal, would be 516 mm long for 290 MHz; it gives you 2.15 dBi of "antenna gain", so you get down to a solid angle of -139 dB, and it captures signal over about 0.2 m², I think. This subtends -139 dB of solid angle at about 1000 km.

That's pretty impressive — two such chips can communicate over 1000 km with nothing between them but half-wave dipoles, and furthermore without a license.

On Spaceship Earth, though, it's a little tricky to have nothing between them, and 1000 km is far too short for moonbounce. If you were to use a 30 dBi dish antenna to transmit, that would get you to about 30 000 km, which isn't even all the way around Spaceship Earth; if you use another one to receive, that gets you to 900 000 km, which is far enough for moonbounce — but then you only have about 3 dB of headroom, and the moon sucks most of that up; typical moonbounce path losses are around 240 dB.

The 315 MHz UHF unlicensed (ultra-low-power/short-range device in US and Japan, not ITU) frequency band it uses (which extends down to 290 mHz, at the top of the VHF band) is mostly used by garage door openers, keyless car openers, and whatnot. It should have reasonable building penetration, better than cellphone signals.

E-skip ionospheric propagation apparently doesn't reach 290 MHz (250 MHz seems to be the limit), but it might be subject to tropospheric ducting from atmospheric temperature inversions, and to transequatorial propagation, and otherwise is limited to 3570 m √(h/m) line-of-sight radio horizon. Reaching 1000 km with 3570 m √(h/m) would require a stratospheric balloon or drone at 79 km altitude. A terrestrial structure like my office at about 60 m only has a line of sight of around 30 km. (Probably all the people remotely

unlocking their cars would provide too much interference, but maybe not.) Totally unsurprising structures like trees might be only 10 m tall, and thus have a line of sight up to about 10 km.

Tropospheric ducting, if it's possible, has the additional advantage that the strength of radio waves trapped in the atmospheric duct only drops off as the reciprocal of distance, rather than its square.

If you were to try to do something cute and moonbouncelike, maybe you could use a mountain. For example, a few kilometers from Las Cruces, Organ Needle reaches 2700 m above sea level, while Las Cruces itself is at only 1200 m. A dish (or radome) pointed at Organ Needle would effectively transmit isotropically from 1500 m, giving it a line-of-sight range of over 100 km, as long as the reflected power was high enough.

Power usage

16 mA × 3.6 V / 48 MHz works out to about a nanojoule per instruction. 95 mA × 3.6 V / 600 kbps works out to 570 nJ per bit.

17 mA at 1.8 volts is roughly the power supplied by the solar cell from a solar calculator, and just about at the right voltage already; two such cells in series with a capacitor ought to provide plenty of power for the device as long as it's only transmitting with a relatively light duty cycle.

Topics

Kinect modeling

Kragen Javier Sitaker, 2016-09-16 (1 minute)

So I have a bunch of depthmaps (aka range images) of a sculpture, acquired with a handheld Kinect. Now I want to generate a 3-D model of them.

http://pointclouds.org/documentation/tutorials/in_hand_scanner.php#in-hand-scanner (dependent on color; uses ICP) and http://wiki.ros.org/rgbdslam are two ROS-related things that do this kind of thing.

http://pointclouds.org/documentation/tutorials/registration_api.php#registration-api has an overview of the process, including an alarmingly large number of options for how to do each step (although apparently SIFT and NARF are the usual feature detectors?), and http://pointclouds.org/documentation/tutorials/interactive_icp.php#interactive-icp has a demo thingy (which uses Blender, but only to generate the input as a .ply file), using the ICP algorithm that is also used in http://pointclouds.org/documentation/tutorials/iterative_closest_point.php#iterative-closest-point. It looks pretty easy to use but I don't know how well it works for incomplete point clouds with errors.

But I don't really want a point cloud. I want a triangle mesh so I can render it efficiently and then simulate projecting images onto the sculpture.

NARF, the "normal aligned radial feature", is https://www.willowgarage.com/sites/default/files/icra2011_3dfeatures.pdf.

Topics

- Graphics (p. 3483) (91 notes)
- Mathematical optimization (p. 3611) (29 notes)
- 3-D modeling (p. 3300) (9 notes)
- Kinect

The etymology of "tradeoff"

Kragen Javier Sitaker, 2016-08-11 (5 minutes)

The word "tradeoff" means giving up some desirable attribute, like fuel efficiency, in order to gain another one, like crash safety. It comes from the phrasal verb "trade off", which means the same thing. But is this sense of "tradeoff" metaphorical or literal?

This is not as simple a question as it may seem. "Trade", taken literally, means to give another person something and receive something in return: commerce or barter. That's not what's happening in a "tradeoff", so it would seem to be metaphorical.

But you can't use "tradeoff" to describe literal commerce or barter. You can't say this:

*I met Billy at the park for a tradeoff of my bicycle for his clothes iron.

"Tradeoff" just doesn't have that sense.

Furthermore, it turns out that it isn't the literal sense of "trade", either. Or at least it wasn't. Douglas Harper's wonderful Etymonline tells me that what we now think of as the literal sense of "trade" was a sort of metaphorical or extended sense, at which point the literal meaning of "trade" was to occupy your time in something, such as milling flour. It's not quite clear when it acquired this new metaphorical sense of bartering; it was definitely established as a verb by the 18th century CE, but as a noun it is attested as early as the 16th century CE.

But that wasn't the original literal meaning of "trade", either. To call milling flour a "trade" was a metaphorical extension of the literal 16th-century-CE meaning of "trade", which was "a path", or as a verb, "to walk a path," like "tread".

But that wasn't the original literal meaning of "trade", either, because when merchants from the Hanseatic Federation of Free Cities introduced "trade" into English, probably in the 14th century CE, it referred to the course *sailed by a ship*. Using it to describe the path a person walked through a forest on was a metaphorical extension, likening the person to a ship. And that's why the trade winds are called "trade winds".

(The Proto-Germanic root *tred-* from which "tread" and "trade" comes, however, already meant "walk," as "tread" does today.)

So the process by which words acquiring meanings is, many times, an evolution from poetic fancy to well-known metaphor to tired cliché to accepted meaning. "Trade" has gone through at least four literal meanings in this chain since its introduction into English: "sailing course", "pathway" or "walk", "occupation", and finally "commerce".

So what's the literal meaning of "tradeoff"? It didn't exist as a word until 1959, but the phrasal verb "trade off" occurs rarely in the 19th century, with the meaning "trade away in commerce":

If we turn to Ireland, also the land of free trade, we see an almost total inability to **trade off** labor in exchange for either food or clothing. Canada has free trade, yet she is unable to **trade off** labor for food, and Canadians are forced to get employment within the Union. Next, we see the farmer of Canada seeking to send his food to be exchanged in the markets of the Union for that labor which could not be employed at home.

The American Whig Review, August 1850, "What Constitutes Real Freedom of Trade ?", Vol. VI, No. XXXII, p.130 (144th page of 696 in the DjVu file, 18th page of No. XXXII)

The owner of the palm-nuts must go to the caravan trader and trade off the palm-nuts for beads, brass rods, or powder.

(Apparently an 1886 United States Consular Report, probably about the Independent State of the Congo, but I'm not succeeding in finding it.)

These are the only two uses of the phrasal verb "trade off" that I've been able to find from the 19th century, although there are many purely coincidental occurrences of that 2-gram, in constructions like "the slave trade off the coast of Cuba".

However, it doesn't seem that the noun "tradeoff" ever referred to engaging in commerce. The earliest uses I've found are from the mid-1960s, after 1963, and they all seem to use "tradeoff" or "trade-off" in the sense of a design tradeoff.

Topics

- History (p. 3500) (71 notes)
- Etymology (p. 3447) (3 notes)
- English

Making a logic gate of a single MOSFET

Kragen Javier Sitaker, 2016-06-28 (5 minutes)

If we interpret "connected to power supply" as a 1 bit and "unconnected to power supply" as a 0 bit, a single SPDT electromechanical relay can compute any of the following logic functions:

• Buffering and NOT, with the input signal and ground connected across the winding and the power supply connected to the armature. Of course this use of relays is what they were invented for and why they're called "relays".
• AND and AND-NOT (abjunction), with a second input signal instead of the power supply connected to the armature.

This is in some sense very device-efficient, computing as it does two separate logic operations per relay. With DPDT relays, you can compute even more. This signaling scheme also permits arbitrary use of wired-OR, which means it's eminently bus-compatible.

Relays, being electromagnetic rather than electrostatic devices, are not troubled by floating inputs. If we use a different signaling scheme, in which "connected to power supply" is 1 and "connected to ground" is 0, then the situation changes somewhat.

• Buffering connects the input signal and ground across the winding, the power supply to the normally open contact, ground to the normally closed contact, and the armature to the output.
• NOT is the same, but with the connections to the contacts reversed, or alternatively with the input signal and the power supply connected across the winding. Making both changes converts the relay back into a buffer.
• AND connects input signal A and ground across the winding, input signal B to the normally open contact, and ground to the normally closed contact.
• AND-NOT is the same, but with the contacts reversed, or the connections to the winding reversed.
• XOR connects the two input signals to the two sides of the winding, power and ground to the normally closed and normally open contacts respectively, and the output to the armature.
• Reversing the power and ground on XOR gives you XNOR.
• OR can no longer be done with a wired-or; that's a short. But you can connect input signal A and ground across the winding, power to the normally-open contact, and input signal B to the normally closed contact.
• I don't see a way to build NAND or NOR as single relays in this system; I think you need two separate relays to compute one of them.

You still want to use the other signaling scheme for buses, so you need a sort of level-shifter relay.

CMOS logic

Normal CMOS logic uses four MOSFETs per two-input NAND or NOR gate. Although this is very simple compared to a TTL gate, you could desire something more parsimonious if you're going to build stuff out of discrete MOSFETs. MOSFETs share with relays the property that when they are switched "on", they have a low-resistance, electrically isolated, bidirectional path between the source and drain electrodes. Wouldn't it be nice if you could use a single MOSFET as a multiple-input logic gate, rather than needing several of them?

I haven't been able to figure out a way to do this. There are a few different obstacles.

• There's nothing equivalent to the first signaling scheme, where one of the bits is represented by letting the input float. When a MOSFET input is left to float, it capacitively retains whatever value it had before, or possibly whatever capacitive charge is induced on it by electrical fields in the vicinity. You can of course tie down all your MOSFET inputs with pullup or pulldown resistors, but then you have at minimum two devices per gate, not one.
• MOSFET *outputs* are either connected to the input or left to float. That means that if you want to connect the output to one of two different things (such as ground or Vdd, or ground and an input signal) you need two MOSFETs, not one.
• MOSFET signaling voltages are somewhat troublesome; enhancement-mode n-channel MOSFETs are turned off when the gate voltage is the same as the source, and the "source" is really whichever of the source and drain is more negative. If you bring the gate voltage up, it starts to allow current to flow. Depletion-mode MOSFETs at first seem more promising, since you have conduction until you apply a gate voltage to cut it off, but now your gate voltage has to be *negative* compared to the source.

I don't know if subthreshold behavior might have an answer; otherwise I suspect that single-MOSFET gates are not going to work.

A MOSFET version of RTL, however, works quite easily, with one MOSFET and up to a few resistors per gate.

Topics

• Electronics (p. 3430) (138 notes)

Arduino radio

Kragen Javier Sitaker, 2016-07-30 (4 minutes)

The Arduino's PWM output can output at up to 31.25kHz with 256 levels, because the counter increments at 8MHz. You can get a 4MHz square wave out of that counter, but maybe more interestingly, that square wave has harmonics at 12MHz, 20MHz, 28MHz, and so on. You should be able to amplify the original signal with a transistor or two and filter out the higher harmonics as a way to generate an RF signal. Maybe even more interesting, if you change the cycle length and duty cycle, you can generate different harmonics.

Now, if you want to generate an FM signal, it's going to be a little bit tricky; you need a harmonic in the range of 87.5 to 108 MHz, which is not impossible, but the tricky part is that the peak deviation is 75kHz, which is a bit under 0.1% of the frequency. So you need some way to somewhat reliably vary the frequency you're generating by quantities less than 0.003%. Maybe you can do this using the AVR's external RC oscillator on the Xtal1 pin with a varactor to adjust the clock frequency of the whole chip a bit.

However, an AM signal should be a lot easier. This is in the 540 kHz to 1610 kHz range, and the Arduino is capable of generating these signals more or less directly; 1600 kHz, 1333 kHz, 1142 kHz, 1000 kHz, 888 kHz, 800 kHz, 727 kHz, 666 kHz, 615 kHz, and 571 kHz are all available subharmonics of the 8MHz timebase, and if you want to use the third harmonic of a frequency that's three times lower, you could very easily get three times as many stations; but that isn't necessary and probably creates more filtering headaches than it's worth.

In the middle of this range, at 800 kHz, you have 10 bits available. If you stick to pure PWM modulation, you then have 10 power levels available for each oscillation, but that's measuring from DC; you really only have 5. But if you're transmitting a max-5kHz audio signal, which is all AM radio receivers can really handle, your Nyquist frequency is 10ksps, so you have 80 oscillations available per "sample" to dither with. This means you really have more like 400 amplitudes to play with — almost 9 bits, 54 dB SNR — and more at lower frequencies. So you should be able to transmit a perfectly respectable-sounding AM signal.

If you simply adjust the PWM threshold between, say, 5 and 10, with a repetition time of 10 cycles, to adjust the amplitude of the 800 kHz signal, you will introduce a "DC" bias as well, which will vary with the amplitude. If this is undesired, you might want to alternate between >50% and <50% duty cycle at some frequency, but doing this without going out of phase may be tricky.

The dithering will produce subharmonic distortion, but if the dither changes levels relatively infrequently, most of the subharmonic can be down in the near-audio range. For example, if you adjust the PWM limit by 1 at irregular intervals averaging 40000 times per second, I think nearly all of the dithering noise will be in the 20kHz to 60kHz range. And you'll only need to service a timing interrupt once every 200 cycles, making it feasible to do other things at the

same time.

A small magnetic loop antenna, maybe one stolen from an AM radio receiver, is probably the best way to do the coupling to the air.

Topics

Clanking replicators

Kragen Javier Sitaker, 2016-11-30 (3 minutes)

So I've come to the conclusion that the most significant thing to focus on right now is getting clanking replicators up and running: programmable machines based on bulk-material-processing technology (i.e. the way all machines except for a few experimental STM and AFM setups in research labs operate) that can reproduce themselves much faster than the economic growth rate, say in hours to weeks rather than decades.

I think this is possible now, there's a sort of "arms race" underway to get it to happen, and whoever succeeds will have a massive economic advantage, comparable to but larger than the discontinuity in the shift from hunter-gatherer and pastoral-nomad lifestyles to agriculture.

Freitas is the one who's written the most about this in the past. Sipper also has a page on the issue.

I've just downloaded Freitas and Merkle's 2004 book KSRM, which seems to be the latest survey of the space; it probably inspired Adrian Bowyer to start RepRap. Freitas in particular seems focused on molecular nanotechnology now, but it seems likely that MNT is somedistance further down the road.

The race is unremarked and can be carried out at small scale and without exotic materials or, probably, much special resources. Once a clanking replicator is created, exponential growth should be rapid.

Many words are available from fiction and more careful speculation for such a project: Autofacs, Second Variety, Screamers, Berserkers, Auxons, and Replicators. Auxon is a positive term (from Lackner and Wendt's 1995 proposal), and Autofac is sort of neutral; the others are all nightmares.

I have a lot of reading to do now.

Mechanical vs. electronic computation

I think mechanical control systems might be adequate, and they won't require the exotic high-purity materials that semiconductor devices do. Freitas I guess didn't think that was going to be a problem in 1980.

However, this requires a mechanical system capable of universal computation. Reif wrote a survey chapter in 2008 on mechanical computation which seems to suggest that nobody has built a mechanical universal computer yet.

The Curta I calculator had only 571 parts, while Vaucanson's swan (according to Freitas) had over 1000. I think Calculus Vaporis could probably be implemented with a similar parts count using lookup tables for combinational logic.

Topics

- Economics (p. 3424) (33 notes)
- Self-replication (p. 3703) (24 notes)

Life octaves

Kragen Javier Sitaker, 2018-10-28 (2 minutes)

Biological humans can most often hear about 10 octaves, from 20 Hz to somewhere in the 10–20 kHz range, although slower pressure oscillations can affect them. Analogously, though they seem to be conscious from birth, they usually cannot remember much that happened before the age of 2 years or so. We could say, then, that they can perceive 6 octaves of their lifetime.

The first octave is from 2 years to 4 years. In the first octave, they most often learn to walk, talk, and theorize about others' minds.

The second octave is from 4 years to 8 years.

The third octave is from 8 years to 16 years.

The fourth octave is from 16 years to 32 years. Many humans cease to learn when they enter this octave, instead busying themselves with trivialities. During this octave they pass from being enslaved by their instincts to being enslaved by their delusions and their habits. Their brains and the rest of their bodies continue to grow, and their peak physical and mental capacities are usually reached during this time.

The fifth octave is from 32 years to 64 years. This is the octave of greatest exploitation, in which humans most often exploit their previously-gained capabilities, which remain more or less consistent during this time. A few continue to develop during this time. Nearly all survive to the end of it.

The sixth octave is from 64 years to 128 years, although quite possibly no biological human has yet lived beyond 122 years. This corresponds roughly to the human idea of "old age"; their physical and mental capacities decline and their risk of death rises superexponentially. During this octave, many humans regret having wasted their previous opportunities, and many blame others for this and become bitter.

Topics

- Psychology (p. 3669) (18 notes)
- Humans

Sulfuric acid dehydration printing

Kragen Javier Sitaker, 2019-12-18 (updated 2019-12-19) (3 minutes)

The CandyFab did 3-D printing in sugar by blowing hot air onto the sugar to melt it, a process that requires delicate temperature control to avoid caramelizing the sugar all the way to carbon foam.

But carbon foam is in some ways a more useful material than sugar. It has a much higher strength-to-weight ratio, it's vastly more heat-resistant, it's less dense, and often it can be prepared as a conductor or an insulator depending on processing temperature.

In addition to hot air, you can also convert sugar to carbon foam by dripping or squirting concentrated sulfuric acid on it, every high-school chemistry teacher's favorite scary chemical reaction. In a powder-bed 3-D printing process like that used by the CandyFab, this permits you to selectively deposit carbon foam in a sugar powder bed. This could be useful for a couple of different reasons: first, you might have fillers that are sensitive to the heat needed to melt sugar, but not to sulfuric acid, perhaps styrofoam beads or something similar; and, second, you might be able to inject the sulfuric acid more precisely or more quickly than you can inject the heat. In Needle binder injection printing (p. 1492) I outlined a variant of this process that would work well with sulfuric acid as the "binder" being injected deep within the powder bed.

A third possibility is injecting a susceptor, then "baking" the whole powder bed with microwaves or with a dielectric heater; for example, vegetable oil should be a sufficient susceptor to dehydrate sugar in a domestic microwave oven, but other candidates include silicon carbide, graphite, and magnetite. These should work with other kinds of powder-bed processes that require post-heating as well, like those described in 3-D printing by flux deposition (p. 466), enabling more rapid heating of large powder-bed-embedded objects than can be achieved by heat conduction alone.

Possible fillers for this process are highly varied, and they can be selectively deposited in the powder bed, as fluxing agents are in the process described in 3-D printing by flux deposition (p. 466). The simplest is powdered, sieved coke, which will simply produce a denser carbon foam and costs US$0.70/kg, according to Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196). The cheapest is silica sand for construction, US$0.012/kg; carbon foam ought to stick well enough to that. An alternative to sugar that similarly foams up by dehydration, and sticks fabulously to silica, is dried sodium silicate (waterglass), US$1.10/kg. Copper or brass (US$4/kg) could form conductive traces; steel wool or glass fiber (US$6/kg) could provide tensile strength.

Sugar, being water-soluble, can also be used as a binder simply by squirting a bit of water onto it; but it will virtually never dry out by itself at room temperature --- you'd have to bake it.

(A quick stovetop experiment shows that granulated table sugar, when heated to dehydration, is able to bind construction sand together at about 25% sugar, but not at about 10% sugar. Presumably this depends not only on the quantities of sugar and sand but also their grain size distributions. Even at 25% sugar, the mass is quite crumbly;

it disintegrates with a touch. 50% sugar is quite a bit more solid.)

Topics

Prototyping stuff

Kragen Javier Sitaker, 2016-08-11 (1 minute)

I have a lot of stuff I want to make prototypes of that I haven't gotten around to yet.

• A compiler for a comfortable programming language. Emitting relatively naïve native code, like within a factor of 5 of native code, should be pretty easy. Ur-Scheme did it. But something with structs, iterators (maybe downward funarg blocks?) and maybe some escape analysis or something would be better. And for OO stuff, Piumarta's tiny MOP seems like it should be able to deliver dynamic dispatch performance within a small factor of a regular function call.
• A deterministic virtual machine. This is closely tied to the previous one.
• Some fucking laser-cut furniture.
• A video game that's fun to play. Maybe tie in with information archival somehow.
• Executable models of electromagnetic and mechanical engineering phenomena.
• Some kind of live-within live programming environment, like Emacs, Smalltalk, and actually sort of Unix.
• Something that does something interesting with machine vision.
• An optimization system that's applicable to laser-cut designs, using executable models of mechanical engineering phenomena.
• A solar heating and dehumidifying system.
• A system for drawing process flow diagrams.
• A ZUI. Another one, one with text and server-side persistence.
• A usable time-series database, unless someone else has written one by now.

Topics

An RPN CPU instruction set doubling as user interface

Kragen Javier Sitaker, 2017-07-19 (updated 2019-07-10) (21 minutes)

(See Making the CPU instruction set a usable interactive user interface (p. 59) for related ideas.)

The Olivetti Programma 101 in 1964 and its VWXZ keys

I was reading about the Olivetti Programma 101 desktop computer today. It cost US$3200 when it came out in 1964; they sold forty thousand of them. It could be used as a printing calculator (LED displays were still in the future) and you could load a 120-instruction program from a magnetic card. Its magnetostrictive delay-line memory was somewhat anemic at some 240 bytes, circulating with a bit over 2 milliseconds of cycle time. It's one of the reasonable contenders for the title of "first personal computer".

The thing that struck me as interesting about this machine was that four of the buttons (labeled V, W, X, and Z) were "start program" buttons: they jumped to four of the thirty-two available labels. So you could write an interactive program that enhanced the calculator's capabilities with four new functions invoked with these keys. When the calculator finished feeding the magnetic card through the reader, it would come to rest atop these keys, so it could bear human-readable labels for them, explaining what the newly-loaded program had programmed them to do.

The keyboard *is* the instruction set

RPN calculators, of which I assume the Programma was one, enjoy a pleasantly simple way of programming them: the sequence of computational steps to execute is the same as the sequence of keystrokes to do the same computation interactively, and parameter passing and result return is implicit. The difference is just that in "program" mode, the program steps are added to a program instead of being executed. (The user experience might arguably be better if you did that *as well as* executing them, rather than *instead of* executing them, but for it to be an actual improvement over the traditional HP experience, that probably requires enough screen real estate to simultaneously display the program steps being recorded and the example values.)

This means, in effect, that your CPU instruction set is simultaneously your user interface, which suggests that you might have instructions for such odd keystrokes as "multiply the current top of stack by 10 and add 5". These are obviously far easier to implement as code than as transistors, and as a result machines like the HP 9100A were heavily microcoded, to the point that they had to invent a new kind of ROM to store the calculator's microcode.

The VWXZ approach, however, suggests an alternative to microcoding: implement most or all keys as procedure calls rather than a CPU instruction. Better yet, implement them as CPU

instructions that call particular subroutines. Then, you can avoid microcode and the need to have two levels of programmability in your computer. If you can spare the RAM space for a sort of "interrupt vector" for these keys, then you can make those keys and those instructions reprogrammable.

(If you can do this and also write some kind of idle-time handler that runs when the computer is waiting for an instruction from the keyboard and doesn't get one, you can incrementally extend the computer's instruction set into an arbitrary application.)

How teensy can you make the machine?

We probably can't make do with 32 instructions and keys on the keyboard. A modern "four-function" calculator has the digits, ".", "=", +, -, ×, ÷, MR/C, M+, M-, ON/C, CE, +/-, √, and %, a total of 24 keys; you probably need at least those instructions or their stack equivalents for a usable calculator, plus more than 8 instructions that aren't one of those keys. So you probably need at least 64 keys and instructions, which is what the HP 9100A had.

How much space do you need for a program address? The HP 9100A needed a 32-kibibit ROM for its microcode, which is also how much RAM the late-1970s personal computers needed for a BASIC interpreter; the 9100A also could hold 14 6-bit instructions per register, of which it had 23 implemented as core memory, for a total of 23·14·6 = 1932 more bits. This was sufficiently limiting that they started shipping an HP 9100B within the year with double the RAM, 3864 bits http://www.hp9825.com/html/the_9100_part_2.html. The earlier Olivetti had 240 bytes of delay-line memory, which I suspect were 6-bit bytes; this gives a similar number of 1440 bits.

Let's figure that stack-machine code will probably be a bit more compact than the 9100A's microcode, especially with magic procedure-call instructions, but not all that much, maybe a factor of 2 or 3. Then you need something like ten kibibits of program memory, a bit over 1700 instructions. You could impose, say, a four-instruction alignment requirement on the vectors for the keys, which would cut the vector size down to 9 bits. So vectors for all 64 possible instructions would require only 576 bits, 6% of total memory. Of course you need hard-wired functions for some instructions.

You can probably make do with half of the program memory space being ROM.

So how is this shaping up? We have an interactively usable, fully programmable computer whose memory space consists of 512 24-bit words, of which half are ROM and half are RAM, for a total of 6144 bits of RAM, 59% more than the HP 9100B. 16 of the 256 words of RAM contain 32 instruction addresses (plus six leftover bits) that define the meanings of 32 of the 64 possible six-bit instructions (and keys); the other 32 instructions are hard-wired, perhaps taken from the F18a core. Those 240 words of RAM can contain 960 instructions, which can invoke routines in the other 1024 instructions stored in ROM. These are stack-machine instructions, so you have to spend about half of them on stack manipulation to make up for not having operand fields, so this is roughly comparable to 1000 machine instructions for the 386 or SPARC (500 in RAM, 500 in ROM): barely enough for a simple compiler or assembler, plenty for a video game, and probably not enough for a working TCP/IP stack.

(It's a great deal more than the 1024 bits of RAM in the Atari 2600 or the 1152 bits of RAM in an F18a core, but less than the 16384 bits of RAM in an ATMega328 Arduino, and dramatically less than its 262144 bits of Flash.)

But these 6144 bits of RAM, if implemented as electronic static RAM, will need 12288 transistors — very likely more than the entire processor, maybe even if it's bit-parallel and includes a multiplier. If you could get that memory complexity down a bit, you would have a computer that could scale to much more complex tasks. But this is probably enough to bootstrap with.

If you're building the CPU out of mechanical logic, six 16-position sliders give you a 24-bit word of RAM. All 256 words, then, can be stored in 1536 such sliders. This is more complicated than a Curta calculator or a pocket watch, but not in the same world of difficulty as many mechanical machines that already physically exist. In fact, it's probably a lot less demanding than the Jaquet-Droz automata.

Comparison to the GreenArrays F18A

The F18A has a ten-item parameter stack and IIRC a nine-item control stack; more deeply nested code will not be able to return. In its case, since they're 18 bits wide, they amount to 342 bits. On this hypothetical machine, your parameter stack would be 24 bits, but the return stack could be narrower, as little as 11 bits if you didn't want to use it for loop counters; so you could still fit ten parameters and nine return addresses into 350 bits, which is 5.7% of the size of the RAM and therefore probably a good investment.

(The F18A also has some named memory pointer registers, one of which is also used for multiplication, and a few other miscellaneous features.)

Arithmetic overflow and usability

My experience with integer math in computer programs is that I have to think about overflow incessantly with 8-bit variables, frequently with 16-bit variables, and almost never with 32-bit variables. I have no experience programming with 24-bit variables, but it seems like they would probably be pretty easy. Maybe I should be thinking about floating point.

So the experience of bootstrapping this machine is probably that the bare CPU gives you a 24-bit integer hexadecimal or octal arithmetic calculator with addition, subtraction, and maybe multiplication, and then you can write programs for division, square roots, logarithms, transcendental functions, and whatnot. Or you can instead write programs that don't care about transcendental functions and instead do other things, like assemblers and BASIC interpreters, so you can bootstrap to higher levels of abstraction.

Modeless machine-code programming by example

You could maybe eliminate the distinction between compiling and interpreting modes by always recording the user's keystrokes into some memory buffer or other, thus always preserving the option of executing them later. (See A two-operand calculator supporting programming by demonstration (p. 2387) for more elaboration on

this theme, in a non-stack-machine context.)

Display refresh, task segments, and multithreading

If you're executing machine code interactively to do calculations, you probably want to see the values you're operating on. If the CPU has registers (for example, a top-of-operand-stack and next-on-operand-stack register), you probably want to see those registers rather than, say, some memory location. But more advanced applications might want to display something custom, which might not be numbers. One way to do this would be for hardware to copy these registers onto the display every, say, 15 or 20 milliseconds; if the system that does this raises an interrupt a little earlier, then a custom interrupt handler could arrange to set those registers to the desired display value for a few dozen microseconds. Another way would be to update the display explicitly rather than implicitly, possibly also from a timer interrupt handler.

This ties in with how the machine's control flow weaves together keyboard response, background-task service, and interruption of accidental infinite loops. A standard single-threaded computer runs a top-level infinite event loop with its PC either at a halt instruction or running some background task most of the time; when you press a key, an interrupt is raised, the background task is suspended (or the halt is raised), and the PC moves into the keyboard interrupt handler; this buffers a keystroke and perhaps sets a flag or two, and upon return, the main event loop takes note of the keystroke (sooner or later) and acts upon it, perhaps invoking other subroutines.

Here we're proposing that the PC normally be pointing *to the keyboard*, such that when the CPU attempts to fetch an instruction, the fetch blocks until a key is pressed; possibly a timer interrupt might interrupt the blocked fetch from time to time, running some background task, before resuming the blocked fetch; when the keystroke executes, it may call some other subroutine, pushing the address of the keyboard onto the stack for a while until the subroutine finishes and the CPU awaits the next keystroke.

One consequence of this is that you need some way to keep the PC from incrementing out of the keyboard space. This could be done with saturating arithmetic (placing the keyboard at the end of the address space), an equivalent special case in the PC increment logic at some other arbitrary address, or in a non-special-case way by limiting the width of the PC increment logic. For example, you could simply omit the carry out of the 7th bit of the PC (and everything beyond), such that every 128 memory words formed a separate "task segment" — if control falls off the end of one, it just wraps back to its beginning. Then you memory-map the keyboard to an entire task segment. (I think I got this solution from the GreenArrays chips, but of course an 8086 will do much the same thing — IP is a 16-bit register and doesn't overflow into CS; if you execute off the end of a 64K code segment, it will wrap back to the beginning.)

However, in this model, quite aside from how it may or may not handle PC incrementation, two problems are not entirely resolved. First, how do you recover control if you accidentally invoke an infinite loop which never returns into the keyboard address space?

Second, how can a best-effort background task (as opposed to an isochronous task like updating the display or incrementing an RTC) yield its time slice to interactive computation? After all, from the point of view of an interrupt handler, the difference between being invoked when the machine is idle and while it's running some computation is just a matter of whether it's returning to the keyboard's address or somewhere else; it would be ugly to have it introspect that in order to decide whether to return before doing its quantum of work.

I'm not yet convinced there's a clean solution to those problems within the keyboard-is-instruction-set, no-monitor-program-needed constraint.

The 8080 RST instruction and keyboard bucky bits

Each instruction on the Intel 8080 was one byte, not counting possible operand bytes following the opcode byte in the instruction stream, so there were only 256 possible instructions. 8 of these precious slots were given over to an "RST" instruction, glossed "restart", which included a 3-bit operand field "nnn"; in octal, it had the opcode 03N3. These called a subroutine at address 000 0N0 (again, in octal), and the machine's hardware interrupt, as I understand it, invoked the eighth of these eight subroutines in the same way as if an RST 7 instruction had been issued.

Since they called a subroutine in the usual way, pushing the PC on the stack, the code thus invoked could RET to the program that had invoked the RST instruction (or that happened to be running when the hardware interrupt arose), which would then continue its execution as if nothing untoward had happened. In essence, the eight RST instructions provided eight programmable opcodes, vectored to subroutines in RAM or ROM.

The question arises: if some of your opcodes are such vectorable subroutine calls, and you're relying on this for your user interface, what is the proper number of them to have? The Olivetti Programma 101's four programmable keys is probably too few, as is the 8080's eight RST opcode bytes. You usually want to have more than eight actions available to the user at any given time when they are using an application, especially if they are going to be composing text. And it's reasonable to think that high-level application code might consist largely of calls to existing subroutines, although some amount of immediate data is surely needed; being able to vector a significant chunk of the instruction set to such subroutines might be a really cool way to extend the instruction set in an application-specific way, taking language-oriented programming down to the machine-code level.

One constraint here is the physical keyboard size. The keyboard I'm typing this on has 88 keys, like a piano. (It's a "Genius" "Luxemate 100" USB keyboard chosen because it was cheap and small enough to easily fit into my backpack, despite having kind of a shitty feel and a non-coiling cord.) A minimal typewriter-style keyboard probably has around 50 keys, as does the "Cifra SC-9100" scientific calculator I profiled in Usability of scientific calculators (p. 2379). A keyboard with 256 or 512 keys would be unwieldy. Even

the 101-key keyboards that are common on PCs nowadays are unwieldy; you can't reach most of their keys without moving your hands. The Android 4 on-screen keyboard on the phone I have here has only 34 keys, and it requires constant mode-switching and autocorrect to approximate usability. Termux, a Unix CLI environment for Android, adds 17 more keys, bringing the total to 51, and does reach a more reasonable level of usability.

StoneKnifeForth, which is an i386 compiler that compiles itself into a Linux ELF executable in two pages of source code (or 7 pages counting comments), consists of 50 subroutines and variables. More elaborate software modules need more, but probably it's only rarely useful to have many times more subroutines accessible than that.

So, an interesting design alternative: have an 8-bit instruction set of which 64 bytes — perhaps the ones corresponding to ASCII digits and lowercase — are directly generated by the keyboard and vectored like the 8080's RST instructions are vectored:

```
  +  0 1 2 3 4 5 6 7 8 9 a b c d e f
0x20   ! " # $ % & ' ( ) * + , - . /
0x30 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
0x60 ` a b c d e f g h i j k l m n o
0x70 p q r s t u v w x y z { | } ~ DEL
```

This omits uppercase letters, @, [, \,], ^, _, and the control characters other than DEL; none of these are essential for arithmetic, and this was probably a deliberate design decision on the part of the ASCII committee: a single contiguous 32-character chunk of the code space is adequate for FORTRAN, except for the alphabet.

In addition to these 64 keys, corresponding directly to bytes that invoke subroutine calls in the CPU, you can have two "bucky keys" to generate the rest of the possible 8-bit bytes. One of them ("Meta") should obviously set the high bit, while the other one probably needs to act like Shift for the lowercase letters, which is to say that it maps 0x61 to 0x41, 0x70 to 0x50, and so on. The simplest thing for this "Shift" to do would be to clear the 0x20 bit, which would map the digits and most punctuation to control characters. In particular, backspace would be shift-"(", tab would be shift-")", CR would be shift-"-", and LF would be shift-"*". This may be too outré to use in practice, although at least DEL is on a key of its own.

Traditional keyboards, as represented by libvte and xterm anyway, ignore Ctrl for pretty much everything in the 0x20 and 0x30 rows, except that Ctrl-/ yields ^_, US.

The idea is that, even if the majority of the keys on the keyboard correspond to vectored-subroutine instructions, some key combinations would instead correspond to hardwired instructions that would allow you to regain control and rebootstrap the system even if the subroutines attached to your normal keys got horked.

(Consider instead the approach of providing "Shift" and "Ctrl" keys that individually do what you would expect: shift-a is A, ctrl-j is SOH, shift-j is J, ctrl-j is LF. So shift must map 0x60 to 0x40 and 0x70 to 0x50, while ctrl maps 0x60 to 0x00 and 0x70 to 0x10. But technically that leaves open the shift mapping and the ctrl mapping for 0x20 and 0x30, as well as the ctrl-shift mapping for everything. For example, you could have shift map 0x20 and 0x30 to 0xa0 and

0xb0, ctrl-shift map 0x20 and 0x30 to 0x80 and 0x90 and map 0x60 and 0x70 to 0xc0 and 0xd0, and ctrl map 0x20 and 0x30 to 0xe0 and 0xf0. There's probably a viable option in this space that can be achieved with a few logic gates, but for now I will leave the option open. Furthermore, for now I won't consider the possibility of a second bucky bit for a Meta key, or a third bucky bit for key release events, although detecting long key presses is clearly necessary for many important applications, such as Tetris. These suggest nine- or ten-bit instruction "bytes", which would be mostly vectored subroutines.)

Topics

Short words

Kragen Javier Sitaker, 2019-12-10 (updated 2019-12-11) (4 minutes)

I like programming-language tokens to be short; if alphabetical, I like them to be comprehensible. This can lead code to look like Urbit, but no matter. In particular I think Perl's "my" is significantly better than "var", "val", or "let", and JS's "const" is a lose. Similarly, Darius Bacon's language Cant uses "#yes" and "#no" rather than "true" or "false".

The most frequent 128 three-letter words in English, according to the British National Corpus, are:

the and was for you are not but had his she her all has one can who him its two out did now may new any see how get way our got own too say erm day yes man use put old why off end men set yet six war car saw let far law big act job age run try pay ten mrs ago ask few air god sir lot cos bed tax top art cut bad per boy bit son sea red nor gon low buy sat met cup oil led lay eye arm win hot sun ran box sit tea won sex add aid dog key mum bar eat mhm gas hit dad dry fit inc aim due die leg ian bus aye ltd tom

Most of these are at least real words, though "Inc.", "Ltd.", "Mrs.", "cos", "gon", "mhm", "Ian", and "erm" did make it in there, some of which may not be real words, depending on your definition. "tom" (presumably mostly "Tom") occurs 5063 times.

There are 39 two-letter words that occur more frequently than that:

of to in it is on he be by at as or we an do if so no up my me go er us oh mr mm ca am uk wo na de st dr ah ii tv ec

These are mostly words, but it starts getting pretty dubious at the end there.

There are 320 four-letter words that are this common, but 320 is too large a vocabulary. The most common 128 are:

that with have this they from were been will what said more them some into then time like only your just also know well very than most over back much many yeah work down make good such year must last take even here come both does made same when want life need used home each part went look came four give mean next case find long five says took away seen fact less done area help hand best head side days john left week form face room tell able high told half eyes keep once road open full knew feel ever name mind door body book main show upon gave real view line city felt kind idea read sort care else free thus past love play land gone

The most common 128 words in English, other than those mentioned above, are:

a i which there their would about could other these people first should think between years being those because three through still after right going before government might under however world another while again against never something thought house number different really children within always without local system great during small place although little things social group second quite party every company women later given important point information national often school money public night further found since better around british having thing london taken perhaps state family water though already possible nothing where business large young whether enough development country almost council power until himself political become times service members change problem doing court towards major anything others police either problems interest probably asked available labour today education

A potential disadvantage to using real words in your programming language is that people are more likely to try to use them as identifiers, and depending on the language design, that may or may

not be possible.

Desbarrerarme: a UI for speaking to people

Kragen Javier Sitaker, 2015-09-03 (5 minutes)

Facebook chat is dangerously addictive in part because it is so easy to use, and constantly offers you new opportunities for what to do. However, aside from the political problems with using it, it's somewhat repulsive in that its HTML UI often hangs for long periods of time and makes your machine slow.

But it has a number of good points, which I want to incorporate into a UI of my own for talking on chat:

• Saying "OK" in your current conversation is three keystrokes: O, K, and Enter. No need to choose a subject line, no need to invoke a "new message" command.
• Nevertheless, you can send arbitrarily long messages, with formatting. FB has removed the boldface from asterisks it used to support, and it tends to mangle complex formatting, but it's still perfectly usable for multiple paragraphs filled to your screen width.
• You can move among recent conversations with single keystrokes, ↑ and ↓, to be specific.
• It suggests that you interact with people you haven't interacted with in a while, to a limited extent, by using their presence information.
• Regardless of which device you're interacting from, you always see the same message history. Opening a chat shows you your previous conversations with the person.
• It supports links and embedded images.

So my plan is to rig up something similar, but for email. As with Gmail, you'll see the emails in a thread above your text box at the bottom, with repeated crap elided. By default, you'll have only a single thread with a given person, and you'll always be replying to their latest message, if any. If they change the subject line, it will be a new thread. You should be able to answer ten emails from ten different people with one-word answers in ten seconds, even if you don't have an internet connection.

Each thread has a simple read/not-read flag. When you open it, it gets marked as "read". When you receive a new message in it, it gets marked as "not-read".

Furthermore, I want to use this for chat systems other than mail, including everything Pidgin supports.

To allow scaling to a larger number of historical threads, and more important threads, than Facebook, I want to do some amount of search-based navigation — to let you see just the results of a search. XXX

I want to use this to try out a theoretical design for distributed user interfaces I called "rumor-oriented programming". Every time you receive a message, mark a message to be sent, successfully send a message, start writing a draft, or update a draft, that information is recorded as a "rumor" in the "rumorset" of your Desbarrerarme installation. When you synchronize your Desbarrerarme installations, which will happen constantly while you're online, they spread all the

new rumors to each other.

The user interface is rendered from a query on the current rumorset state, a query which is not sensitive to the order in which the rumors arrived at the current node. Therefore, once two Desbarrerarme nodes have finished synchronizing, their user interfaces are guaranteed to render the same information.

However, there is a little bit of information, like which thread you're currently looking at and where your cursor is in the draft, that should not be replicated. This is "control state", and not replicating it is crucial to preserving usability in the face of multiple clients being used by, potentially, multiple people.

To keep things simple, I want to write the user interface view as a pure function of the rumorset state and control state. But, to keep things efficient, I want to cache and reuse the results of previous computations. This is much more important for the rumorset part of the question, which could contain gigabytes of information, than for the user interface, which might be a megabyte or, in the form of drawing operations, a couple of kilobytes.

I also want to avoid complete recomputations of large results. For example, the leftmost column of the screen layout should contain the

Topics

Improving “science” in eSpeak's lexicon

Kragen Javier Sitaker, 2007 to 2009 (updated 2019-06-27)
(15 minutes)

So I've been playing around with speech synthesis software tonight. eSpeak looks a lot nicer than Festival, just in that it's much easier to adjust its speed, correct its pronunciation, and play with variations: whisper, different accents, pitch, word spacing, creaky voice. I got to thinking, what would a logical policy for updating its lexicon look like? I thought the results I came up with were interesting. Maybe some other people will be interested too.

The problem

eSpeak gets “neuroscience” and “pseudoscience” wrong, pronouncing them with a `[[s,i@ns]]` rather than a `[[s'aI@ns]]`. It also gets “omniscience” and “prescience” wrong, or at least pronounces them rather differently than I would:

```
$ ~/pkgs/espeak-1.37-source/src/speak -v en/en-r+f2 -s 250 -x "The
    science of neuroscience is not a scientific or quasiscientific
    pseudoscience.  Conscientiously pursue omniscience and prescience."

 D@2 s'aI@ns Vv n'3:r-@s,i@ns I2z n,0t#@ saI@nt'IfIk _:_:D@ kw,eIzaIsi@nt'IfIk sjo
o'u:d@s,i@ns
 k,0nsI2;'EnS@sli p3sj'u: '0mnIs,i@ns _:_:and pr'i:si@ns
```

I would pronounce the “science” in “omniscience” and “prescience” as `[[S@ns]]` and put the accent on another syllable.

There's a special rule for “scien” beginning a word, and for “conscience”:

```
en_list:conscience      k0nS@ns
en_rules:       _sc) ie (n      aI@
en_rules:?8     _sc) ie (n      aIa2
```

However, Jonathan Duddington has said he wants to keep the eSpeak distribution small, so he “wouldn't want to include too many unusual or specialist words”. (See http://sourceforge.net/forum/forum.php?thread_id=1700280&foruo
om_id=538920 where he talks about why he doesn't want to import the Festival lexicon.) Already, `espeak-data/en_dict` is 80KB, which is half the size of the `speak` binary.

Replacement strategies

There are several possible strategies that a maintainer could adopt in order to improve the coverage of their special-case word files without letting them get large. Suppose that there is a scalar metric of “goodness” that can be applied independently to each special case. Here are three plausible strategies, ordered from least to most stringent.

• C-: They could never remove items from the file, adding new items as long as they were better than the worst item in the file. This will probably cause the average quality of the entries in the file to gradually decline, because many of the most important entries were probably added early on. It will eventually result in a very large file with very low average quality per entry, but very comprehensive coverage.
• C+: They could keep the number of items in the file fixed, adding new items as long as they were better than the worst item in the file. This will cause the program to gradually work better, but each new version will introduce regressions --- words that the previous version pronounced correctly, but the new one does not.
• A: They could never remove items, but add new items as long as they improved the median item quality of the file --- that is, as long as the new item improved the program's performance more than most of the items in the file. This will gradually slow down and eventually stop the addition of new items, because that median quality will gradually increase.

I am going to approximate "quality" with "frequency", on the theory that mispronouncing a rare word is always better than mispronouncing a common one.

Note the analogy to Google's famous hiring policy: only hiring candidates who raised their average ability.

Evaluating word frequencies

Are these "science" words significant enough to include? `en_list` only contains 2869 lines, maybe 2400 of which are words. So maybe only the top 2400 or so exceptions to the normal rules of pronunciation are currently considered for inclusion.

Some time ago, I tabulated the frequencies of words in the British National Corpus and put the results online at http://canonical.org/~kragen/sw/wordlist. It has 109557 lines, ordered from the most common words ("the", "of", and "and", each occurring millions of times) to the least common (with a cutoff of 5 occurrences, because most of the words with fewer were actually misspellings).

I selected 20 lines at random from `en_list` with the following results:

```
kragen@thrifty:~/pkgs/espeak-1.37-source/dictsource$ ~/bin/unsort < en_list | hea
od -20
this            %DIs          $nounf $strend $verbsf
barbeque        bA@b@kju:
con             k0n
?5 thu  TIR       // Thursday
_:      koUl@n
Ukraine        ju:kr'eIn
peculiar        pI2kju:lI3
unread          Vnr'Ed        $only
inference       Inf@r@ns
José           hoUs'eI
unsure          VnS'U@
```

```
survey                  $verb
ë       $accent
epistle         I2pIs@L
Munich          mju:nIk
scenic          si:nIk
synthesise      sInT@saIz
corps           kO@             $only
rajah           rA:dZA:
transports      transpo@t|s     $nounf
```

Where do these special cases appear in the British National Corpus tabulation? Here are some results, edited for readability:

```
kragen@thrifty:~/pkgs/espeak-1.37-source/dictsource$ grep -niE ' (this|barbeque
   |con|thu|ukraine|peculiar|unread|inference|José|unsure|survey|epistle|munich
   |scenic|synthesise|corps|rajah|transports)$' /home/kragen/devel/wordlist
22:463240 this
1178:7999 survey
5102:1441 peculiar
5831:1200 corps

7165:888 ukraine
8977:634 munich
9045:627 unsure
10552:494 inference

11134:455 con

15127:275 scenic
29899:82 epistle
31386:74 transports
34270:62 synthesise

37255:52 unread
73679:11 thu
74154:11 rajah
87737:8 barbeque
```

The 50th-percentile among the sample of 20 (of which two weren't words, and a third wasn't found) seems to be line 11 134 with the word "con". That is, the exceptions in `en_list` are mostly drawn from the most frequently used eleven thousand words in the language. (Maybe words like "barbeque", "rajah", and "unread" should be dropped.)

So under the policies "C+" and "C-", any word that is more common than "barbeque", at position 87737 in the British National Corpus tabulation, (or maybe some word even a bit rarer than that) should be added to the file. (Under policy "C+", some word would be removed to compensate, raising the threshold.) Under the policy "A", the threshold would be "con", at position 11 134.

Unfortunately, José is missing. I think I excluded accented characters when I tabulated the frequencies initially.

Anyway, that gives us a way to compare the "science" words:

```
kragen@thrifty:~/pkgs/espeak-1.37-source/dictsource$ grep -n scien[tc]
    /home/kragen/devel/wordlist
```

```
870:10597 science
1614:5922 scientific
2584:3547 scientists
3865:2088 sciences
3977:2005 scientist
5342:1355 conscience

13365:338 conscientious
16976:227 scientifically
25757:109 consciences
26015:107 conscientiously
27861:93 unscientific
37040:53 omniscient
44349:36 prescient
49031:29 neuroscience
49706:28 prescience
50457:27 scientificity
50587:27 omniscience
53155:24 scientism
62346:17 geoscience
66943:14 scientia
67285:14 neuroscientists
68176:14 conscientiousness
82060:9 geoscientists
84433:8 scientology
84434:8 scienter

86513:8 geosciences
90235:7 neurosciences
93073:7 biosciences
93074:7 bioscience
95039:6 scientifique
95591:6 pseudoscience
103190:5 presciently
103191:5 prescientific
```

Of these, only those more common than “conscience” seem to deserve a place in `en_list`. How does eSpeak do now?

```
$ ~/pkgs/espeak-1.37-source/src/speak -v en/en-r+f2 -s 250 -x "Science is
    scientific and done by scientists, who work in the sciences.  A
    scientist with a conscience may be conscientious.  Those with
    scientifically-minded consciences will conscientiously avoid
    unscientific claims of omniscient beings or prescient prophets."
 s'aI@ns I2z saI@nt'IfIk _:_:and d'Vn baI s'aI@nt#Ists
 _:_:h,u: w'3:k I2nD@2 s'aI@nsI2z
 a2 s'aI@nt#Ist wI2D a2 k'0nS@ns m'eI bi: k,0nsI2;'EnS@s
 DoUz wI2D saI@nt'IfIkli m'aIndI2d k'0nS@nsI2z wIl k,0nsI2;'EnS@sli; a2v'OId
 VnsaI@nt'IfIk kl'eImz Vv '0mnIs,i@nt b'i:;INz _:_:O@ pr'i:si@nt pr'OfIts
```

It pronounces everything correctly until it gets to "omniscient" and "prescient", and maybe its pronunciations for those are correct, but at least they’re not the pronunciations I would use.

Under policy “A”, those words are not common enough to add to `en_list`, because they would lower the average frequency of words in

en_list unless you removed a less common word to compensate.

Under policies "C+" and "C-", not only "omniscient" and "prescient" qualify, but so do "neuroscience", "geoscience", "neuroscientists", and "geoscience", which eSpeak currently mispronounces.

(Including all the exceptions that as rare as "prescient" might quadruple the size of en_list, and perhaps en_dict as a result, if arbitrary spellings were as common among rare words as they are among common words. Think of that as an upper bound. Including all the exceptions as rare as "neuroscientists" might multiply its size by seven. This is the downside of policy "C-", but it does not happen with policy "C+". On the other hand, under policy "C+", even "prescient" might not survive long after being added.)

Recommendation

There is a better solution than adding a bunch of one-word special cases to en_list.

Probably in this case the solution is to change the special case for "conscience" to a special case for "conscien..." and change the "scien..." rule to a "...scien..." rule; that covers all the words except for "omniscien..." and "prescien...". Covering those two takes only two more rules in en_rules, if it's considered worthwhile; but "conscience" is ten times as common as both of those together, "con" three times as common, but "barbeque" 18 times less common.

Alternatives

I think there is a need for a larger en_list and en_rules to be available, even if they aren't part of the standard distribution. eSpeak's current footprint for a single language is about 160KB for the executable and 80KB for the dictionary. But it would be useful in many cases even if its dictionary were 800KB (as perhaps it would be with the Festival lexicon) or 8MB.

And for a better user interface for making changes to the dictionary, and especially en_rules, since currently it's hard to know what words you're changing the pronunciation of when you change en_rules, and you have to master a phonological orthography system to make any contribution at all. And then there's no git-like infrastructure for sharing your changes, and even learning git is a pretty big barrier to contributions.

If, instead, you could twist a knob to jog back to the last mispronounced word, then hold down a button and say its correct pronunciation, the barrier to contributions would be much lower. You would need a reasonable phonological analysis system (like in a speech-to-text system) to turn the spoken word into the string of phonemes. Then, if you could share your accumulated corrections with all other users of the software with the push of a button, the process of coming up with the tens of thousands of special cases would be a lot quicker.

Update from 2019: eSpeak is super awesome now

The above is about eSpeak 1.37 from perhaps 2008. I currently have eSpeak 1.48.03 from 2014 installed, and en_dict is now 116K

instead of 80K. The en/en-r voice used above doesn't exist any more, but the en-us voice is a fairly close equivalent:

```
$ espeak -v en-us+f2 -s 250 -x "The science of neuroscience is not a scientific
     or quasiscientific pseudoscience.  Conscientiously pursue
     omniscience and prescience."

 D@2 s'aI@ns Vv n'U@r@s,aI@ns Iz n,0t#@ saI@nt'IfIk_:_: O@ kw,eIzaIsaI@nt'IfIk s'o
ou:doUs,aI@ns
 k,0nsI2;'EnS@sli p3s'u: 0mn'IsI;@ns_:_: and pr'i:si@ns
```

It now pronounces "neuroscience" and "pseudoscience" correctly. The relevant part of `en_rules` is as follows:

```
    sc) ie (nc     aI@
        ie (ntiC   aI@
   _sc) ie (n      aI@
?8 _sc) ie (n      aIa#
```

I think that means that now the "ie" in any instance of "scienc" will be pronounced as "aI@", regardless of whether it's at the beginning of the word, which is what the "_" in the last two entries means, as explained in docs/dictionary.html in the eSpeak source code.

My other example now renders as follows:

```
$ espeak -v en-us+f2 -s 250 -x "Science is
        scientific and done by scientists, who work in the sciences.  A
        scientist with a conscience may be conscientious.  Those with
        scientifically-minded consciences will conscientiously avoid
        unscientific claims of omniscient beings or prescient prophets."

s'aI@ns Iz saI@nt'IfIk_:_: and d'Vn baI s'aI@ntIsts
 h,u: w'3:k InD@2 s'aI@nsI#z
 a# s'aI@ntIst wID a# k'0nS@ns m'eI bi: k,0nsI2;'EnS@s
 DoUz wID saI@nt'IfIklim'aIndI#d k'0nS@nsI#z wI2l k,0nsI2;'EnS@sli; a#v'OId
```

(line break inserted)

```
VnsaI@nt'IfIk kl'eImz Vv 0mn'IS@nt b'i:;I2Nz_:_: O@ pr'i:si@nt pr'0fI2ts
```

This is different in several details from the above, but overall it doesn't seem to be worse in any way. Also, eSpeak now has an `--ipa` option, which produces the following output instead:

```
s'aɪəns ɪz saɪənt'ɪfɪk ænd d'ʌn baɪ s'aɪəntɪsts
 hˌu: w'ɜ:k ɪnðə s'aɪənsɪz
ɐ s'aɪəntɪst wɪð ɐ k'ɑ:nʃəns m'eɪ bi: kˌɑ:nsɪ'ɛnʃəs
 ðoʊz wɪð saɪənt'ɪfiklim'aɪndɪd k'ɑ:nʃənsɪz wɪl kˌɑ:nsɪ'ɛnʃəsli ɐv'ɔɪd ʌnsaɪənt'ɪfɪk
kl'eɪmz ʌv ɑ:mn'ɪʃənt b'i:ɪŋz ɔ:ɹ pɹ'i:siənt pɹ'ɑ:fɪts
```

To me, this is dramatically more readable, but it is omitting some details that are important to at least eSpeak's pronunciation; for example, the `_:_:` pause above doesn't seem to appear, nor does the distinction between I (stressed) and I2 (unstressed, but not reduced like the undocumented I#). You can use it to translate from eSpeak's internal format to IPA by using `[[]]`:

```
$ espeak -v en-us+f2 -s 250 --ipa "[[h,u: w'3:k InD@2 s'aInsI#z]]"
 h ˌu: w ˈɜ:k ɪnðə sˈaɪnsǂz
```

This makes it easy to compare the old and new pronunciations simultaneously by ear and by reading the IPA transcription, which reveals a few different improvements:

```
$ espeak -v en-us+f2 -s 250 --ipa "[[D@2 s'aI@ns Vv n'3:r-@s,i@ns I2z n,0t#@ saI@
ont'IfIk _:_:
0@ kw,eIzaIsi@nt'IfIk sj'u:d@s,i@ns]].
[[k,0nsI2;'EnS@sli p3sj'u: '0mnIs,i@ns _:_:and pr'i:si@ns]].
The science of neuroscience is not a scientific or quasiscientific pseudoscience.
Conscientiously pursue omniscience and prescience."


ðə sˈaɪəns ʌv nˈɜ:ɹəsˌiəns ɪz nˌɑ:ɾə saɪəntˈɪfɪk ɔ:ɹ kwˌeɪzaɪsiə
ontˈɪfɪk sjˈu:dəsˌiəns
 kˌɑ:nsɪˈɛnʃəsli pəsjˈu: ˈɑ:mnɪsˌiəns ænd pɹˈi:siənz

 ðə sˈaɪəns ʌv nˈʊɹɹəsˌaɪəns ɪz nˌɑ:ɾə saɪəntˈɪfɪk ɔ:ɹ kwˌeɪzaɪs
aɪəntˈɪfɪk sˈu:doʊsˌaɪəns
 kˌɑ:nsɪˈɛnʃəsli pəsˈu: ɑ:mnˈɪsɪəns ænd pɹˈi:siənz
```

This also means you can use it with `-q` as a fairly reliable converter from standard English orthography to IPA:

```
ðɪs ˈɑ:lsoʊ mˈi:nz ju: kæn jˈu:z ɪt wɪðkjˈu: æz ɐ ɹˈæpɪd ænd fˈɛɹli ɹɪlˈaɪəbəl kənvˈɜ:ɾɚ
fɹʌm stˈændɚd ˈɪŋglɪʃ ɔ:ɹθˈɑ:gɹəfi tʊ ˌaɪpˌi:ˈeɪ
```

It's a little slow for use in this mode; converting the first 83955 words of the KJV took 1'57" on my laptop, which is only 718 words per second, about three times faster than speech. But this speed is sufficient to solve many problems with. The particular problem that made me update this note tonight is that of finding sets of minimal pairs of English words for ESL learners to learn to distinguish the phonemes in, the hard part of which for a computer is finding out what the pronunciations of the English words are; the following command lines generated a decent pronouncing dictionary in just over 5 minutes:

```
$ espeak -v en-us+f2 --ipa -q < /usr/share/dict/words > words-ipa-2
$ paste /usr/share/dict/words words-ipa-2 > pronunciation-dictionary
```

Topics

First impressions on using the μMath+ calculator program for Android

Kragen Javier Sitaker, 2019-05-21 (13 minutes)

I just installed this app called "μMath+" or "microMathematics plus v2.18.0" from F-Droid on my hand computer, and it's pretty cool. It's the first calculator for Android that's good enough to criticize, so I will.

Conceptual overview

You have "worksheets" which consist of sequences of "elements" or "objects", stacked vertically and horizontally; the five types of elements are text fragments, images, equations (really assignment statements and function definitions), result views, and plots.

Text fragments are just paragraphs; the five available styles are "Chapter", "Section", "Subsection", "Subsubsection", and "Text body", all with or without numbering. There's a per-document flag to word-wrap them at some line length.

The expressions in the assignment statements are displayed using conventional math notation, and include Σ, Π, ∫, and derivatives. Values include complex floating-point numbers, Octave-like ranges of them (called "intervals"), multidimensional arrays of them (I think up to three dimensions), and versions of these dimensioned with units. Functions are defined in the assignment syntax by assigning to a function with formal parameters, as "f(x) = x² - 4". "Result views" pair a user-entered expression, on the left, with its computed result, on the right. Plots come in 2-D, 3-D, and heatmap versions; normally you evaluate a function on an "interval" to get an array result which you can plot on the y-axis. ("Intervals" work very similarly to arrays in APL and related languages, and even more so the independent variables I talked about in Relational modeling and APL (p. 1217) and A principled rethinking of array languages like APL (p. 1995). They are not the interval objects used in interval arithmetic, though they have some things in common.)

Moreover, you can export your whole document as LaTeX, though the plots are PNGs — high-resolution PNGs with transparency, but PNGs.

All in all, it can do all kinds of cool stuff.

Results

So you can, for example, write

```
E := 3 V
I := 0.1 A
E/I =
```

and it will tell you "30.0 Ω". You can change these values and recalculate. You can write

```
x := [-5, -4.9 .. 5]
y := x^3 + -2·x^2 + x
```

and then plot x versus y. In fact, you don't even need to define a variable for y; you can enter the polynomial directly on the y-axis of the plot, though because of the horizontal layout of the formula there, it takes up a lot of space. By long-pressing on the plot, you can get a button to view the values of x and y in a table.

You can intersperse your formulas and plots with explanatory text, as in Jupyter, but, at least in theory, somewhat more intimately — you aren't constrained to vertical stacking — but I haven't found a way to make that work well yet.

Missing features

Given that it does all this array and interval stuff, I was sort of hoping it would handle matrix arithmetic (matrix products and inner products and least-squares solutions of systems of differential equations and whatnot) but it doesn't.

UI infelicities

Unfortunately I'm not very happy using it. It feels very clumsy.

It's hard to figure out how to do anything, and it takes a lot of clicks, because the equation editor is a structure editor, but there's no visualization of the structure to work on, and you suffer the usual gulfs of execution. For example, x^2 is an exponentiation node, which you can create by scrolling the toolbar to the right to the create-exponentiation-node button and then pressing it, or by using the "^" key on the keyboard, and then typing "x". At times, to reach the exponent from there, you can press the soft keyboard's Enter key (when it displays as "→|") but at times you have to tap the spot on the display where you want to put the "2". It *is* possible to replace an existing node with a new node that has it as a child; for example, you can place the cursor before the "x" and insert "--", which makes the "x" node the second child of a new subtraction node, so you can transform it into $(1 - x)^2$. So far I haven't found a way to reverse this transformation (other than using undo — fortunately it does have multiple undo) or to put the focus on the exponentiation node itself so I can transform it into $x^2 + 1$.

Analogously, I haven't figured out how to change the layout, for example changing the order of elements or their stacking direction.

The use of long-press to activate many functions exacerbates the discoverability problem.

The only way to see the value of anything is to add a result view or plot to the document, and typically then to long-press on the result to bring up a dialog with a table of all of its values. This is, however, pretty awesome in that it allows you to bring up a table of all the numeric values being plotted in a plot.

You can't define an array by listing the items, as R = [1, 2.2, 4.7, 10, 22, 47]. You have to assign the individual items one at a time: R[1] := 1, R[2] := 2.2, and so on.

Error handling leaves a lot to be desired. If there's an error anywhere in a worksheet, nothing in the worksheet will evaluate. Usually it highlights the error with a red border, which you can tap on to get a transient notification telling you what the error was. I've

had times where I couldn't find the error, though, so I opened up a new worksheet. Really good error handling might include suggesting similarly spelled variables when one is not defined, or offering to create a definition for it, or offering to change it the same way you changed the definition that used to exist.

The "New document" menu item discards your current worksheet without confirmation, apparently irretrievably — though only if you're editing the default autosave.mmt.

Every time the screen orientation changes, the worksheet scrolls back to the top. Zooming is also troublesome; positioning in the document after a zoom is unpredictable, perhaps because the zoom feedback moves the document horizontally as well as vertically, while the actual resulting position is much more horizontally constrained. At times zoom is disabled for no apparent reason, and at times the document moves multiple screen widths diagonally when I'm pinch-zooming in place.

The formula formatting looks ugly; it's using a sans-serif monoline font without even any obliquing for the variables, except of course that Σ and ∫ are in a serif font with diagonal stress, and Π must be too but it looks like a child's drawing. The parentheses and brackets do not match the rest of the font, being thinner, and they aren't properly spaced; "(x + 1)!" will butt the "!" up to the parens. Superscripts and subscripts are smaller than the text they modify, but still too large and too far away; then at four levels of superscript the font has gotten far too small.

It includes export to LaTeX and HTML, but although it seems to use SVG internally (in its documentation, and maybe for its plotting), it doesn't seem to be able to export SVG.

The 3-D plots are lovely but being able to interactively rotate them to get a good angle would be a big plus. This is a general problem with a lot of things: to change a property you have to bring up a dialog box, but in order to see the result of changing the property, you have to close it. The even more egregious result of this is that there's no way to resize a 2-D plot by dragging; you have to bring up a dialog box on top of it, type in a new width in pixels, and tap OK. (And the default plot size is pretty small.)

It's possible to plot multiple functions on the same plot, and it's possible to plot parametric functions, but doing both at once seems to be impossible.

There doesn't seem to be a way to label plot axes by anything other than the actual expression that produces them, which would be useful if you have, for example, two plots covering different intervals of the same axis, which you will necessarily have to name with different variables.

You can't just sum an array or an interval; you have to go the whole nine yards with $\Sigma_{i=1}^{9} x_i$ instead of just Σx.

You can define n := [0, 1 .. 32] and then ask 2^n = and get a result, but you can't just inline the interval value in the superscript. In fact, I think that's true of all operations on intervals — the interval literal can't occur in any context other than an assignment statement.

Because it uses the regular Android soft keyboard, numeric entry is a pain; you often have to switch modes, and the number keys are always tiny.

Bugs

The HTML export doesn't include plots, or, if it does, they aren't showing up in Chrome.

Although dimensional analysis works for simple quantities, if you do this you get "V/A" instead of Ω for the units, although they're at least still correct values:

```
E0 := [2.5, 2.6 .. 3.5]
E := E0 · 1 V
I := 0.1 A
E/I =
```

However, if you make I also an "interval" without units, then you get the error message on E/I: "There are indirectly referenced intervals: [E0]". Dividing E0/I directly gives a matrix-like display of results, though with no real way to see which result corresponds to which inputs.

Although you can define an array by enumerating values as described earlier and it displays the same way intervals do, attempting to divide E/R produces the error, "Array is not allowed for this field."

I mentioned above that four levels of subscript produce text that is too small. Five levels returns to the normal size again, but it's not the normal size for your zoom level; it's the normal size for your display. So if you zoom out, your fifth superscript will be huge compared to everything else in the document, while if you zoom in far enough, it will be tiny compared to everything else, because it always stays the same size on the screen.

How to do better

Interactive calculator (p. 2771) and drag-and-drop calculator for touch devices (p. 1045) talk a bit about how calculators could take advantage of multitouch screens, but for pure formula entry, I feel like it's really easy to do better than using the standard onscreen soft keyboard — the built-in calculator app does it already. The exception is when you're assigning a name to something, when you need to be able to type the name, or when you have so many variables and functions that you need to find one by text search instead of picking from an LRU list.

The standard Android calculator these days displays the result of each formula incrementally as you edit it. This is a big help for simple calculations like 500×46.5. μMath+ doesn't, for some reason.

Note that the ersatz Casio calculator profiled in Usability of scientific calculators (p. 2379), though very limited in hardware, got a bunch of things right: once you *have* a formula, you can attach it to a name. Unfortunately this, along with the whole RPN universe, adds an extra gulf of execution to the problem: you have to figure out that the way to get "y = x²" on the screen is to first get "x²" on the screen and only then activate "STO Y", as the ersatz Casio's keyboard labels call it.

Going further, you could imagine writing *actual generalized equations* rather than assignment statements, then submitting the glob of equations to some kind of solver; or, in the opposite direction in some sense, writing explicit algorithms with loops and conditionals.

OCR of handwritten equations might also be a good approach, given how clumsy interacting on the touchscreen is.

Structure editors are hard to make usable, but not impossible. A useful step would be providing visible handles to each of the tree nodes you might be trying to select.

Given the amount of symbolic computation and ad-hoc UI interaction tailoring this problem needs, writing it in Java was probably not a good idea; a development environment that better supports symbolic computation and iterative development, especially code changes without program restarts, would probably work better.

Topics

• Programming (p. 3658) (286 notes)
• Human–computer interaction (p. 3493) (76 notes)
• Multitouch (p. 3591) (12 notes)
• Calculators (p. 3362) (11 notes)
• Hand computers (p. 3492) (10 notes)

Magic sinewave filter

Kragen Javier Sitaker, 2019-12-17 (6 minutes)

Don Lancaster's "magic sinewaves" are functions from discrete time to {-1, 0, +1} intended for use, among other things, in modulating an H-bridge to approximate a sine wave for power electronics. I think there's a way to use them to get a reasonably good and extremely efficient frequency-selective sparse filter (see Sparse filters (p. 834)).

Magic sinewaves

The simplest approximation of a sine wave of period, say, 4 samples, is a square wave: [-1, -1, +1, +1, -1, -1, +1, +1, ...]. But that isn't a very good approximation; it has a pretty loud third harmonic. (This is above Nyquist, so suppose we're using a zero-order hold here.) A better approximation, though lower amplitude, includes a 0 period instead: [0, -1, 0, +1, 0, -1, 0, +1, ...]. This has less harmonic distortion. But if we expand this out with a zero-order hold it's still obviously imperfect: [0, 0, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, +1, +1, +1, +1, +1, 0, 0, 0, 0, ...]. We can twiddle the samples around a little to kind of dither the transition; for example, [0, -1, 0, -1, -1, -1, 0, -1, 0, 0, 0, +1, 0, +1, +1, +1, 0, +1, 0, 0, 0, ...]. This way, although we don't get lower total harmonic distortion, we can push more of it up to higher frequencies, which makes it easier to filter out. Again, this gives us lower amplitude for the fundamental.

If you want a sine wave, why not just use an actual sine wave, like [0, -0.3, -0.6, -0.8, -1, -1, -1, -0.8, -0.6, -0.3, 0, 0.3, 0.6, 0.8, 1, 1, 1, 0.8, 0.6, 0.3...]? In the case of power electronics with an H-bridge, it's because your H-bridge wastes no power when it's off, a tiny bit of power when it's fully on (either forward or reverse), and a massive amount of power when it's partway on. So for an efficient system you want to switch between fully on and fully off as quickly (and infrequently!) as possible.

For sample rates that are higher compared to the signal frequency, it's easy to push all the harmonic distortion many octaves away from the signal, making it easy to filter with efficient passive filters; this is pretty much the same principle behind delta-sigma DACs, though those usually use {0, 1} rather than {-1, 0, 1}.

Filtering

The example "reference signal" waveforms above have period 20. If we were to multiply them elementwise by some signal, we would downconvert one phase of the frequency component with period 20 to DC, and then we could extract it by merely summing. The actual sine wave would give us the pure frequency component, while the {-1, 0, +1} signals mix in some pretty significant harmonic distortion. But multiplying by them is trivial: you add or subtract samples from your running total, no multiplication required. You can do this a second time with a second reference signal in quadrature with the first to get a complete measurement of the amplitude and phase of a given frequency.

If you maintain, say, 60 totals buckets, such that bucket i contains a total of the samples $x[j]$ such that $j \% 60 == i$, you can run this

analysis over the period-20 component of your whole signal by doing 60 additions and subtractions; so, too, for components whose periods are other factors of 60, such as 30, 15, 12, 10, 6, 5, and 4. If instead of just maintaining 60 such buckets, you calculate a feedback comb filter $y[n] = x[n] + y[n - 60]$, you can calculate such totals for any given segment of the signal by subtracting 60 subsequent y values at the beginning of that segment from 60 corresponding y values at its end; and the usual tricks to get a triangular or otherwise approximately gaussian temporal window apply, as described in some other notes here.

You could, instead, do this kind of correlation by generating a dithered approximation "reference signal" dynamically, for example by running a delta-sigma conversion of the output of a free-running Goertzel or Minsky oscillator, and use that to decide whether to process the current sample by adding, subtracting, or neither, to each of your I and Q accumulators. This has the great advantage that you can dynamically vary the frequency of your local oscillator, thus tracking chirp signals, such as whistles (see Whistle detection (p. 357).) A lightweight low-pass filter applied to the input signal should be sufficient to eliminate the high-frequency signals that could otherwise produce spurious correlations, but with delta-sigma conversion of more than first order, I think the dither noise might be sufficiently random to not need this.

Delta-sigma conversion in this case might be nothing more than a matter of the Bresenham line-drawing algorithm.

Prefix sums

If the number of transitions per period is relatively small, as with magic-sinewave waveforms designed to reduce switching losses by switching less frequently, it may be more efficient to work from a prefix sum of the signal rather than the signal itself. For example, the initial reference waveform suggested above has only four transitions per period; rather than doing five additions and five subtractions per period, you could do two additions and two subtractions from the prefix sum. Of course the prefix sum itself requires 20 additions per period to compute, so this is only an improvement if you can share the prefix sum with some other computation, such as the detection of a second frequency.

Topics

Fractal palettes

Kragen Javier Sitaker, 2019-04-02 (7 minutes)

The most conventional way to plot the Mandelbrot set and similar fractals is to compute the "escape time" for each pixel (the number of iterations needed for the point to exceed some "bailout value" beyond which we can be sure there is no return) and use it as an index into a cyclic one-dimensional "palette" of colors. For example, if it takes a given pixel 331 iterations to exceed the bailout value, and our palette is of 256 colors, we draw that pixel as color #75 from that palette.

8-bit color

In the early-1990s days of 8-bit color like VGA and cgsix, this approach had some very significant virtues. These devices required a number of compromises. Redrawing a whole high-resolution screen involved more memory bandwidth than our graphics cards could manage during a video frame (typically 13.9 ms rather than the modern 16.7 ms); full-screen video games like Doom and Quake typically ran at reduced resolution, and video, when it was possible at all, typically played in a postage-stamp-sized window. And, although they could typically display 262,144 different colors, they could only display 256 of them at a time, so that the framebuffer could be only 8 bits deep; full-color images needed to be dithered down to 256 colors for display. Many algorithms were devised to formulate optimal palettes for approximating a given image and dithering to those palettes with different tradeoffs between CPU usage, spatial sharpness, color fidelity, and crawling artifacts in video; we still see these today in animated GIFs. A few demos changed palette entries while the screen was being painted in order to get 256 colors per scan line instead of 256 colors per screen, but this technique required very tricky timing, and perhaps as a result never saw general use. Mozilla (later renamed "Netscape"), which had to display more than one image at a time, settled on dithering all images to a standard 216-color palette (6 levels of red, 6 of green, 6 of blue) and so WWW pages that wanted to avoid this extra dithering would use only these "web safe" colors.

It's perhaps worth mentioning that DRAM cost US$40 per megabyte from about 1992 to about 1996, due to a price-fixing cartel among RAM manufacturers during this time, so a 1280×1024 framebuffer required US$50 worth of RAM if it was 8 bits deep, a number which soared to US$150 for 24 bits, and that's not even including a second buffer for page-flipping double-buffering. So only specialized high-end video cards like Targas and the ones in Silicon Graphics machines offered 24-bit color ("TrueColor") or even the 16-bit "TrueColor" that is now universal on hand computers ("cellphones").

The escape-time-indexed-palette approach provided a few great benefits, some of which are specific to the hardware of the time:

- It ensured that the image used only 256 colors, so no dithering was needed; the display hardware could display the image natively.
- It allowed "color cycling", in which the screen was updated by

altering the hardware palette (once per frame) rather than repainting pixels into the framebuffer — functionality TrueColor cards couldn't match. Color cycling as such involved cyclicly permuting the entries in the palette. The famous Amiga bouncing-ball demo used a form of color cycling to fake real-time high-resolution 3-D on hardware that couldn't come close to doing it for real; updating the palette was also a common technique to smoothly fade images to black.

• It gives you a whole orthogonal dimension of expressivity for fractal images. You can put a color gradient in part of the palette, followed by a sharply contrasting color starting a different gradient, or a sequence of sharply contrasting colors, or colors alternating between two gradients. These techniques can produce very different images when applied to the same mapping from pixel coordinates to escape times, emphasizing some aspects of the image while soft-pedaling other parts. Furthermore, you can apply all of this instantly to an already-completed image — an enormous advantage on hardware where even a simple Mandelbrot took several seconds to render.

• In many cases, it produced large areas of solid color, which permits popular lossless compression algorithms (like the LZW used in GIF) to compress the image substantially.

Another popular option was to use palettes in the same way, but index them with something other than the escape time. For example, if you use the argument of the bailout-exceeding value to index into the palette, each region that would have been a solid color with the escape-time algorithm instead becomes a cyclic progression through the colors — a gradient, perhaps — typically repeating N times.

Generalizing palettes

This suggests a useful generalization of the palette concept to me. The initial fractal rendering process computes for each pixel a vector of result values: the number of iterations, the complex value of the escaping value, and so on. Different rendering processes might produce different sets of values; presumably John Milnor's distance-estimation algorithm, for example, won't produce those, but it might produce a different vector. Then the "palette" function maps each such vector to a color. One special case is to use just the number of iterations modulo the length of a given list of colors, but many others are possible. Other special cases include Fractint's "outside=real" and "outside=imag" options, which add the real or imaginary parts of the escaping point to the number of iterations before indexing the palette, as well as an option "outside=tdis" to use the total distance traveled by the orbit of the escaping point, again cyclically indexing the palette with the result. Fractint also has a "biomorph" option, which is Clifford Pickover's invention of overriding the color that would otherwise be used (based on the iteration count or whatever) with an image-wide constant if either the real or the imaginary component of the escaping point is less than the bailout. And a "decomp" option, which overrides the outside color in another similar way, which I think should be the same as "outside=atan" but which doesn't seem to be. And so on.

But what if your "palette" were a two-dimensional color gradient, indexed in one dimension by number of iterations and in another by the angle of the escaping point? What if the "palette" includes a truly continuous gradient, rather than a 256-color approximation to one?

What if the "palette" varies with time, as in color cycling, but in a more general fashion?

Topics

Bytecode pubsub

Kragen Javier Sitaker, 2019-12-04 (6 minutes)

In a publish-subscribe system you have a message bus to which messages are posted. The message bus may be a literal wire (as in thinnet Ethernet, CAN, and RS-485), a computer (as in Ethernet with a switch), a running program on a computer (as in D-BUS), a group of running programs on one or more computers (as in IRC or IP multicast), a data storage device (sort of as in Kafka), or some combination. The messages posted to it are then received by some subset of subscribers to the message bus.

Why filtering?

The simplest approach is to send every message to every subscriber, but this has both performance problems and security problems: making a copy of the message for a subscriber who doesn't want it is wasted work, and maybe so is the work the subscriber has to do to determine that they don't want it; and if you have a security policy that prohibits some subscribers from looking at some messages, this approach makes that policy entirely dependent on those subscribers not attempting to violate it.

So most pub-sub systems have some kind of filtering system that only sends each message to some subset of subscribers. These filtering systems can be more or less expressive; for example, you can have a disjoint set of topics (like IRC and IP multicast), a hierarchy of topics (like ZeroMQ), a non-disjoint set of tags with single-tag subscriptions, boolean tag subscriptions, boolean queries on message field values, and so on. The more elaborate filters let through a more precise approximation of the messages the subscriber is really interested in, wasting less work on forwarding messages the subscriber will ultimately discard.

Turing-complete interests

The subscriber's real interests may be Turing-complete (assuming the subscriber is a computer program --- human interests might be more complex); determining whether a packet fulfills them or not may in fact be uncomputable. In Fast secure pubsub (p. 545) I talked about running a subscriber-provided interest function in a time-limited sandbox where its accesses to message fields are recorded; if it rejects the message, those accesses are added to a cache so that any future messages with the same values in those fields will also be rejected without rerunning the function, thus saving time. Similarly, if it accepts the message or times out, any future messages that are similar in that way will be forwarded to the subscriber. (And senders can provide a filter function that determines whether or not a subscriber is allowed to examine a message; its behavior differs in that if it times out, the message is not forwarded.)

In addition to just "accepting" a message, the interest function might reasonably take other actions as well; in particular, it might post a message somewhere, and it might map the accepted message to a smaller message, so that less data needs to be copied to the subscriber itself. However, these actions are much harder to memoize with the

purely-sandboxing approach described above. Suppose the incoming message says {x: 32, y: 31, topic: "mouse"}, and the interest function inspects the topic and x fields before mapping the message to the message {p: 32} to be sent to the subscriber. The sandbox is able to determine that the y field does not matter, so future messages with the same x and mouse fields should be handled the same way. But it has no way to determine whether the message {x: 48, topic: "mouse"} should even be accepted, much less whether the resulting message should be {p: 32}, {p: 48}, {p: 16}, or something else.

Non-Turing-complete interests

But a different approach is suggested by BPF and Bitcoin Script, as described in Scriptable windowing for Wercam (p. 1256) in a different context. Instead of having the subscribers send a Turing-complete program to the message bus, they can send a program in a non-Turing-complete bytecode, perhaps one without loops or subroutines, so its execution time can be statically bounded.

This is pretty close to the original purpose of BPF and its 1980 predecessor CSPF: the packet-dumping program, tcpdump or whatever, gives the kernel a "subscription request" in the form of a BPF program, and the kernel evaluates all such program on all incoming packets, forwarding only the accepted packets to the userspace program.

The subscriber can generate the bytecode program by doing abstract interpretation of the Turing-complete program representing its interests, somewhat like a tracing JIT, but using abstract values. This generates a safe conservative bytecode approximation of its original Turing-complete program; this bytecode can then be sent to the message bus to do the prefiltering.

There is an example of how to do this in Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) in the section "Abstract interpretation with non-standard semantics".

This abstract-interpretation approach is applicable to a variety of situations in which a non-Turing-complete program is required, especially if a conservative approximation is acceptable. So, for example, given some Turing-complete specification of a security rule to make a message visible only to certain subscribers, a conservative approximation is not acceptable; this approach is only applicable to that problem if the full execution tree can be successfully explored.

Database queries

From a certain point of view, a database query is just a pubsub subscription that is immediately run on a stored history of past events; but this point of view doesn't have an obvious way to account for sorting specifications and joins, which do things like index traversals and intermediate materializations. However, the comparison operators used to construct and traverse indices, as well as the tuplewise computations used to filter and transform result streams, could profitably be specified by such bytecode chunks, rather than by an ever-growing set of data types built into the database engine.

Topics

Tagging parsers

Kragen Javier Sitaker, 2018-11-23 (updated 2018-12-10) (9 minutes)

See also Minimal imperative language (p. 2175).

I've written a PEG parser generator with inline semantic actions in itself; it was 66 lines of code, although it was fairly minimal. A more complete implementation would be a bit longer. An embedded DSL for Python that parses to syntax trees, called Tack, was 27 lines of code, but eliminating the extra crap from the syntax tree took another 13 lines on top of the grammar itself.

Inline semantic actions have the difficulty that they are host-language-specific, so if you want to parse the same data format in different programming languages, you need to rewrite the grammar over and over again. In today's polyglot programming environment, this is a serious disadvantage for the development of grammar libraries over time, as each host language has its own private set of grammars.

(And of course embedded DSLs have this problem to an even greater degree.)

As an example grammar, consider this numerical expression grammar, in a very minimal PEG language:

```
sp <- ' ' / '\n' / '\t'.
_ <- sp _ / .
digit <- '0' / '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' / '9'.
digits <- digit digits / digit.
num <- (digits ('.' digits / '.' / ) / '.' digits) _.
i <- num / '(' _ a ')' _.
e <- i '**' _ i / i.
m <- e ('/' / '*') _ m / e.
a <- m ('+' / '-') _ a / m.
```

If we have repetition * and nonempty repetition + in the language, we can simplify it a bit:

```
_ <- (' ' / '\n' / '\t')*.
digit <- '0' / '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' / '9'.
num <- (digit+ ('.' digit* / ) / '.' digit+) _.
i <- num / '(' _ a ')' _.
e <- i '**' _ i / i.
m <- e (('/' / '*') _ e)*.
a <- m (('+' / '-') _ m)*.
```

This is more or less the example grammar I used for Tack, but the tree Tack builds for it includes all the whitespace and is tagged with all the nonterminals. (Additionally, because Tack doesn't support repetition, it associates the wrong way.)

XML

A possible solution that occurred to me was to put XML-like tags into your grammar, instead of relying on the nonterminals. Then the parser's job is merely to associate these tags with points in the input text. For example:

```
_: (' ' | '\n' | '\t')*.
digit: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
atom: <num> (digit+ ('.' digit* | ) | '.' digit+) </num> _ | '(' _ a ')' _.
e: <exp> <base>atom</base> '**' _ <pow>atom</pow> </exp> | atom.
m: <term> e (<op>('/' | '*' | '%')</op> _ e)+ </term> | e.
a: <sum> m (<op>('+' | '-')</op> _ m)+ </sum> | m.
```

We can think of the parser as adding markup to the text. In fact, we can actually implement it that way, if we like. So, given the text `13 + 4 * 5 + 4 ** 3 * 1`, it will convert it to the following, with some liberty given to spacing:

```
<sum>
  <num>13</num>
  <op>+</op>
  <term><num>4</num> <op>*</op> <num>5</num></term>
  <op>+</op>
  <term>
    <exp><base><num>4</num></base> ** <pow><num>3</num></pow></exp>
    <op>*</op>
    <num>1</num>
  </term>
</sum>
```

This may not be the easiest way to read it, but it makes it super easy to apply stylesheets to it, and each span of text is tagged for the necessary processing.

If we weren't trying to look like XML, a terser and perhaps clearer syntax would be the following:

```
_: (' ' | '\n' | '\t')*.
digit: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
atom: <num digit+ ('.' digit* | ) | '.' digit+> _ | '(' _ a ')' _.
e: <exp <base atom> '**' _ <pow atom>> | atom.
m: <term e (<op '/' | '*' | '%'> _ e)+> | e.
a: <sum m (<op '+' | '-'> _ m)+> | m.
```

Or maybe with more C-like syntax:

```
_: (' ' | '\n' | '\t')*;
digit: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
atom: num { digit+ ('.' digit* | ) | '.' digit+ } _ | '(' _ a ')' _;
e: exp { base { atom } '**' _ pow { atom } } | atom;
m: term { e (op { '/' | '*' | '%' } _ e)+ } | e;
a: sum { m (op { '+' | '-' } _ m)+ } | m;
```

Or syntax chosen to be less noisy and more compact, using `;` for alternation, `{}` for grouping, `foo()` for tagging, and `.` for termination:

```
_: {' '; '\n'; '\t'}*.
digit: '0'; '1'; '2'; '3'; '4'; '5'; '6'; '7'; '8'; '9'.
atom: num(digit+ {'.' digit*; }; '.' digit+) _; '(' _ a ')' _.
e: exp(base(atom) '**' _ pow(atom)); atom.
m: term(e { op('/'; '*'; '%') _ e }+); e.
a: sum(m { op('+'; '-') _ m }+); m.
```

APIs

At the API level, aside from just generating some kind of S-expressions or XML, you could imagine either DOM-style or SAX-style interfaces. SAX-style interfaces offer the possibility of rejecting a candidate parse from the host language; for example, when parsing C, you need to distinguish between type identifiers and other identifiers, which is beyond the powers of PEGs, but straightforward if the parser can call out to a host-language symbol table. But this is somewhat more complicated than in real SAX, since it means we need to invoke host-language actions for tentative parses that may fail.

JSON

Nowadays, of course, JSON — arbitrarily nested dicts and lists, in Python parlance — is a much more popular data model than XML. What would this expression grammar look like if it were to produce a JSON structure instead? Maybe like this:

```
_ = (' ' | '\n' | '\t')*.
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
atom = #(digit+ ('.' digit* | ) | '.' digit+) _ | '(' _ a ')' _.
e = { type: `exp` base: atom '**' _ exp: atom } | atom.
m = { type: `term` head: e factors: {rator: ('/' | '*' | '%') _ rand: e}+ } | e.
a = { type: `sum` head: m terms: {rator: ('+' | '-') _ rand: m}+ } | m.
```

Here {} constructs a finite map (an "object"), `+` or `*` makes a list of each of the things it matched if they yielded some value (rather than just matching text), `#` converts the matched text into a number (perhaps overly specific to this purpose), `` `backquotes` `` enclose a literal string to be inserted into the JSON (rather than matched against the input), and `foo: bar` assigns the value produced by `bar` to the key `foo` in the object at hand. Every element of the grammar parses to some value (usually a string) but in the absence of some capturing key, the value is discarded. So the example `13 + 4 * 5 + 4 ** 3 * 1` from before becomes the following:

```
{ "type": "sum"
, "head": 13
, "terms": [ { "rator": "+"
             , "rand": { "type": "term"
                       , "head": 4
                       , "factors": [{"rator": "*", "rand": 5}]
                       }
             }
           , { "rator": "+"
             , "rand": { "type": "term"
                       , "head": {"type": "exp" , "base": 4 , "exp": 3}
                       , "factors": [{"rator": "*", "rand": 1}]
                       }
             }
           ]
}
```

OMeta

OMeta has the following syntax:

```
meta E {
  dig ::= '0' | ... | '9';
  num ::= <dig>+;
  fac ::= <fac> '*' <num>
        | <fac> '/' <num>
        | <num>;
  exp ::= <exp> '+' <fac>
        | <exp> '-' <fac>
        | <fac>;
}
```

It also supports ~negation and `kleene*` closure. Note that it's using left-recursion, which is generally fatal to PEGs, but they found a hack to Packrat parsing that allows it to work in cases like this one (though they never characterized cleanly which cases it worked for). This allows them to easily get the right associativity in this case.

They used inline semantic actions like peg-bootstrap, with postposition identifiers:

```
exp ::= <exp>:x '+' <fac>:y => `(+ ,x ,y)
      | <exp>:x '-' <fac>:y => `(- ,x ,y)
      | <fac>;
```

They also had inline semantic predicates:

```
largeNumber ::= <number>:n ?(> n 100) => n;
```

And they took advantage of the angle brackets to provide parameters to parameterized productions:

```
cRange x y ::= <char>:c ?(>= c x) ?(<= c y) => c;
... <cRange 'a' 'z'> ...
```

and they take advantage of this to hack a tokenizer into some of their grammars.

Perl6 regexes

Perl 6 regexes permit things like this:

```
my regex header { \s* '[' (\w+) ']' \h* \n+ }
my regex identifier  { \w+ }
my regex kvpair { \s* <key=identifier> '=' <value=identifier> \n+ }
my regex section {
    <header>
    <kvpair>*
}
```

Here `<header>` is invoking the regex named `header`, and `<key=identifier>` invokes the regex named `identifier` and binds its results to the variable `key`, which would default to `identifier` if it weren't specified (so `<header>`

saves the result under the name `header`.)

Inside-out quotes and inline definition (nested rather than flat)

The above suffers a bit from excessive quoting, especially in lines like this:

```
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
```

Wouldn't it be nicer to be able to write `0|1|2|3|4|5|6|7|8|9` as we can in regular expressions? (In this case, of course, `[0-9]` or `\d` might be nicer still.)

We could use a Perl-regexp-like notation in which alphanumeric text matches itself by default, without needing quotes, and uses non-alphanumeric characters to escape to metacharacter land. Furthermore we can embed our nonterminals within an expression. (Perhaps this is similar to Perl 6 patterns?) This requires two different ways of tagging segments of the pattern; for example, we could use `<>` and `[]`.

```
[e <exp <base atom> '**' _ <pow atom>>; atom]
```

Or we could use capital and small letters.

AST rewriting

In addition to the DSL for defining PEGs, it might be useful to have a DSL for describing structural rewritings of ASTs in terms of pattern-replacement pairs. It could use a pure tree-rewriting paradigm, but still support iterative computation.

Topics

- Programming (p. 3658) (286 notes)
- Parsing (p. 3618) (15 notes)
- Automata theory (p. 3335) (11 notes)
- Parsing Expression Grammars (PEGs) (p. 3620) (4 notes)
- OMeta (p. 3605) (3 notes)
- JSON (p. 3534) (2 notes)
- XML
- Perl 6

VCR oscilloscope

Kragen Javier Sitaker, 2017-05-10 (updated 2017-06-20) (2 minutes)

An NTSC video signal is 6MHz wide. A VCR records it on the tape in the cassette with maybe one 60Hz field per diagonal pass of the head across the tape, with reasonably good analog fidelity. A VHS tape uses a different modulation scheme, with about 3MHz of video (luminance) bandwidth and another 400kHz of chroma bandwidth. S-VHS has 5.4MHz

The tricky part of making a decent (20MHz) oscilloscope out of garbage is high-speed signal detection: either direct analog display or analog-to-digital conversion. Perhaps recording the signal on a video tape, perhaps with sped-up heads, would enable you to do the conversion over a longer period of time using a slower converter, converting different samples on each pass, with the tape paused.

A NTSC VHS VCR head rotates at 1800 rpm, each rotation covering a 60Hz field, lasting 16.7 milliseconds. If you were to rotate it four times as fast, 7200 rpm, each such track would only last 4.2 milliseconds, but could plausibly have up to 12MHz bandwidth, or 21.6MHz for S-VHS.

4.2 milliseconds is a ridiculously long recording time; at 60 megasamples per second, it's 250,000 samples, hundreds of times longer than is necessary for a storage oscilloscope. So, if it were mechanically practical to speed the heads up further, it would be a good idea.

(See also files TV oscilloscope (p. 1253), Laser printer oscilloscope (p. 449), and CCD oscilloscope (p. 1861).)

Topics

Three phase oscillating belt

Kragen Javier Sitaker, 2018-10-28 (4 minutes)

Belt drives or cable drives can transmit power somewhat more flexibly and, I think, efficiently than shaft drives. A belt or cable doesn't need enough material to be rigid, just enough to withstand the tension, and it can in theory move arbitrarily fast.

Belt or cable speeds, however, are limited by centrifugal forces around pulleys, which need to have larger radii in order to proportionally reduce the centrifugal forces at a given linear speed — and the area of the pulley increases as the square of its radius. And, of course, at least half the material of the belt is "wasted" in the sense that it's not carrying any energy, just being returned from the power source to the load under minimal tension.

Suppose that instead of an endless belt we use a straight tension cable, as in a child's pullstring toy, and instead of transmitting power continuously, we transmit power intermittently. Perhaps 90% of the time, the rope is moving from the load to the source at its maximum safe tension of 1000 N and at 3 m/s, and 10% of the time, it's moving from the source to the load at 100 N and roughly 30 m/s. Such a rope is transmitting an average power of 2700 W, the equivalent of 12 amps at 230 volts.

0.42 mm² of UHMWPE string can handle 1000 N, assuming 2.4 GPa and no safety factor. Such a string weighs 410 mg/m; 100 m of it weighs 41 g. 100 N accelerates 41 g at 2400 m/s², so it can reach 30 m/s in 12.3 ms. This means you could quite reasonably transmit power in this way with a 10 Hz oscillation in the line, which means the string would move only about 30 cm.

Piano wire also has a yield stress of about 2.5 GPa, although to avoid fatigue you're supposed to stick to a third of that or so. So a similar amount of piano wire could also transmit a similar load. It would weigh 8 times as much, and it would have more friction, but it wouldn't be as vulnerable to overheating.

(Hmm, I realize now that the 100 N restoring force is really nothing more than a centripetal force.)

If instead of one cable under tension, you have two, you could maintain continuous power with no intermittency; but, if the power being transmitted doesn't vary, you could never exceed the power that one cable could transmit at a time, so in a sense the other cable is wasted.

If, instead of two cables, you have three cables, you can have two cables transmitting power at any given time while the third is returning. This allows two-thirds of the mass and volume of your transmission cable to be used for power transmission, while keeping the power transmission perfectly consistent. Furthermore, this can avoid the large accelerations implied in the single-cable skewed triangle I suggested earlier; the speed of the cable that's about to go into takeup can diminish gradually while the cable that's just gone into the power cycle can smoothly speed up, maintaining a constant total speed. This is highly desirable, given that the mechanisms driven by the cable can easily weigh much more than the cable, and even 100 m is not very far in some contexts.

It should be emphasized that this kind of three-phase power transmission cannot use the same sinusoidal waveforms used by electrical AC power transmission if two cables are always to be kept in tension.

The three phases can drive a common differential via freewheel clutches.

Topics

Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm

Kragen Javier Sitaker, 2019-01-04 (updated 2019-11-12) (26 minutes)

I was thinking about how to implement the grayscale morphological "erosion" and "dilation" operations efficiently with a large irregular structuring element, and I think I've found some interesting algorithms; they may be novel.

In the contex of sampled grayscale images, erosion $I \ominus k$ maps each pixel in the image I to the minimum of some neighborhood k around that pixel, the neighborhood being the "structuring element", which I will call a "kernel" for brevity. (Dilation $\oplus$ is a similar operation with some differences that I won't mention.)

The naïve approach

The naïve approach to computing this gets slower with large kernels. Consider computing, in one dimension, a kernel that consists of the hotspot pixel and $n-1$ pixels to the right:

```
for (int x = 0; x < w; x++) {
    out[x] = in[x];
    for (int i = 1; i < n; i++) out[x] = min(in[(x+i) % w], out[x]);
)
```

This takes time per pixel proportional to n.

Sliding-window minimum in linear time

In one dimension, there's a well-known linear-time algorithm for this "sliding-window minimum" or "sliding range minimum query" problem, using a deque d containing a nondecreasing subsequence of the window such that the leftmost element in the deque is always the minimum of the window. Incrementing the left edge of the window may leave the deque unchanged, or it may involve dropping the oldest value, if that value falls out of the window; then the next value must be the minimum of the remaining pixels in the window. Achieving this merely requires that, when we increment the right edge of the window, we remove any elements in the deque that are larger than the new pixel, and then append that new pixel. This results in the pixels in the deque being in nondecreasing order, which means that any possible larger pixels will be in a block at the end of the deque, so we can remove them by popping from its end.

This sounds trickier than it is. In pseudocode:

```
d = deque()
for x in range(len(inpix)):
    while d and inpix[d[-1]] > inpix[x]:
        d.pop()
```

```
    d.append(x)
    if x - d[0] == n:
        d.popleft()
    yield inpix[d[0]]
```

Each pixel is pushed onto the deque exactly once and removed from it exactly once, at either the left or right, and each pixel comparison either results in pushing a pixel or popping one, so there can't be more than two comparisons per pixel overall. So the algorithm is linear-time despite its nested-loop appearance.

This depends on the assumption that the deque operations are constant-time operations, which is easy to guarantee even in the worst case if we have a bound on the deque size, which we do; it can't be larger than *n*. So we can render this into C with variable-length arrays as follows (see http://canonical.org/~kragen/sw/dev3/erosion1d.c):

```
unsigned d[n], di = 0, dj = 0;
for (int x = 0; x < w; x++) {
  while (di != dj && in[d[(dj-1) % n]] > in[x]) dj--;
  d[dj++ % n] = x;
  if (x - d[di % n] == n) di++;
  out[x] = in[d[di % n]];
}
```

Despite the divisions in the inner loop, for one-byte pixels, this takes 45–65 nanoseconds per pixel on my laptop with window widths ranging from 1 to 10000, although it *is* a bit quicker with one-pixel windows. (You could probably run it a lot faster without the divisions.)

This algorithm clearly has worse constant factors than the naïve algorithm, so the naïve algorithm is probably faster for sufficiently small kernels, but I haven't optimized and measured to see exactly how big the kernel needs to be for this algorithm to be faster. I'm pretty sure the naïve algorithm is going to be faster for 3-pixel-wide kernels.

I think this algorithm may be due to Richard Harter in 2001, who called it "the ascending minima algorithm", but I'm not sure.

As an interesting side note, a slight variation of this algorithm computes the convex hull of a series of points in 2-D in linear time. It consists of a basic pass which computes the upper convex hull, which is applied a second time under a suitable transformation to compute the lower convex hull, which together (perhaps with vertical lines to join them) form the overall convex hull. Each pass iterates over the points in increasing order of X coordinate, just as the sliding-window algorithm does; the upper convex hull is accumulated on a stack (rather than a deque) and points are popped off the stack if, considering the new point being added, they would make the hull non-convex, rather than if they are greater than the new point being added. This maintains the invariant that the sequence is convex upwards, rather than that it is nondecreasing. This is important for efficiently finding affine-arithmetic approximations of empirical data, as explored in An affine-arithmetic database index for rapid historical securities formula queries (p. 2275).

There's a completely different algorithm with similar linear performance published by van Herk in 1992 in a paper entitled, "A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels" and concurrently by Gil and Werman. It

divides the array into N-sample blocks and computes prefix-sum minima within each block going both left "h" and right "g". Any N-sample window will include some number $x \in (0, N]$ of samples in its leftmost block and some number $N - x \in [0, N)$ of samples in the block to the right; the minimum of the x samples in the left block can be found in the "h" array for that block, and the minimum of the $N - x$ samples in the right block can be found in the "g" array for that block. The minimum of these two numbers is the minimum over the whole block.

Separable kernels

As with box-filter convolution and Gaussian convolution, we can decompose erosions with certain two-dimensional kernels into compositions of two erosions with one-dimensional kernels, one in X and one in Y, thanks to a sort of distributive law (sometimes called "the chain rule"):

$$(I \ominus k_1) \ominus k_2 = I \ominus (k_1 \oplus k_2)$$

This is great, because it means we can erode an image with a paraxial rectangle of any size and shape in 90–130 nanoseconds per pixel, because it takes six pixel-minimum operations per pixel. (This is faster than just generalizing the van Herk–Gil–Werman algorithm to rectangles, which takes 15 pixel-minimum operations per pixel.) Great, right? But it's a paraxial rectangle. One could wish for something more. For example, many real-world images have features that are rotationally invariant, so a circle or annulus or something might be a more interesting kernel.

The distributive law says that eroding with two kernels is the same as eroding with the dilation of the kernels, which in this context is just their Minkowski sum. For example, dilating a kernel with a kernel that is a line in whatever orientation, that sort of pulls it apart in the direction of the line, filling in the gap by adding two straight facets in between, facets of the orientation and length of the line. So eroding an already-eroded image with such a kernel is the same as expanding its erosion kernel in that way.

So if you erode with two kernels that are lines, you get the erosion of a parallelogram, or in the degenerate case, a longer line. But so far we've only covered how to erode efficiently with horizontal and vertical lines.

How about diagonal lines? The one-dimensional sliding-window minimum algorithm is just as happy to run along a diagonal of the image pixels as along a row or column. If your kernel is literally just a diagonal line of pixels like [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)], then you might get some checkerboarding artifacts in the resulting image, but if you're also using a horizontal or vertical kernel, those should go away.

This carries over to skewed-line kernels like [(-4, -2), (-2, -1), (0, 0), (2, 1), (4, 2)] or [(0, 0), (1, 3), (2, 6)]. These have big gaps in them, so they would be shitty kernels by themselves, but used to dilate a kernel that fills those gaps on its own, they can give you not only excellent approximations of circles but also lines and arbitrarily-oriented capsules.

It turns out this approach to approximating circular kernels was

published by Adams in 1993 in the paper, "Radial decomposition of discs and spheres", except for the bit about skewed-line kernels with gaps in them.

Bresenham lines

I was also thinking that you could get a good approximation of a line kernel by running the one-dimensional algorithm along gap-free rasterized diagonal lines — essentially running it in the X or Y direction, but occasionally "jumping the groove" to an adjacent raster; this will give you a slightly different kernel at each pixel, but probably the errors are insignificant for most applications. But now I think that there is no real advantage to that approach over the precise approach described earlier.

It turns out that Soille, Breen, and Jones published essentially this algorithm in 1996 in their paper, "Recursive implementation of erosions and dilations along discrete lines at arbitrary angles", but using the van Herk/Gil–Werman one-dimensional sliding-window minimum algorithm. They note this approximation drawback:

> …it is important to note that the shape of the SE [kernel] may vary slightly from one pixel to another. Indeed, except for horizontal, vertical, and diagonal directions, a discrete line in the square grid contains 4- as well as 8-connected pixels. Hence, when the SE of length *k* moves along such a discrete line, its shape varies accordingly to its position along the line. In practice, this is not a major drawback provided that the angle specified is actually defined in the neighborhood corresponding to the extent of the SE.

Then they go on to define a "translation-invariant implementation" that avoids this non-major drawback, which has a great deal in common with what I've described above but may not be exactly the same; I'm not sure yet. It does seem to work better for their radial-decomposition purpose.

Also they point out a use of this algorithm to detect linear features by combining linear openings at different angles by max, rather than by composing linear erosions and dilations to get a round kernel.

Gapped one-dimensional kernels to accelerate the naïve algorithm

One way to get an erosion with a 9-pixel-wide kernel is to erode with a 3-pixel-wide kernel four times. But a better way — aside from the deque algorithm described previously — is to erode first with the kernel [-1, 0, 1] and then the kernel [-3, 0, 3]; and, to get only a 7-pixel kernel, you could erode with [-1, 0, 1] and then [-2, 0, 2]. If you want a 27-pixel-wide kernel, you can do [-1, 0, 1] ∘ [-3, 0, 3] ∘ [-9, 0, 9], and by reducing the width of this last kernel, you can get any number less than 27, too. (Though, see below (‡) for why using kernels with two pixels is better than using kernels with three.)

This points out a log-linear-time algorithm to erode with large kernels (linear in the image, logarithmic in the kernel) in a single dimension, which, unlike the deque algorithm, is easy to parallelize and incrementalize. For example, this algorithm can be straightforwardly applied to several scan lines in parallel using vector instructions, while the deque algorithm can't. (The van Herk/Gil–Werman algorithm can, though.)

(And of course it applies to sliding-window-minimum-type problems in non-image-processing contexts, as well.)

This implementation of the algorithm takes 16 nanoseconds per byte to compute a 50-byte erosion on a 10-million-byte input without paying proper attention to the boundary conditions:

```
static inline char cmin(char a, char b) { return (a < b) ? a : b; }
int x;
for (x = 0; x < w- 3; x++) out[x] = cmin(in[x],  cmin( in[x+1],  in[x+ 2]));
for (x = 0; x < w- 9; x++) out[x] = cmin(out[x], cmin(out[x+3], out[x+ 6]));
for (x = 0; x < w-27; x++) out[x] = cmin(out[x], cmin(out[x+9], out[x+18]));
for (x = 0; x < w-50; x++) out[x] = cmin(out[x], out[x+23]);
```

This is about four times as fast as my implementation of the deque algorithm above, although probably that's just because of the three divisions in its inner loop.

But you can pipeline this algorithm into requiring only a single pass over the input:

```
for (x = 0; x < w-50; x++) {
  out[x+23+18+6] = cmin(in[x+23+18+6], cmin(in[x+23+18+6+1], in[x+23+18+6+2]));
  out[x+23+18] = cmin(out[x+23+18], cmin(out[x+23+18+3], out[x+23+18+6]));
  out[x+23] = cmin(out[x+23], cmin(out[x+23+9], out[x+23+18]));
  out[x] = cmin(out[x], out[x+23]);
}
```

For some reason, this version runs at the same speed as the many-pass version, even over the same 10-million-byte input.

Union kernels

This can be generalized! And, not surprisingly, someone already has.

$I \ominus (k_1 \cup k_2)$, the erosion by a union, is the same as $(I \ominus k_1) \wedge (I \ominus k_2)$, where $\wedge$ is the pixelwise minimum operation. Urbach and Wilkinson published an algorithm in 2008 (doi 10.1.1.442.4549) that decomposes an arbitrary kernel ("flat", they say, meaning that — as in all of my discussion above — the kernel contains only full and empty pixels, no shades of gray) into scan lines ("chords").

Urbach and Wilkinson use the fact that the erosion by a kernel consisting of N consecutive 1s on a single scan line, composed with erosion by a kernel consisting of two 1s on a single scan line at (possibly non-consecutive) positions 0 and M (the kernel [0, M] in the notation I used above), computes the erosion by a kernel of N+M consecutive 1s. This is a special case of the distributive law mentioned earlier, that $(A \ominus B) \ominus C = A \ominus (B \oplus C)$ — the N+M consecutive 1s are the dilation of the N consecutive 1s and the two 1s at positions 0 and M. So Urbach and Wilkinson use this to compute the erosions of the image by "chord" kernels of lengths 1, 2, 4, 8, 16, … with one comparison operation per pixel per binary chord length. Given these, they can compute the erosion of the image by any arbitrary chord length by taking one of those images and eroding it by another two-separated-pixel kernel in, again, one comparison operation per pixel; for example, to get erosion by an 11-pixel chord, you erode the 8-pixel-chord-eroded image with the kernel 1 0 0 0 0 1, that is, with pixels in positions 0 and 5.

A union of such chord kernels can thus compute the erosion by an

arbitrary flat kernel with two comparison operations per pixel per chord of the kernel, once results for enough chord kernels of powers of 2 have been computed.

This algorithm is claimed to be considerably faster than the others I mentioned above, and now that I understand it, I believe it.

However, I think we can do better. You can see this as a special case of $(A \ominus B) \ominus C = A \ominus (B \oplus C)$, but you can also see it as a special case of $I \ominus (k_1 \cup k_2) = (I \ominus k_1) \wedge (I \ominus k_2)$, where (except in the last step, where different scan lines are brought together) the two kernels are always the same kernel with different pixel shifts. (We can probably assume that the pixel shifts are free until the kernel gets large compared to the image size.) So the generalized problem is to minimize the cost of a DAG where the nodes are of the form (I_0 shifted (x_0, y_0)) *op* (I_1 shifted (x_1, y_1)), where *op* is either $\vee$ or $\wedge$, to build up the kernel we want, starting with only the identity kernel, then taking unions and intersections of (shifted) existing kernels.

In some cases it is clear that there are better alternative strategies; for example, with a vertically symmetric kernel like a circle, you can compute each chord length once instead of twice, and if there are N adjacent scan lines with the same chord, you can use $\lceil \lg N \rceil$ comparisons to compute that block rather than N-1. Urbach and Wilkinson's paper also gives an H-shaped example kernel of 49×49 pixels, consisting of three overlapping 1×49 pixel lines (one horizontal and two vertical), which also clearly has a much simpler decomposition: a 49-pixel vertical line, unioned with itself shifted 48 pixels to the right, and a horizontal line of between 47 and 49 pixels. This H could be decomposed into horizontal chords of 2, 4, 8, 16, 32, and 47 pixels, vertical chords of 2, 4, 8, 16, 32, and 49 pixels, and two more (shifted) pixelwise minimum operations to combine the three; this requires 14 pixelwise minimum operations rather than the 54 they implicitly report in their paper.

I *think* it is the case that $I \ominus (k_1 \cap k_2) = (I \ominus k_1) \vee (I \ominus k_2)$, thus allowing us to intersect erosion kernels as well as take their union — but I'm not sure right now, because that would seem to suggest that if you intersect the identity kernel with a shifted version of itself, you get a well-defined dilation rather than an ill-defined erosion with an empty set; I'm not sure this doesn't suggest that sometimes you'll get a dilation or a combination of dilation and erosion in less pathological cases.

If we exclude intersections and have only unions, then there is a computable algorithm for finding the optimum DAG: exhaustive search of all the possible ways to compute all kernels that could fit into our desired kernel somewhere. I think we can use A* search to get a better, perhaps even computationally tractable, strategy for finding the optimum DAG. Failing that, a heuristic optimization algorithm that starts with the feasible Urbach–Wilkinson strategy and attempts to improve it is a reasonable approach.

Many of these search algorithms could probably be sped up with a precomputed database of all the possible DAG nodes accessible in, say, four or five pixelwise minimum operations, if limited to some sort of reasonable shift radius.

My suggestion (‡) in an earlier section that making a pipeline out of erosion operations would be most efficient when each kernel in the pipeline had three pixels active was ill-founded; in fact the optimum

is two, because you find the minimum of two pixels in one operation, not two.

Urbach–Wilkinson with an ascending minima stack

The Urbach–Wilkinson algorithm computes one eroded image for each distinct horizontal chord length in time logarithmic in the maximum chord length and linear in the number of chord lengths, then combines these horizontally-eroded images with vertical and horizontal offsets to get the final result. Above I point out that there are sometimes more efficient ways to combine the horizontally-eroded images that require less than the one operation per chord Urbach–Wilkinson give. Also, though, we can compute those horizontally-eroded images in a more efficient manner than Urbach and Wilkinson's algorithm when we have a small number of chord lengths — by using the ascending-minima algorithm.

As an extreme case, consider using Urbach–Wilkinson to erode by a paraxial square 65×65 kernel. Each final image pixel is the minimum of 65 vertically-adjacent pixels in a single intermediate image that is eroded by a 65-pixel horizontal-line kernel; to compute this intermediate image, the UW algorithm first computes the erosions by horizontal-line kernels of 2, 4, 8, 16, 32, and 64 pixels, requiring 6 min operations per pixel. A seventh min operation per pixel gives the 65-pixel erosion we needed. (This doesn't require nearly as much memory traffic as you might think, because you can pipeline it just as you can some of the algorithms discussed earlier, or as in Evaluating DSP operations in minimal buffer space by pipelining (p. 321).)

By contrast, the ascending-minima algorithm does 2 pixel-min operations per pixel, plus some pointer comparisons, to achieve this or any other single-window-width sliding-window erosion.

You might think that the ascending-minima algorithm would require a separate deque for multiple window widths. But consider what happens if we implement the deque as a stack of all nondecreasing pixels on the line, with a "bottom pointer" that we sometimes increment and never decrement. When we push a new pixel onto the stack, we need to check to see if the bottom pointer must be incremented because its pixel has aged out; when we pop pixels from the stack in order to push a smaller pixel, we need to ensure that the bottom pointer points to the newly pushed pixel or something to its left.

But the actual contents of the stack do not depend on the bottom pointer, and the bottom pointer is the only thing that depends on the window width. So many bottom pointers can share the same pixel stack. And so adding more bottom pointers doesn't add more pixel comparisons; those remain at 2 per pixel, regardless of the number of chord lengths demanded. The extra bottom pointers do, however, add more index comparisons, which are about 2 per chord length per pixel.

Conceivably in practice the extra log-N factor is too small to matter.

Of course this whole technique to get an arbitrary number of sliding windows out of a single ascending-minima stack is applicable

to other applications of sliding-window RMQ, not just image erosion.

You could try to make a sort of "spaghetti stack" out of this approach by making a note under each pixel of the pixel index at which it gets popped off the stack and of the index of the pixel under it on the stack, but in this simple form, this modification doesn't buy you anything useful — computing a sliding-window RMQ is already as fast as it's going to get, and it doesn't make it fast to compute a random-access RMQ, because the "stack" isn't random-access — you have to chase the pointers up it, and you can't traverse it downwards at all except by linearly searching all the pixels. If you additionally note the replacement of each pixel (if any) when you pop it, a sort of next-child pointer, it becomes possible to traverse the tree downwards in a useful way, but you still have no guarantee of the kind of balance or random access among children that would make this efficient — if the pixels happen to be in descending order, the root will have them all as its children.

You could maybe do some kind of skip-list or tree-balancing thing, but this avenue of investigation seems progressively less appealing.

Linear convolution

You can use the Urbach–Wilkinson decomposition into "chords" to accelerate the usual kind of convolution, too, the kind where you take a weighted sum of the selected neighborhood pixels instead of their maximum or minimum. See Real-time bokeh algorithms, and other convolution tricks (p. 2661) for details. (It also includes some tricks for decomposing angled lines that can be used for the morphological operations discussed in this note.)

Range minimum query

All of the algorithms mentioned here work by comparing intact pixels (as opposed to, say, computing histograms of pixel values; there are somewhat efficient algorithms for general rank-order filtering, which is a generalization of erosion, dilation, and median-filtering, that work that way — see Median filtering (p. 3155)). This means that we can augment the pixel values being considered with extra metadata. For example, we could compute the luminance of each pixel and use that for the comparisons, but drag the original pixel data along with the luminance so that the color follows as it should. Or we could tag the original pixel coordinates onto the end of the pixel data, thus solving the range-minimum-query problem generalized to arbitrary window shapes — when we find the minimum pixel within the erosion window by these algorithms, it comes with its original coordinates.

See further applications in Query evaluation with interval-annotated trees over sequences (p. 1423).

The Urbach–Wilkinson algorithm constructs an index data structure of size $O(N \lg N)$ which answers range minimum queries in constant time. There is a known RMQ algorithm based on Cartesian trees that also answers range minimum queries in constant time, but using an index data structure of size only $O(N)$. I don't understand it well enough yet to compare the two algorithms.

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)

Database explorer

Kragen Javier Sitaker, 2017-06-20 (2 minutes)

Zing! Poof! Bam! Let's explore this database, composing queries that make sense, one step at a time, with feedback after each keystroke.

Start typing the name of a table or column. See the names of the matching tables, along with their data. Tab, type another, see them together. See likely candidate join columns highlighted, with quick keystrokes to select them.

Behind the scenes, the database explorer is shitting out hundreds of queries a second and pulling a few rows from each one, using multiple different worker threads in case one of the queries is unresponsive.

An operand stack is displayed, updated after each keystroke. Available keystroke commands apply to the top stack items, but shuffling them around is easy. Since each value on the stack is an entire relation (associated with the expression that produced it) it can occupy potentially much more space than you have on your screen. The displayed summary includes the most common values in each column and, in some cases, other presentations such as maps for geodata.

As you type them, numbers and literal strings are incrementally searched for at the beginnings of all indexed columns and, if time permits, unindexed columns too.

The interface is modeless, so all the command ("accelerator") keys are modified with control or alt, and the available commands are always displayed in the UI.

Once you have the query you want, you can convert it to SQL.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Databases (p. 3400) (20 notes)
- SQL (p. 3729) (6 notes)

Using bytecode won't make your interpreter fast

Kragen Javier Sitaker, 2007 to 2009 (26 minutes)

This is responding to this paragraph of a post by Ola Bini, "Can Bytecodes Perform Well?" [0]:

> Let's see, what's next? Oh yeah. Bytecodes are always slow. No argument here. C++ will always beat a bytecode based engine. And C will almost always beat C++. And assembler (in the right hands) will usually beat C. But wait... Isn't Java bytecode based? And doesn't Java 6 perform on the same level as C in many cases, and in some cases performing better than C? Well, yes, it does... And wasn't Smalltalk always based on bytecodes? Most Smalltalk engines performed very well. Why is MRI switching to bytecodes for 1.9? And why has Python always been bytecode based? And why is the CLR bytecode based? Why was even Pascal using bytecodes back in the day? (OK, that is cheating... Pascal used bytecodes for portability, not performance, but it still worked well in that aspect too). Erlang is bytecode based.

He's taking issue with a comment of mine on his previous post; I quoted him saying, "[Rubinius] is byte code based. This means it's easier to handle performance," and flippantly responded, "If by 'handle' you mean 'not have'." [1]

Summary of My Rebuttal

I didn't say bytecodes are always slow. Bini's fast examples are actually native-code compilers, not bytecode interpreters. Bini's examples that aren't native-code compilers are slow. Python, for example, is slow when the inner loop is in Python bytecode. Erlang used to be really slow when it was a bytecode interpreter. OCaml's bytecode interpreter is considerably slower than its compiled code. But the OCaml native-code compiler is considerably simpler than the interpreter. On-the-fly "JIT" compilers don't need bytecode. In fact, bytecode may be a bad input format for JIT compilers. There exists an ahead-of-time Java compiler that doesn't use bytecode, and it produces code comparable to that produced by JIT compilers.

There are valid reasons to use bytecode, but performance is not one of them.

Several of Ola Bini's statements are dubious.

Detailed explanations of each of these statements follow.

I Didn't Say Bytecodes Are Always Slow

My original statement was not that "bytecodes are always slow," but I can see how it would be interpreted that way. I said that building a language interpreter around bytecode makes it easier to not have performance. More specifically, I'm saying that building an interpreter as a bytecode interpreter does not make it fast, not even necessarily faster than today's popular Ruby interpreter.

Bini's Fast Examples Are Actually Native-Code Compilers

Building an interpreter as an on-the-fly compiler *does* generally make it faster than a bytecode interpreter; recent Java virtual machines, the CLR, and the fast Smalltalk engines he's talking about are all on-the-fly native-code compilers, not bytecode interpreters. The fact that they take bytecodes rather than source code as input is incidental.

However, several of them are still fairly slow in absolute terms, even though they're much faster than interpreters.

At the time that Urs Hölzle wrote his dissertation [12], the ParcPlace Smalltalk system (release 4.0) ran about ten times faster than a deliberately naïve native-code compiler for Self (see section 4.4 of the dissertation), but it ran still around 4-10 times slower than C++ (see section 7.3.1). I don't know how fast Smalltalk systems are now, but I have the impression that they haven't sped up much, except for Strongtalk.

The Erlang HiPE just-in-time native-code compiler gets speedups of about 4 over the BEAM bytecode interpreter, but it's still about ten times slower than C in the "shootout" microbenchmarks. (See the section below about Erlang.)

Bini's Examples That Aren't Native-Code Compilers Are Slow

Python, the bytecode Pascal compiler/interpreters, the BEAM bytecode Erlang interpreter, and bytecode implementations of Smalltalk (such as Squeak) are and were painfully slow for anything CPU-intensive. I used the UCSD P-System on a 4MHz Z80, and I use Python and Squeak today, and they are all painfully slow when you're doing anything CPU-bound --- unless you can push the inner loops out of the bytecode interpreter, as with NumPy. Typically the performance penalty over machine code is around a factor of 100; register-based "bytecode" interpreters like Wheat's, and Lua's often get the penalty down to a factor of 10 to 30.

You can see this pattern in the Great Programming Language Shootout results; the default weightings mostly represent CPU-intensive tasks.

http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=all

Python, For Example, Is Slow When The Inner Loop Is In Python Bytecode

If you look at the Python results, the tests where it was less than ten times slower than the C version are, with two exceptions, exactly the ones where the inner loop of the test was performed in C.

http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=python&lan°
°g2=gcc

chameneos: tests thread rendezvous speed;
cheap-concurrency: tests coroutine creation speed; Python is faster than C here;
k-nucleotide: largely tests string hashing speed (note Python is 9x

slower than C in this test);
nsieve: the "inner loop" here is essentially a NumPy-like array operation:

```
a[i + i::i] = (0,) * (m // i - 2 + (not not m % i))
```

regex-dna: the inner loop is in the sre regex engine, which is in C
pidigits: the inner loop is in gmpy, the Python binding to the GNU Multiple Precision library, which is in C
reverse-complement: the inner loops are in string.translate and reversing a list, which are done in C
partial-sums: this appears to be an exception; a straightforwardly written iterative Python solution is only six times slower than the similarly straightforwardly iterative C solution, doing a bunch of floating-point.
sum-file: the entire program contains no iteration in Python bytecode; the inner and only loop consists of four primitives implemented in C: sum, itertools.imap, int, and iter(sys.stdin).

Python uses "generator" objects for the cheap-concurrency test; these are implemented in the CPython interpreter as normal function activation records [[which presumably have some name]] rather than full threads. But the C version of the test uses full POSIX threads. A C version that used user-level "green threads" or "fibers" would probably do a little better, but it would still have to allocate a whole stack for each thread, rather than running them all on the same stack. The nearest C equivalent is probably Adam Dunkels's "protothreads".

I hypothesize that the "inner loop" in the chameneos case is actually in the underlying POSIX thread library, since the Python version is only 50% slower than the C version.

I don't know why the partial-sums test is so close.

On the tests where Python does worst, the Python program is 100–250 times slower than the corresponding C program; these are generally the tests where the Python version is most similar in structure to the C version. See, for instance, "recursive" (279x), "n-body" (163x), or "mandelbrot" (117x).

http://shootout.alioth.debian.org/gp4/benchmark.php?test=recursive&lang=all
http://shootout.alioth.debian.org/gp4/benchmark.php?test=nbody&lang=all
http://shootout.alioth.debian.org/gp4/benchmark.php?test=mandelbrot&lang=python&id=3

The large amount of variability between Python's best and worst results is mostly accounted for by variation in how much of the benchmark's inner loop is inside of some extension module or primitive implemented in machine code, and how much is actually executed by the bytecode interpreter.

Useful resources for comparing language implementation performances in this manner include Doug Bagley's original Great Programming Language Shootout from 2001 [10], the shootout.alioth.debian.org "Computer Language Benchmarks Game" version thereof, the Win32 version thereof at http://dada.perl.it/shootout/ , and Kernighan's "Timing Trials" paper [9].

Erlang Used to be Really Slow When it Was a Bytecode Interpreter

Erlang today can use a native-code compiler called HiPE. The first versions of HiPE compiled bytecode to native code; current versions compile Erlang source to native code. HiPE sped up various Erlang microbenchmarks by factors of up to about 4 over the BEAM bytecode interpreter. [4] (Of course, some microbenchmarks hardly sped up at all.)

Many people think Erlang is still fairly slow; in the shootout microbenchmarks, even with HiPE, it tends to come in about 10x slower than C.

http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=hipe&lang2o o=gcc

OCaml's Bytecode Interpreter Is Considerably Slower Than Its Compiled Code

Although I previously saw speed ratios of only about a factor of 4 in some code, eyeballing numbers from the Win32 Shootout suggests that ocamlopt-generated native code is more typically 20–100 times faster than bytecode run by its bytecode interpreter.

http://dada.perl.it/shootout/ocamlb.html http://dada.perl.it/shootout/ocaml.html

OCaml's Native-Code Compiler Is Considerably Simpler Than Its Interpreter

This is kind of an interesting case, because both the native-code and bytecoded implementations of OCaml were implemented by a small team and are fairly compact. In Bini's post, he asserts, "Granted, a naive implementation of a bytecode engine will not perform well. But that is true for a compiler too."

David A. Wheeler's SLOCCount 2.26 tells me that the "bytecomp" and "byterun" subdirectories of OCaml-3.09.2 contain 12683 lines of C (in "byterun") and 8816 lines of ML (in "bytecomp"); I believe these 21.5KLOC represent the whole bytecode interpreter. (The parser, type-checker, standard library, etc., is shared with the native-code compiler.)

SLOCCount also tells me that the "asmcomp" and "asmrun" subdirectories contain 9457 lines of ML, 3910 lines of assembly, and 1077 lines of C, for a total of about 14.5KLOC. This includes backends for i386, ARM, AMD64, Itanium, PowerPC/POWER, SPARC, MIPS, HP-PA, and Alpha; the backends other than the i386 backend total about 2200 lines of ML and 3400 lines of assembly, so you could reduce that to about 10KLOC for a single CPU architecture.

In short, the native-code compiler for OCaml is about half the size of its interpreter, and so is presumably much more "naive", in the sense that it required much less effort to implement. (SLOCCount estimates three person-years, compared to five for the interpreter.) But it still performs one to two orders of magnitude better than the bytecode interpreter.

Although Bini anticipates this criticism by referring to OCaml's "extremely stringent type requirements" and calls it a

"bondage-tightly typed language", this point generalizes beyond OCaml.

It's true that OCaml's semantics require much less type-checking and dynamic dispatching than Java, Python, or Ruby, and the overhead of type-checking in these more-dynamically-typed languages can be substantial compared to the overhead of dispatching bytecodes, which makes the savings of not dispatching bytecodes less noticeable. But the next few sections show that the performance advantages of native-code compilation are not limited to static languages like OCaml; indeed, the performance benefits Bini attributes to the use of bytecode are actually due to the use of native-code compilation.

On-The-Fly "JIT" Compilers Don't Need Bytecode

In Bini's post, he asserts:

> ...Java and the CLR family of languages use bytecodes because it gives the runtime system the opportunity to dynamically change the machine code running based on statistics and criteria found out during runtime. ... This is not possible in a clean compilation to machine code. Java would never have had the performance it has now if it weren't for the bytecodes.

This is simply mistaken. There are on-the-fly machine-code compilers that work from source code instead of bytecode. SBCL and the current HiPE compiler are examples. There are some advantages to compiling directly to machine code from source code rather than compiling from a bytecode format intended for interpretation; typically the translation to bytecode erases a lot of information that is helpful to optimization.

Bytecode is a Bad Input Format for On-The-Fly "JIT" Compilers

As one of the HiPE papers [4] explains:

> A new feature, described further below, is that the HiPE compiler can compile directly from Core Erlang [a restricted subset of Erlang]. When used in this way, the compiler compiles a whole module at a time, and performs global analyses and optimizations which are significantly more difficult to perform (and thus not available) in the traditional mode [which compiled from the BEAM bytecodes].

The GCC Java compiler, GCJ, can compile either Java source of Java bytecode into machine code for many different processors. In the past, it could do some optimizations when compiling from Java source that it couldn't do when compiling from Java bytecode [6]. However, apparently, now it can do essentially the same set of optimizations for Java bytecode [5] [7] --- it just took more work --- and GCJ is switching to the Eclipse Java frontend for parsing Java, starting with GCC 4.3 [8]. In this case, the standard bytecode permits successful independent development of successive links in the toolchain.

The above notes are about traditional all-at-once or "ahead-of-time" compilers, rather than the piecemeal profile-directed-optimizing compilation used by good Smalltalk implementations, Self, and HotSpot. There are some differences: JIT compilers generally have to run faster than ahead-of-time compilers. I don't believe this negates my point; although parsing code takes some time, the compiler can maintain a "bytecode" parsed intermediate representation of the program without supporting it as an *input* format, and you can just as well cache an AST. (Also, bytecode-like linear quadruple and stack-machine intermediate representations in compilers are losing mindshare these days to other alternatives like CPS and GCC's tree-SSA.)

In general, optimizing an intermediate representation (such as a bytecode format) for high-speed interpretation tends to make it worse for dynamic machine-code generation, and vice versa. Good formats for dynamic machine-code generation tend to contain most or all of the information in the original source code; good formats for rapid interpretation erase as much of that information as possible.

Self pioneered the techniques of profile-based optimization, specialization, and type feedback that Bini refers to above, and which make it possible for current Java JIT compilers to do a reasonably good job, despite the dynamic nature of the Java language. Self's bytecode format was little more than an RPN tokenized version of the source code, with blocks separated into their own bytecode objects; quite different from the Smalltalk-80 bytecode format, in which control structures are generally already inlined, and with special bytecodes for instance variable access, local variable access, and common method names.

More recent efficient bytecode-VM designs like Lua's are register-based rather than stack-based, which again, makes interpretation faster, but dynamic compilation more difficult.

I don't know for sure why Microsoft's architects chose to make the CLR bytecode-based, but my guess is that it's a weak method of source-code obfuscation. Some of their customers would have balked at shipping source code to their "assemblies", the way they have to do if they're written in Perl or Erlang, and so they would have continued using Java or shipping blobs of x86 machine code instead.

Java Compilers That Don't Use Bytecode Produce Comparable Code

Bini's statement, "Java would never have had the performance it has now if it weren't for the bytecodes," is mistaken for another reason, other than merely that other programming languages. As mentioned above, GCJ can compile Java from source code to machine code without an intermediate bytecode step; so clearly the use of bytecode is not crucial to whatever performance GCJ-compiled code achieves. So what does it achieve?

In many cases, despite not using specialization or profile-directed feedback (which are easier to do in JIT compilers than in ahead-of-time compilers like GCJ), GCJ-compiled programs run around the same speed as JIT-compiled programs. The only systematic set of benchmarks I've found, from 2004, shows GCJ-compiled programs running a little less than half as fast as the

same program in the 1.4 or 1.5 IBM or Sun JDK. [11]

There is also a variety of anecdotal evidence from real applications; some of it shows programs running a little faster with GCJ, and some of it shows them running a little slower. I haven't been able to find anything comparing the performance of GCJ 4.x, which was supposed to have a lot of new optimizations, to anything else, or a comparison of Java 6 to any version of GCJ.

As far as I know, GCJ does not yet take advantage of type feedback from the compiled program, which could put it at a substantial performance disadvantage to a JIT compiler.

• "Performance measurements: Java and C++", by Jean-Marc Vanel, 2003

http://jmvanel.free.fr/perf/java-cpp.html

• "Compiling Java with GCJ", by Per Bothner, 2003-01-01, published in Linux Journal, says, "Truthfully, running a program compiled by GCJ is not always noticeably faster than running it on a JIT-based Java implementation; sometimes it even may be slower ... GCJ is often significantly faster than alternative JVMs, and it is getting faster as people improve it. ... Running the Kawa test suite using GCJ vs. JDK1.3.1, GCJ is about twice as fast..."

http://www.linuxjournal.com/article/4860

• "comparison between native gcj and bytecode", mailing list post by Erik Poupaert, 2003-01-05, to Prof. Laurie Hendren, posted to sablecc-list, saying, "There are reasons to believe that gcj already beats the JDK with regards to performance."

http://www.sable.mcgill.ca/listarchives/sablecc-list/msg00898.html

• "Performance comparison", mailing list post by Norman Hendrich, posted to java@gcc.gnu on 2002-07-29, showing his program running with GCJ at 72% of its JDK 1.3.1 speed and 47% of its JDK 1.4.0 speed

http://gcc.gnu.org/ml/java/2002-07/msg00121.html

• "Linux Number Crunching: Benchmarking Compilers and Languages for ia32", by Scott Robert Ladd, 2003-01-04, in which he benchmarked some computational astronomy code to measure floating-point performance; GCJ's best speeds were within 10% of the best 1.3 and 1.4 JIT JVM speeds (faster than the JIT VMs on one machine, slower on the other), but roughly three times slower than the Intel C++ and Fortran compilers.

http://web.archive.org/web/20040803034751/http://www.coyotegulch.com/reviews/almabench.html An incomplete updated version (missing the results!) is online at http://www.coyotegulch.com/reviews/number_crunching/

There are Valid Reasons to Use Bytecode, Just Not Speed

Bytecode can be considerably more compact than machine code, source code, or even gzipped source code, as a distribution format, and it uses dramatically less memory than abstract syntax trees, machine code, or source code. Bytecode interpreters are considerably more portable than native-code compiler backends. Finally, it's a lot easier to hack single-stepping and other debugging facilities into a

bytecode interpreter than to implement them for even one CPU/OS combination, let alone the whole range of CPU/OS combinations a particular language implementation might need to support.

These can all be valid reasons for choosing a bytecode-interpreter architecture for your language implementation rather than a native-code-compiler architecture, although they're not nearly as strong now as they were in the past (say, when Pascal was being developed, and there were literally dozens of incompatible CPU architectures in common use, some with several operating systems.) But performance is not a valid reason for this.

I wrote a kragen-tol post with more detail about this in March. [2]

Someone might argue that a machine-code compiler is inherently more complex than a bytecode interpreter, but I don't think that's necessarily true. The OCaml interpreter/compiler comparison above is one data point. As another, in 2003, I hacked together a machine-code "compiler" from parse trees for arithmetic expressions, using gcc to actually generate the machine code, in just a few hours [3]. But I don't have enough experience building machine-code-generating backends to say for sure.

Several of Ola Bini's Statements are Dubious

Bini ends his post, "So please, stop spreading this myth. It is NOT true and it has NEVER been true."

I think the myth he refers to is that bytecode interpreters are slow. But as shown both by his post and this note, pure bytecode interpreters are indeed considerably slower than native machine code. While native-code compilers that compile from bytecode can produce fast code, that's not because they use bytecode as an input format; that's because they're native-code compilers, often highly-tuned native-code compilers that use run-time profiling and type feedback information, and native code can run pretty fast. Even naively-generated native code rarely runs as slow as bytecode in a bytecode interpreter. (See section 4.5 of Urs Hölzle's dissertation [12]; the "non-inlining compiler" of Self-93 was 2600 lines of C++, one-fifth the size of the OCaml bytecode interpreter, and Hölzle argues that bytecode interpretation would be hard-pressed to do better.)

But Bini originally said, "[Rubinius] is byte code based. This means it's easier to handle performance." It's a pretty ambiguous statement (easier than what? what does "handle" mean?) but I think this note has adequately outlined the degree to which the obvious interpretations are false. While there are slower language implementation techniques than bytecode interpreters available, such as those used by bash or Tcl 7, they aren't in wide use.

Rubinius may well achieve excellent performance, or it may not, but its use of bytecode is not particularly relevant to that goal.

References

[0] "Can Bytecodes Perform Well?", by Ola Bini, 2007-09-24, on his blog

http://ola-bini.blogspot.com/2007/09/can-bytecodes-perform-well.html

[1] flippant comment on Ola Bini's blog post "Rubinius is Important", by Kragen Javier Sitaker

http://ola-bini.blogspot.com/2007/09/rubinius-is-important.html#comment-2690811425565194675

[2] "OCaml vs. SBCL, and various other interpreters", Kragen Sitaker, posted to the kragen-tol mailing list on 2007-03-12

http://lists.canonical.org/pipermail/kragen-tol/2007-March/000852.html

[3] "compiling Python arithmetic expressions to machine code", Kragen Sitaker, posted to the kragen-hacks mailing list on 2003-02-14

http://lists.canonical.org/pipermail/kragen-hacks/2003-February/000364.html

[4] "All you wanted to know about the HiPE compiler (but might have been afraid to ask)", by K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl, July 2003, 7 pp.

http://user.it.uu.se/~kostis/Papers/erlang03.pdf or http://www.erlang.se/workshop/2003/paper/p36-sagonas.pdf linked from http://www.erlang.se/publications/publications.shtml

[5] Mailing list post "Reconsidering gcjx", from Tom Tromey to the java@gcc.gnu and gcc@gcc.gnu mailing lists, posted 2006-01-26; in particular, see the part "Technical approach", which lists three optimizations previously available only with the .java front end.

http://gcc.gnu.org/ml/gcc/2006-01/msg01034.html

[6] Question 4.2 in the GCJ FAQ, "Can GCJ only handle source code?", most of which answer is written by Per Bothner; the page says it was last updated 2007-08-19.

http://gcc.gnu.org/java/faq.html#4_2

[7] Mailing list post on the thread "Reconsidering gcjx", from Tom Tromey to the java@gcc.gnu and gcc@gcc.gnu mailing lists, posted 2006-01-29, in which he talks about losing "a small optimization related to String "+" operations".

http://gcc.gnu.org/ml/gcc/2006-01/msg01095.html

[8] GCJ news item from 2007-01-08: "We've merged the gcj-eclipse branch to svn trunk... This new code will appear in GCC 4.3." Currently this is on the GCJ home page:

http://www.gnu.org/software/gcc/java/index.html But eventually it will probably move to the "Less Recent GCJ news" page: http://www.gnu.org/software/gcc/java/news.html

[9] "Timing Trials, or, the Trials of Timing: Experiments with Scripting and User-Interface Languages", by Brian W. Kernighan and Christopher J. Van Wyk, 1998

http://cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html

[10] Doug Bagley's Great Programming Language Shootout is now no longer available from Doug's site, but it can be found on the Internet Archive:

http://web.archive.org/web/20040805144133/www.bagley.org/~doug/shootout/index3.shtml

[11] "Performance Comparison of Java/.NET Runtimes (Oct 2004)", by Kazuyuki Shudo, 2005-11-20

http://www.shudo.net/jit/perf/

[12] "Adaptive Optimization for Self: Reconciling High

Performance with Exploratory Programming", by Urs Hölzle, August 1994, his doctoral dissertation.

[[??? XXX]] Linked from the Sun Self Research Papers page: http://research.sun.com/self/papers/papers.html

[13] [[Chambers]]

Notes from Hölzle's dissertation:

> Thus, we believe that type feedback is probably easier to add to a conventional batch-style compilation system. ... As mentioned above, static compilation has the advantage that the compiler has complete information since optimization starts after a complete program execution. ... On the other hand, a dynamic recompilation system has a significant advantage because it can dynamically adapt to changes in the program's behavior.

(section 5.6, p. 42, "Adding type feedback to a conventional system")

> The combination of SOAR's software and hardware features was very successful when compared with other Smalltalk implementations on CISC machines: with a 400 ns cycle time, SOAR ran as fast as the 70 ns microcoded Xerox Dorado workstation and about 25% faster than the Deutsch-Schiffman Smalltalk system running on a Sun-3 with a cycle time of about 200 ns. However, as we will see in Chapter 8, the optimization techniques used by the SELF compiler greatly reduces the performance benefit of special hardware support.

(section 2.5.4.2, p. 11; SOAR is a non-dynamic native-code compiler running on a customized version of the Berkeley RISC II processor)

Chambers's dissertation mentions some stuff about Dorado comparative performance:

> The definition of Smalltalk-80 specifies that source code methods are translated into byte codes, the machine instructions of a stack machine. Originally, Smalltalk-80 ran on Xerox Dorados implementing this instruction set in microcode [Deu83]. Subsequent software implementations of Smalltalk-80 on stock hardware supplied a virtual machine that interpreted these byte codes in software. Needless to say this interpretation was quite slow [Kra83].

Kra83 is:

> [Kra83] Glenn Krasner, editor. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, Reading, MA, 1983.

Topics

• Programming (p. 3658) (286 notes)

Compact namespace sharing

Kragen Javier Sitaker, 2016-07-25 (7 minutes)

Smalltalk bytecode is super compact, partly because it doesn't include literal constants, addresses, and method selectors in the bytecode itself; instead they're stored in a separate per-method vector and referred to by index. Java bytecode uses the same approach, but the result is that even though the bytecode itself is very compact, the .class files can easily be far from compact, more like huge and bloated.

You could imagine sharing, say, a single constant pool over an entire static program, just as some Smalltalks have shared a single object table over an entire dynamic program. Then, each constant would only be listed once in the pool. But it would be a large pool, so an arbitrary index into it could occupy many bits, such as 16 or even more. By contrast, Smalltalk bytecodes are only 8 bits, and typically only 3–5 of these are occupied in indexing into such a pool. This static locality of reference gives us potentially very considerable compression, but at the cost of duplication among the "dictionaries" used for this compression.

This is not the only possible compression mechanism for this information — you could also imagine things like move-to-front coding, and Smalltalk also has a fixed dictionary of special constants accessed by a separate opcode — but if that's the one we're using, we can consider how to optimize it better.

Function calls and returns are a kind of context switch: after the transfer of control, the variables and constants of interest to the new code are no longer the same as those of interest to the old code, much as in a switch between threads or processes. In 1973, Carl Hewitt, Peter Bishop, and Richard Steiger suggested conceptualizing each function's activation record as a separate "actor", like a process; "actors [or, if you will, virtual processors, activation frames, …] … it is impossible to determine whether a given object is "really" represented as…a hash table, a function, or a process," they said.

Context switches can have widely varying costs. Context switches that happen more often have a proportionally higher total cost, whatever other factors may be involved; and context switches that overwrite more state also have a proportionally higher cost. In the context of function calls and returns, a return that need only restore a saved PC and SP is inexpensive; a return that must restore %eip, %esp, %ebx, %ebp, %esi, and %edi is more expensive; and a return that must additionally restore floating-point registers, vector registers, and some kind of dynamic exception handling context, is more expensive still.

However, the alternative to overwriting more state is typically adding another level of indirection, and thus cost, to things that happen when you're not context switching. Bernie Greenberg's Multics Emacs paper explains his choice of a heavyweight context-switch mechanism for switching buffers in Multics Emacs as follows:

> The implementation of multiple buffers was viewed as a task of multiplexing the extant function of the editor over several buffers. The buffer being edited is defined by about two dozen Lisp variables of the basic editor, identifying the

current Editorline, its current (open/closed) state, the first and last Editorlines of the buffer, the list of marks, and so forth. Switching buffers (i.e., switching the attention of the editor, as the user sees it) need consist only of switching the values of all of these variables. Neither the interactive driver nor the redisplay need be cognizant of the existence of multiple buffers; the redisplay will simply find that a different "current Editorline" exists if buffers are switched between calls to it. What is more, the only functions in the basic editor that have to be aware of the existence of multiple buffers are those that deal with many buffers, switch them, etc. All other code simply references the buffer state variables, and operates upon the current buffer.

The function in the basic editor which implements the command that switches buffers does so by saving up the values of all of the relevant Lisp variables, that define the buffer, and placing a saved image (a list of their values) as a property of the Lisp symbol whose name is the current buffer's. The similarly saved list of the target buffer's is retrieved, and the contained values of the buffer state variables instated. A new buffer is created simply by replacing the "instatement" step with initialization of the state variables to default values for an empty buffer. Buffer destruction is accomplished simply by removing the saved state embedded as a property: all pointers to the buffer will vanish thereby, and the MacLisp garbage collector will take care of the rest.

The alternate approach to multiple buffers would have been to have the buffer state variables referenced indirectly through some pointer which is simply replaced to change buffers. This approach, in spite of not being feasible in Lisp, is less desirable than the current approach, for it distributes cost at variable reference time, not buffer-switching time, and the former is much more common.

This is also the reason shallow binding is usually preferred to deep binding in implementing dynamically-scoped languages: shallow binding makes function calls expensive, but variable access cheap.

So suppose that, when we enter a function, we load a whole passel of values into the registers of the machine, virtual or otherwise; there's a passel pointer before the beginning of the function code, and it is used to initialize the registers. Some of these registers may contain "constants" that we want to be able to use, others contain initial values for local variables, and others still may be instance variables of an object, which will need to be saved back to the object when the function is done.

The wide cache memory buses featured on modern high-performance processors can transfer 64 bytes at a time; wide 128-bit, 256-bit, and 512-bit (16-, 32-, and 64-byte) SIMD registers can be loaded in a single cache access (right?). Hardware designed slightly differently could

fixed levels of hierarchy
wide memory
message queues

Topics

Turning off the power supply for every sample to reduce noise

Kragen Javier Sitaker, 2018-06-18 (2 minutes)

Switching power supplies generate a lot of electrical noise. The AVR has the option of temporarily halting everything on the chip except the ADC during an analog-to-digital conversion in order to remove sources of noise. Could you shut off a switching power supply during analog-to-digital conversions in order to get further precision?

You'd need to have a big enough capacitor to keep the processor going, plus perhaps adequate voltage regulation.

If you were running an AVR ATMega328 on 5V, well, its "typical" power consumption is listed as 1.2 mA when idle at 8 MHz on 5V; we can safely extrapolate to 3 mA at 20 MHz. A 10-bit conversion takes it 260 μs. Let's say it's safe to let the voltage droop from 5.45 V to 4.55 V during that time (hopefully this 20% variation in input voltage during the conversion won't hork the ADC). This requires the capacitor to be at least 870 nanofarads, which is eminently feasible, even in a low-L MLCC.

If this would cause problems for the ADC, you could charge the capacitor up to a higher voltage and use a linear voltage regulator (perhaps an LDO for efficiency) in between the capacitor and the microcontroller. Then the ADC wouldn't see a changing reference voltage.

So it this approach seems workable. It requires that you not be spending all your time doing conversions, though, because otherwise your switcher is going to be generating even more noise that will keep ringing through the system during conversions.

Oh hey, I think it can actually do a 10-bit conversion in more like 67 μs at a higher clock rate. This reduces the required capacitor to more like 220 nF.

Topics

Really simple lab power supply

Kragen Javier Sitaker, 2019-12-10 (7 minutes)

In Lab power supply (p. 2421) and Bench trash power supply (p. 1457) I wrote about adjustable power supplies for electronic benchwork. But I think I was overcomplicating the problem.

Basic design: a PWM-controlled buck converter

An AVR is sufficient for the control problem, though see below about STM32s. You can hook a couple of potentiometers up to it easily enough as a user interface. At some point you will want a screen to see how much current is being drawn, but that's a rathole we can avoid at first by way of using a USART.

To limit the output voltage you can rely on an ATX power supply's high-current 12-volt output regulation; you can use a buck converter consisting of a P-channel MOSFET, a freewheel schottky up from ground, and a small inductor. Such a device can have its voltage controlled, even open-loop, with a PWM or PDM signal from the AVR. Any random NPN BJT will work as a gate pulldown driver unless you use a monster MOSFET; switching frequencies in the tens to hundreds of kHz are not demanding either on the gate capacitance or the microcontroller. (To drain 100 nC of Qg in 2 microseconds at 5 volts, you need a 100-ohm pullup, so a 47-ohm pullup will work even for monster MOSFETs; using such a high-current resistor would start to limit your options for the BJT.) Also most of the P-channel MOSFETs I've seen would work fine.

How much inductance do you need? I don't know. Not very much I think. Enough to avoid discontinuous conduction mode, I guess. If you use a too-big inductor you'll be fine except that you'll also have to use a too-big capacitor to prevent voltage spikes when the load comes unplugged.

A back-biased beefy rectifier from the output to the input will reduce the risk that input-supply crowbarring will nuke the circuit if its output is feeding voltage back into it (for example, because you were charging a battery with it).

Current sensing with a shunt

Bench power supplies need current limiting, both because otherwise it's too easy to get smoke, and because sometimes you want to test things with a current source. Typically an ATX power supply will do some limiting, but not in a pleasant way. Since you want to measure the output current anyway, you might as well do the current limiting in software.

There are lots of ways to measure the output current but probably the easiest is to use a precise sense resistor in series with the inductor to measure the inductor current. High-impedance voltage dividers from the two ends of the sense resistors connect them to the microcontroller's pins, and the difference between the two measurements gives us our current measurement.

To be concrete about current sensing, say you're using an AVR

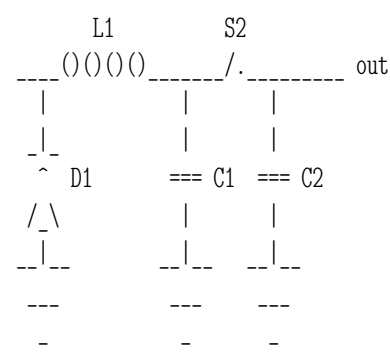
with its 1.1V bandgap reference (1.0V to be safe) and its 10-bit ADC, and you want to measure voltages up to, say, 13V, and currents down to 10 mA, which is not a very ambitious milliammeter but probably adequate for a quick-and-dirty power supply. If your voltage divider is made out of a 100-kilohm resistor and a 6.8-kilohm resistor, the full-scale 13V is divided down to 844 mV, at most 865 counts on the ADC and at least (if the bandgap reference is 1.2V) 720 counts; one count is thus divided down from 18 mV, so you need at least a 1.8-ohm sense resistor, and probably a 2.2-ohm sense resistor in practice. Using such a big sense resistor is an annoying limitation on how much current the power supply could source at low voltages.

STM32 possibilities

Although an AVR would be adequate, an STM32 (see Notes on the STM32 microcontroller family (p. 3176)) would be dramatically better; a 12-bit ADC means you could use a four times smaller sense resistor (0.47 ohms), and the higher 1Msps sampling rate means you can react more quickly to load changes and have more confidence in the current measurement, which in turn means you can use higher switching frequencies and smaller inductors and capacitors, although at some point you start needing an active pullup for the MOSFET gate, maybe a gate driver chip.

Current sensing without a shunt

Instead of using a shunt to sense the current you're charging the output cap with, you could use two caps in parallel across the output with a new switch between them:



I'm not sure what to use for this switch, other than some kind of transistor, but the idea is that you leave it closed almost all the time, but occasionally you open it to find out how much current is flowing. The voltage across C2 will start falling, but C2 is in parallel with the input capacitance of the load, and you don't know what that capacitance is. More useful is that the voltage across C1 will start rising, and the speed with which it's rising tells you the current that's running through L1 at that moment. If C1 is much smaller than C2, say by a factor of 40, the rise in voltage will be much faster than the fall on C2; if you leave the switch open long enough for C1's voltage to rise by 25%, the voltage across C2 will have fallen by 0.6%; and once you close the switch again the droop will be instantly corrected. (Oof, might want a little inductance or resistance there to keep the switch from exploding from the singularity.)

This is handy because, for a good current measure, you'd want C1 to have a pretty precise and stable capacitance, and those are a lot

easier to find in lower capacitance values. Electrolytic and ferroelectric-ceramic capacitors have very imprecise capacitances.

This design can use one ADC pin instead of two, but I think it requires two digital output pins instead of one.

Basic BoM

So, with the shunt design, that works out to a microcontroller, two pots, seven resistors, a random inductor, a random capacitor, a schottky, the protection rectifier, a P-MOSFET, and an NPN BJT, 15 non-microcontroller components. And if you fuck up the firmware the ATX power supply will probably save you but maybe not. All of these except the microcontroller can be easily scavenged.

You can hang a bunch of these puppies off a single ATX power supply if they don't overload it. You might be able to hang a bunch of them off a single STM32; you need a couple of ADC input pins per power-supply line.

Topics

• Electronics (p. 3430) (138 notes)
• Independence (p. 3520) (63 notes)
• AVR microcontrollers (p. 3337) (20 notes)
• STM32 microcontrollers (p. 3733) (7 notes)
• Power supplies (p. 3643) (3 notes)

Secure, self-describing, self-delimiting serialization for Python

Kragen Javier Sitaker, 2017-04-11 (8 minutes)

I find myself somewhat unexpectedly desiring a new serialization format for Python data structures. This is unexpected because Python's standard library already includes several serialization formats: pickle, marshal, json, and xmlrpclib (not counting xdrlib, ConfigParser, and struct), and other formats such as bencode are also widely used in Python.

(bencode might actually be the right solution, but I can't look at the internet to see right now.)

I'm defining a network protocol for a program I'm writing, and one of the things I want to do in this protocol is to pass Python data structures over the wire. I'm not concerned with being able to serialize arbitrary class instances — I'd be satisfied with built-in data types — but I don't want a lot of hassle.

The serialization needs to be:

- Secure — all of pickle, marshal, and xmlrpclib have warnings against using them on data from potentially malicious sources, leaving only json of the standard modules.
- Self-describing — it must be possible to deserialize a serialized value without referring to some external schema information.
- Self-delimiting — it must be possible to deserialize a serialized value from a byte stream and then continue using the byte stream, for example to deserialize another value that follows it. This rules out json; I think it rules out bencode but I'm not sure.
- Dependency-free — its implementation code must be self-contained.

Ideally, the serialization would also be:

- Simple to implement, improving its chances of being secure.
- Fairly transparent, supporting the full range of commonly-used built-in Python data types, preserving the tuple-list distinction, the int-float distinction, the bytes-unicode distinction, non-string dict keys, Booleans, None, and maybe even sets.
- Hashable, in the sense that a given Python value is representable by only a single possible byte stream.
- Capable of serializing class instances, optionally, with an explicit namespace of supported classes.
- Devoid of arbitrary format limitations, such as limiting strings to 2^{32} bytes.
- Pure ASCII except when it's encoding strings that aren't.
- Reasonably efficient.
- Supportable on other programming languages, including old versions of Python.

But it does not need to be:

• Easy to type by hand. (So length fields are okay.)
• Super efficient.
• Capable of serializing functions and/or closures and/or generators.
• Capable of serializing exotic or very stateful Python types like Ellipsis, xranges, code objects, type objects, memoryviews, file objects, and complex numbers.
• Capable of representing and restoring sharing or circular data structures.
• Super easy to read.

Length-free design

```
val ::= bytes | unicode | tuple | list | dict | boolean | none | int | float
bytes ::= '"' stringcontents
unicode ::= 'u"' stringcontents
stringcontents ::= '"' | stringbyte stringcontents
stringbyte ::= [^"\] | '\"' | '\\'
tuple ::= '(' tuplecontents
tuplecontents ::= ')' | val tuplecontents
list ::= '[' listcontents
listcontents ::= ']' | val listcontents
dict ::= "{" dictcontents
dictcontents ::= "}" | val val dictcontents
boolean ::= "T" | "F"
none ::= "N"
int ::= sign digits " "
digits ::= [0-9] | [0-9] digits
float ::= sign digits "e" sign digits " "
sign ::= "-" |
```

The pairs of vals in a dict are key-value pairs. Dict keys must appear in sorted order by the lexicographical ordering of their serializations. Unicode strings are represented in UTF-8. The only whitespace allowed is that within strings and that following numbers. The only case where the next byte is not sufficient to dispatch to the appropriate routine is the int/float dichotomy. Yes, floats are expressed without decimal points, so "31416e-4 " is a reasonable representation for an approximation of π.

In practice this should be slightly more compact than bencode except for large binary strings (which it inflates on average by almost 1% but by 100% in the worst case), and much more readable and writable, but considerably slower and more error-prone.

As an example, `{"announce-list": [["foo"], ["bar"]], "info": {"files": [{"length": 4541, "path": "baz", "safe": False}], (): (1, 1.0)}}` would be represented as `{"announce-list"[["foo"]["bar"]]"info"{"files"[{"length"4541 "path""baz""safe"F}]()(1 1e0 )}}`.

Length-prefixed design

```
val ::= bytelength body
bytelength ::= digits
digits ::= [0-9] | [0-9] digits
body ::= bytes | unicode | tuple | list | dict | boolean | none | int | float
bytes ::= 'H' data
```

```
unicode ::= 'u' data
data ::= "" | [\x00-\xff] data
tuple ::= '(' vals
vals ::= "" | val vals
list ::= '[' vals
dict ::= "{" pairs
pairs ::= "" | val val pairs
boolean ::= "T" | "F"
none ::= "N"
int ::= " " sign digits
float ::= "." sign digits "e" sign digits
sign ::= "-" |
```

Here every `val` begins with a decimal representation of the number of bytes in the body of its serialization, not counting the initial type byte.

As an example, `{"announce-list": [["foo"], ["bar"]], "info": {"files": [{"length": 4541, "path": "baz", "safe": False}], (): (1, 1.0)}}`, the same example value from before, would be represented as `100{13Hannounce-list14[6[3Hfoo6[3Hbar4Hinfo58{5Hfiles36[33{6Hlength4 45414Hpath3Hbaz4Hsafe1F0(8(1 13.1e0`. This is about 10% bigger than the length-free design, and a hell of a lot harder to type or read, especially the parts that seem to say "45414H" and "13.1e0", but can be navigated efficiently and can support large chunks of binary data.

Stack-based design

Pickle deserializes by interpreting stack-based bytecode similar to Python bytecode (which leads one to wonder why they didn't just use Python bytecode). The pickle-version-0 encoding of the sample datum `{"announce-list": [["foo"], ["bar"]], "info": {"files": [{"length": 4541, "path": "baz", "safe": False}], (): (1, 1.0)}}` is the following 188 bytes:

```
(dp0
S'info'
p1
(dp2
S'files'
p3
(lp4
(dp5
S'path'
p6
S'baz'
p7
sS'length'
p8
I4541
sS'safe'
p9
I00
sas(t(I1
F1.0
tp10
ssS'announce-list'
```

```
p11
(lp12
(lp13
S'foo'
p14
aa(lp15
S'bar'
p16
aas.
```

Here:

- `a` appends an item to a list,
- `s` appends a name-value pair to a dict,
- `(` pushes a PostScript-style mark,
- `t` forms a tuple from the items down to the PostScript-style mark,
- `l` forms a list (down to the mark),
- `d` forms a dict,
- `S`, `I`, and `F` encode strings, ints, and floats (up to the end of the line),
- `p` names the item on top of the stack so it can be referred to later if there are more references to it, and
- `.` ends the pickle.

Now, I have no idea why pickle incrementally appends stuff to lists and dicts as it builds them. `pickle.loads("(S'foo'\nS'bar'\nS'baz'\nI37\nd.")` does return `{'foo': 'bar', 'baz': 37}` as you would expect, and changing the `d` to an `l` generates the corresponding list. So I don't know why `a` and `s` exist.

If you wanted to take this approach to make your serialization and deserialization as little code as possible, you could use this approach:

```
op ::= digits intop | '(' | '}' | ']' | ')' | 'T' | 'F' | 'N' | LF
intop ::= ' ' | '-' | 'H' data | 'u' data | 'F' data
digits ::= [0-9] | [0-9] digits
data ::= "" | [\x00-\xff] data
```

Here the '}', ']', and ')' ops play the role of 'd', 'l', and 't' in pickle; LF plays the role of '.'; ' ' specifies that the preceding digits just represent an integer (and '-' is the same, but negates it); 'H' and 'u' specify that the preceding digits are a count of following bytes for a byte string or UTF-8-encoded Unicode string; and 'F' represents a floating-point number in some way that I'm not specifying right now.

As an example, `{"announce-list": [["foo"], ["bar"]], "info": {"files": [{"length": 4541, "path": "baz", "safe": False}], (): (1, 1.0)}}` would be represented as `(13Hannounce-list((3Hfoo](3Hbar]]4Hinfo[5Hfiles((6Hlength4541 4Hpath3Hbaz4HsafeF}]()(1 1F1)}}\n`, which is one byte longer than the length-free design, but retains most of the efficiency advantage of the length-prefixed design. I'm not sure there's a meaningful difference, really...

Topics

- Programming (p. 3658) (286 notes)

House scrubber

Kragen Javier Sitaker, 2016-09-06 (updated 2019-11-25) (13 minutes)

So I was just thinking about how my city is annoyingly polluted. In the absence of successful collective action to reduce the sulfur content of diesel fuels, and to require proper maintenance of motors, it seems like it would be possible at least to handle the problem at a more local level. Like a per-apartment level.

You could seal up your apartment or house, but that has a couple of different problems. The biggest one is that when the amount of CO_2 in the air rises to about 1% you will start to feel that it's a bit hard to breathe, and when it reaches 10%, you will die — even though there is still plenty of oxygen left (barring a Biosphere-II-type surprise).

So this kind of thing is a problem that has been dealt with in a lot of different contexts over the years, including scuba diving, nuclear submarines, the Space Shuttle, the ISS, firefighting, mine rescue, and carbon capture for power-plant flue gas.

The scuba-diving approach, which is also sometimes used in submarines, is to pass the gas over a base that reacts with the CO_2; they use hydroxides of calcium, magnesium, sodium, and lithium, which transform from hydroxides into carbonates. The trouble with these is that you have to keep replacing the hydroxide, which is kind of undesirable. You can buy calcium hydroxide at the hardware store here for use as paint (whitewash), for about US$3 per kilogram, although it's mixed with an unspecified quantity of calcium carbonate, and some other random shit. A kilo of CaOH sucks up about 1.2 kilos of carbon dioxide. So you'd be using about five kilos a week.

On the plus side, it's super low tech. You can just paint it on the wall and let it suck up the CO_2 from there over the next few days. It'll also suck it up just sitting in the bucket, but that will take many years. (Which suggests that you could buy, say, ten tons of it and just let it sit in a crate.)

You can regenerate calcium carbonate back into calcium hydroxide by roasting it, but you have to get it up to 850° or so, which is pretty hot. Magnesium carbonate is more promising: it starts to decompose back into magnesium oxide at just 350°, and if you don't heat it past 700° or so, it remains deeply eager to suck the CO_2 back out of your air. So you could imagine some kind of solar kiln on your roof to bake out the carbon dioxide you'd exhaled over the last day or two. But you're still having to deal with powders, which probably means batch rather than continuous-flow. And they're caustic powders, so if you trip carrying a bucket of this shit, you're going to the hospital. (Magnesium oxide gets a lot more stable if you heat it further, apparently.)

(The ISS uses a similar system, but uses zeolite molecular sieves rather than alkali.)

The US nuclear submarine fleet chose a different option, one used for flue-gas treatment: they use an aqueous solution of ethanolamine. Ethanolamine, like magnesium hydroxide or calcium hydroxide, is eager to suck up CO_2 from your air; unlike them, it does so while

remaining happily liquid in an aqueous solution, so you can pump it around. Even better, you can persuade it to give up the CO_2 by heating it to merely 120° or so, which is a much easier thing to do. On the downside, it's inflammable and toxic. Its cousin diethanolamine works too, and is much less inflammable and toxic. Also on the downside, to get it to absorb the CO_2, you need to compress the gas to several atmospheres, at least 5 but ideally more like 200 atmospheres.

But that seems like the kind of thing you could reasonably have in your house. I mean, your refrigerator is already compressing its R-134a refrigerant to about ten atmospheres, which I think is enough to persuade diethanolamine to take up the CO_2. So you could very reasonably install an "air conditioner" with a similar-size compressor motor, compressing air. (You might be able to allow the air to re-expand to atmospheric pressure through a series of turbines alternating with heat exchangers in order to offset some of the energy use of the compressor.) It would need to pump through about 100 m^3 of air per day per person in order to suck up 1kg of CO_2 per day at a 1% concentration. That's a bit more than a liter per second, or 115mℓ per second at the cooled compression output.

I don't know how much the liquid flow needs to be. Presumably several milliliters per second. Most of the temperature difference in the liquid can be managed with a countercurrent heat exchanger, so that it doesn't represent an ongoing energy waste, in particular the part where the hot liquid coming back into your house has to shed heat into your space to get back down to room temperature.

We can estimate the energy usage of this contraption. Isothermically compressing 1.15ℓ down to 115mℓ involves squeezing, say, 9cm out of a 10cm cylinder with 115cm^2 surface area, against a pressure that rises linearly from 0 to 10 bar; that's 518J, so the compressor needs 518W. Some of this (let's say 90%) can in theory be provided by decompressing the sweetened air through turbines, so you might need 52W. And if you're heating up, say, 10mℓ of something water-like from 40° to 120° each second, that's 80 cal/s, or 330 W, of which you can probably economize 90% with a countercurrent heat exchanger, getting you down to 33 W. And you probably need another 10 W or so for the water pump. Total, about 95W dissipated, out of 950W flowing hither and thither.

Now, of this 95W, some fraction is presumably the actual unavoidable Carnot loss from pumping CO_2 from a (potentially) low-CO_2 environment to a (potentially) high-CO_2 environment (although in fact we're proposing here to pump it from your 10000ppm CO_2 sealed neurotic bunker to the 400ppm outside world). But that is probably a small amount compared to the 10% losses I've assumed in compressing-decompressing and heating-cooling cycles.

(We could apply this same approach to removing CO_2 from the air at scale. Note, however, that this 95W is about the same wattage as the human being that hypothetically exhaled that kilogram per day of CO_2, so you end up using about the same energy to get the CO_2 out of the air with this approach that you originally got out of burning it — and that's after the CO_2 has reached 10 000 ppm! So, while this is a viable approach to carbon dioxide removal in a world where we have a great deal more energy available than we ever used in fossil fuels, it is too expensive at the moment.)

You may be able to use photosynthesis, as in Biosphere II, to remove some of the CO_2 from the atmosphere. However, this will require significant indoor acreage; at normal CO_2 concentrations, you need about as much sunlight to turn your CO_2 back into carbohydrates as it took to turn it into the carbohydrates you ate. Beema bamboo, a thick-walled variety, supposedly produces 50 tons of dry biomass per acre per year (112 tonnes/ha/y), yielding 4000 kcal/kg. That's about 6.6 W/m², which means that your 100-watt body would need 15m² of bamboo to consume the CO_2 it emits. Other high-yielding plants are similar — sugarcane commercially reaches 70 dry tonnes per hectare per year and experimentally has reached 98 dry tonnes/ha/y, which is 5.2 W/m². (Hardier plants like switchgrass and Miscanthus, which thrive without the subtropical conditions that beema and sugarcane demand, are down around 15 to 40 dry tonnes/ha/y).

Most crops, apparently, increase production substantially (varying from 25% to 200%) CO_2-supplemented in a greenhouse, but I don't know if this applies to super-high-biomass-productivity crops like Beema bamboo and sugarcane. 1000 ppm is a common supplementation level. 10 000 ppm is high enough that it will likely cause problems for plants; Alberta's official advice to greenhouse gardeners is to keep below 4500 ppm for the sake of plants, or 5000 ppm for the sake of humans; other sources suggest that over 1200ppm some crops start to "show undesirable growth responses". You'd only reach that level if the plants fall behind your breathing, and since the curve levels off then, at that point the overall system could go unstable — you'd want to keep the concentration in the range where the plants can increase their photosynthesis to respond to elevated CO_2 levels.

One great difficulty with this approach, however, is that if you bring 15m² or 30m² of sunlight into your house to grow plants, you need to have a way to shed the kilowatts of heat you're adding — roughly two orders of magnitude more than those dissipated by the diethanolamine system, which hopefully you can also manage to dissipate outside the house, at least in the summer.

(You may in fact be able to get by with 15m² of *sunlight* concentrated onto a significantly smaller area of *plants*.)

One approach to solving this problem is to keep the plants in a separately-insulated greenhouse, which can exchange air with your dwelling space via a heat exchanger. If you were to follow Alberta's recomendations, you could perhaps send air to the greenhouse at 4000 ppm CO_2 and get it back at 1000 ppm, thus dropping 3000 ppm; the 1kg/day/person mentioned above then is about 4 liters per second or 8 cfm. You can get 10 cfm out of a 75mm-diameter 4-watt 3000 RPM fan that you might mount on your computer's CPU. Another benefit of keeping the greenhouse at a different temperature is that CO_2-supplemented plants may prefer higher temperatures, like 35°, which is very uncomfortable for people; but you could keep it at those higher temperatures without bothering yourself.

If we figure that a city block is normally 100m×100m, and (to play it safe) that we need 30m² of sunlight to fix a person's CO_2, a city block can house some 330 people.

Once we've taken care of the CO_2, there are other things in the air we need to take care of. A variety of artificial objects outgas

pollutants like formaldehyde; in the absence of ventilation to vent these to the outside world, they could accumulate to potentially dangerous levels. One approach is to work hard not to allow things into your house that are likely to outgas pollutants, but that's very difficult.

The approach taken in the ISS is to have a separate trace contaminant control system, which is just an activated-carbon air filter. The Mir space station thermally regenerated its activated carbon with a hot non-oxidizing gas; the ISS uses disposable activated carbon instead, plus an 0.5%-palladium-on-alumina catalytic converter to burn up methane and other things that get past the activated carbon, followed by LiOH sweetening.

Their activated carbon is impregnated with phosphoric acid in order to soak up ammonia off-gassed by human metabolic processes.

What about other carbonates or bicarbonates --- might they be easier to regenerate? They are more difficult, as it turns out. Sodium bicarbonate dehydrates to sodium carbonate at between 50° and 270°, at which temperature the decomposition is complete; Wikipedia says that it's fast at 200°; above 850° it releases carbon dioxide. Potassium carbonate (pearl ash or salt of wormwood) doesn't melt until 891°, rubidium carbonate doesn't decompose until 900°, and lithium carbonate doesn't decompose until 1300°.

See also Notes on a possible household air filter (p. 1961).

Topics

Low-cost green thread locks

Kragen Javier Sitaker, 2016-09-06 (2 minutes)

(This idea is probably bad because we can't eliminate the cost of lock acquisition completely; we still need to conditionally block if the lock is acquired.)

Zero-overhead exception handling uses a map of executable code regions to analyze the stack trace at the time an exception is thrown to discover where to invoke the exception handler. This way, a `try { }` statement, or in C++ the code to destruct a stack object during exception unwinding, has zero run-time overhead in the normal case; it just makes the executable a little bigger. Throwin an exception is slower than using a `setjmp()`/`longjmp()` style of exception handling, but this is still a net win, because exceptions are very rare compared to entering and leaving contexts that add tasks to carry out during stack unwinding.

Analogously, acquiring a lock is a very common operation in threads-and-locks programming, and typically a very expensive one. But we only care what locks you hold when another thread might contend them. In a "green thread" or "fiber" environment, only one thread is actually running at a time, and context switches between threads are relatively uncommon.

In a language like Java, locks are normally managed by entering and leaving `synchronized` statements and methods. These can be discovered by walking the stack in the same way that exception handlers can.

This suggests the following implementation technique for locks in a green-threads environment: don't maintain the runtime state of which locks are held by the current thread at all, although of course you still need to test to see if a lock is held when entering a context that requires it. Instead, when the thread yields to the scheduler, the scheduler walks its stack to reconstruct the set of locks it holds. Then it marks those locks as "held" before choosing which next thread to run.

This slows down context switching, but speeds up lock acquisition, making it affordable to acquire more locks.

Topics

• Programming (p. 3658) (286 notes)
• Facepalm (p. 3450) (24 notes)
• Concurrency (p. 3386) (9 notes)

Raid zim

Kragen Javier Sitaker, 2019-01-17 (updated 2019-02-08) (1 minute)

```
# I couldn't fit this 37GB Wikipedia dump (English with no images)

# <magnet:?xt=urn:btih:267d148412ad50361e0c0384c905fabe35e8539a&dn=wikipedia%5Fen%5Fall%5Fnopic%5F2018-09.zim>
# onto my flash disk because VFAT doesn't support large files.  So, I
# thought, I'd stick it on an ext4 filesystem in a disk image file on
# the VFAT, thus circumventing all of VFAT's restrictions.  Not so
# fast  -  that file can't be 4GB or over either.  So what to do?  We
# can take advantage of Linux software RAID:

set -e
DANGER() {
        :
}

: ${usb?} ${user?} # ensure the $usb and $user variables are set
# This first part takes about 40 minutes because VFAT doesn't support
# sparse files either:
DANGER time truncate -s 2g "$usb"/wikipedia-zim.img.{1..20}

for i in {1..20}; do sudo losetup "/dev/loop$i" "$usb"/wikipedia-zim.img."$i"; done

DANGER sudo time mdadm --create /dev/md0 --level 5 --force --raid-devices=20 /dev/loop{1..20}
# 0.00user 0.02system 0:33.79elapsed 0%CPU (0avgtext+0avgdata 2592maxresident)k
# This seems to work for subsequent mounting but never the first time:
sudo time mdadm --assemble /dev/md0 /dev/loop{1..20}
DANGER time sudo mkfs -t ext4 /dev/md0
# real    2m24.296s
DANGER sudo tune2fs -r 0 /dev/md0   # no root-reserved blocks; frees up about 2GB
sudo mount /dev/md0 /wikipedia-zim/
df -h
# Filesystem                Size  Used Avail Use% Mounted on
# /dev/md0                   38G   48M   38G   1% /wikipedia-zim
# However, Transmission reports 39.9 GB.
DANGER sudo chown "$user"."$user" /wikipedia-zim/

remove_raid() {
    sudo time umount /wikipedia-zim/
    sudo mdadm --stop /dev/md0
    sudo losetup -D
    umount "$usb"
}
```

Topics

- Archival (p. 3322) (34 notes)
- Filesystems (p. 3455) (8 notes)
- Unix (p. 3765) (7 notes)
- Error-correcting codes (p. 3423) (4 notes)
- Raid

Caustic business card

Kragen Javier Sitaker, 2019-04-08 (3 minutes)

Suppose my selective electro-etching process works to produce caustic surfaces. Can I make myself a business card?

I'm not sure how big a business card is. If it's 50 mm × 30 mm and has a 50 × 30 mm reflection, presumably the focal distance needs to be short enough that the sun's ½° size gives adequate resolution to the letters. Minimally we need 5×8 and 5 lines of 20 letters, amounting to 4000 pixels in the caustic. The reflection could potentially be larger than the card itself, but if that's going to be significant, it needs to be curved. A spherical bead would be cool, but let's start by thinking about flat metal plates.

That's about 2.7 pixels per square millimeter (of the image), or about 0.6 mm per pixel. Half a degree is about 9 milliradians, so the projection distance can't be more than about 70 mm to get that resolution. This is feasible but uncomfortably tight.

That is: the sunlight, with ½° = 9 mrad divergence, comes in and illuminates the overall-flat shiny mirror-polished metallic 50 mm × 30 mm surface, producing a 50 mm × 30 mm reflection on a surface in shadow in the same direction as the sun, 70 mm away. Because of the divergence, there's a fuzzy border around the reflection with a radius of 0.3 mm. Tiny variations from flatness across the mirror result in major variations in brightness across the reflection, in particular bright spots from caustics, which also have a fuzziness radius of 0.3 mm. These bright spots spell out my name, email address, and so on. They can reasonably be spaced some 0.6 mm apart, giving a resolution of some 83×50 "pixels", which is 16 5×8 letters on each of 6 lines, a total of 96 character positions. This can be improved somewhat with subpixel positioning and proportional fonts but not a whole lot.

The bright spots reach their maximum brightness when the corresponding 0.6-mm-diameter spot on the mirror is concave parabolic with a spherical radius of curvature of 140 mm. This means that the center is depressed from the edge by $140\text{ mm} - \sqrt{((140\text{ mm})^2 - (0.3\text{ mm})^2)} = 320\text{ nm}$, about half a wavelength of light.

This is small enough that the geometrical optics approximation may not apply and we may have to consider diffraction. (We can perhaps improve the situation there a bit with a blue coating.) The usual Airy limit is $\sin\theta = 1.220\lambda/D$ for a circular aperture; in this case our λ is about 555 nm and our D is about 600 μm, so this works out to $\sin\theta = 0.0011$, which is to say that our first diffraction null is about 1.1 milliradians in radius, 2.2 milliradians in diameter. This is substantially smaller than the 9 milliradians of the sun's disk, so the diffraction effect is significant, but not dominant; we're in good shape.

Topics

• Physics (p. 3632) (119 notes)

Kogluktualuk: an operating system based on caching coarse-grained deterministic computations

Kragen Javier Sitaker, 2016-07-23 (21 minutes)

I'd like to know what you think about this draft. It should take about ten or twenty minutes to read.

In the spirit of Hamming's admonition to look for attacks on the most important problems in your field, I propose a project.

I've been thinking about a kind of Unixy system for distributed, high-performance, fault-tolerant, incremental, reproducible computing, scalable to exabyte datasets. It consists of a transactional deterministic virtual machine, a content-addressed blob store, and a cache management system.

Tentatively I've been calling it Kogluktualuk.

Overview

The basic idea is to generalize build systems like the NixOS build system or apenwarr/Bernstein `redo` to use them for general-purpose computing; given some kind of specification of a deterministic computation that includes all of its input data, you can re-execute the transaction later on and get bit-identical results, as long as you either have stored or can recompute all of that input data.

(I say "transaction", despite the false implication that there is an underlying mutable data store being mutated by the transactions. Kogluktualuk transactions do not mutate state, and they do not have any access to mutable state; they merely generate output from input.)

For example, if "5B2uheZVEd5u61Nc5" is the secure hash of some specification of a deterministic transaction — maybe a command line like

```
in_namespace "4dkcSzVpvmtcgCcUo" do /bin/sort /data/july-trades
```

then perhaps execute("5B2uheZVEd5u61Nc5") will produce an output dataset with the hash "5GWFsognLrfv9tqin"; and if so, then it will *always* produce "5GWFsognLrfv9tqin", and never some other output dataset.

If each blob of input data is specified either by a secure hash, or by a secure hash of a script (and input data) that tells how to compute it, you can be sure that it will be bit-identical if you re-execute the transaction, and so the output data can be cached. This means that if the same computed data is required twice, it can be opportunistically cached, and it's safe to store the cache on inexpensive, unreliable storage; and it is possible to detect and correct incorrect transactions by reproducing the transaction, for example on a different machine.

Running a transaction that has more than one input blob that is identified by the transaction to produce it (rather than directly by its

content hash) offers opportunities for deterministic concurrency and distribution.

For example, in the above transaction, perhaps the namespace "4dkcSzVpvmtcgCcUo" includes input specifications something like the following:

```
/data/july-trades blob 3yCXJMme1nn3mthVP
/bin/sort         do /bin/ejs /src/sort.js
/bin/ejs          do /bin/jsinterp /src/ejs.js /src/ejs.js
/bin/jsinterp     do /bin/cc /src/jsinterp.c
/bin/cc           do /bin/slowcc /src/cc.c
/bin/slowcc       do /bin/as /src/slowcc.s
/bin/as           blob 2r3g5UzEREHrbCyeg
```

This allows the fetching of `/data/july-trades` from a remote blob store to happen in parallel with, if necessary, the recompilation of /bin/sort. Normally, of course, there will be a cached version of /bin/sort; it won't be necessary to bootstrap the entire universe from source code to sort a data blob.

Also, though, if /data/july-trades is large, /bin/sort can split it into a number of smaller blobs, then map a name to the sorted version of each smaller blob:

```
/tmp/xaa.sorted  do /bin/sort /tmp/xaa
/tmp/xab.sorted  do /bin/sort /tmp/xab
/tmp/xac.sorted  do /bin/sort /tmp/xac
```

Then, it can merge those sorted chunk blobs to generate its output blob. These separate sorting operations can run in parallel, and potentially on different machines, in isolated namespaces of their own; this introduces no nondeterminism into the computation of the transaction, because accessing the sorted smaller blob invisibly blocks until it is complete. (Or, at least, enough of it is available to satisfy the read request.)

Once the transaction is complete, the entire modified namespace can evaporate, leaving only a precipitated output blob.

It's desirable both to generate the dependency graphs of the computation dynamically, as in the above example, and to statically audit that all necessary data for a transaction is present. We can obtain both of these properties at once with this filesystem-like level of indirection; processes running inside of a computation are not permitted to request arbitrary hashes — only those hashes statically mapped into their filesystem namespace at startup.

The cache management system is responsible for deciding which deterministically computed output blobs to retain in the content-addressed blob store. Doing this optimally is of course impossible, but doing it adequately should be feasible, since it can see the structure of the global transaction graph and how long each transaction took.

Performance

The grain size of the separate nodes in this computational graph can be quite small before it starts to cost significant efficiency, perhaps 128 kibibytes and a few million instructions, particularly with efficient

fork(), reasonably simple virtual memory mappings, and/or compiler-enforced security.

The VM code (for example, the blob of /bin/sort) produced by the various compilers can be AOT-compiled to native code for particular platforms, and perhaps this can also be stored in the cache. Under some circumstances it might make sense to recompile this native code with greater optimization over time.

Incrementality

I said that Kogluktualuk is "incremental". The way to get an incremental recomputation is to generate a new namespace that shares most of its mappings with an existing namespace, and then run a transaction in the new namespace that has previously run in the old namespace; if the computation decomposes into many subtransactions as suggested above, then only the subtransactions whose input data has changed will need to be rerun.

In the case of the sort example above, unfortunately, that includes the entire merge phase, a problem which could be reduced by passing a partitioning of the keyspace to the smaller-blob-sorting subtransactions, which could then produce as output not a single blob but a large number of blobs, one for each partition of the keyspace. Then you could split the merge phase into many independent merge transactions, perhaps only some of which would need to be rerun.

Dynamic dependency information

Getting this kind of incrementality conveniently requires extracting dynamic dependency information from the transactions as they run, like the Vesta SCM, Composable Memory Transactions, a serializable SQL transaction, or `redo`; the static dependency information is potentially very imprecise.

For example, perhaps the sort transaction above didn't happen to access /bin/grep, even though perhaps it is mapped; it would be useful if that allowed us to keep using the cached output of the sort transaction, even in a new namespace with a new mapping for /bin/grep.

Similarly, in the independent-merge-phase example above, it would be convenient if we could use the dynamic behavior of the merge phase to determine which partitions each merge transaction depended on. Perhaps this one:

```
/output/sorted.ZION-ZNGA  do /bin/merge /tmp/*.sorted.ZION-ZNGA
```

didn't happen to access /tmp/xac.sorted.XOOM-Z, even though it was mapped into its namespace; therefore a re-execution with a different /tmp/xax.sorted.XOOM-Z shouldn't have to re-execute the merge.

At this point, we have reproduced the sort-by-MapReduce from the original MapReduce paper.

How far can this approach go? Can you run your windowing system in it?

It might be feasible to use this approach even down to the level of handling mouse events; maybe moving the mouse generates a new

namespace which differs in that /dev/mouse now says "228,301\n" instead of "205,301\n", and the resulting incremental recomputation is capable of reusing almost all of the previous computation of the screen image to display, writing an /output/framebuffer with the new screen contents, which might be identical except for a mouse pointer and some mouseover highlight colors.

(You might want to write an /output/uistate that becomes the /input/uistate for the next UI transaction, too, and you might want to spawn the framebuffer update off in a subtransaction in case it isn't needed.)

I'm not sure taking it this far is a good idea, for a couple of reasons:

• It seems important to be able to guarantee responsivity for the user interface. This seems incompatible with depending entirely on nondeterministic caching for performance; if you decided to update /src/cc.c with a better-optimizing version, it would be unpleasant to have your next display frame delay three hours while the entire system was potentially recompiled.
• It seems important to be able to display the state of transactions in progress in the user interface, which seems sort of incompatible with the strong guarantee of determinism. But maybe the relationship between the user interface transaction and other transactions can be special.
• It seems like the user interface is a place where you need to be able to put in new hashes in order to fetch them from the cache, but we previously said that's a thing transactions can't do, because it could allow them to depend on data that isn't statically mapped into their namespace.

However, it seems like if you could make it work, you could get transparent persistence and migratability of your desktop environment, plus the ability to roll back to previous checkpoints. You only need to make sure that all of your mouse movements and keystrokes and whatever other events can affect your UI (completion of other transactions, maybe, or the passage of time) are safely recorded, either that or /input/uistate. Every point in time in your user interface has a hash, with which you can retrieve it and then explore other execution paths.

Overall significance

Kogluktualuk is a major advance in computing systems in terms of efficiency, simplicity, scientific reproducibility, digital preservation, practical software freedom, practical parallel computation, and security.

Reproducibility is a crucial requirement for scientific work, and digital preservation is a crucial requirement for historical work. It is often infeasible to determine what factors a computational result depends on. Kogluktualuk computations are reproducible forever, as long as their source data is preserved, and it automatically determines what that data is, so that you can preserve it.

For free software to be more than a theoretical construct, it needs to be possible in practice to recompile the software we are using. This is often unnecessarily difficult in current practice. Kogluktualuk, like Nix, automates this, so that you are guaranteed to be able to recompile anything you can run, except for a tiny bootstrap stub.

It is often difficult in practice to take advantage of the available parallel computational resources, for reasons including the following:

- heterogeneity,
- host exposure to attacks by untrusted code,
- private data exposure to untrusted hosts,
- exposure to untrusted hosts introducing malicious data,
- the increasing likelihood of hardware faults as scale increases,
- the excessive performance overhead of systems like Hadoop, and
- awkward programming interfaces and user interfaces.

Kogluktualuk improves the situation as follows:

- paper over heterogeneity with a uniform virtual machine at an acceptable performance cost,
- isolate untrusted code within sandboxes so it can't attack the host,
- have (I hope) very low performance overhead,
- automatically detect and recover some some kinds of hardware faults, and be configurable to detect and recover from all hardware faults, at a heavy performance cost,
- have the possibility of detecting and defeating malicious data introduction attacks, and
- have (I hope) convenient interfaces.

However, it does not protect against the disclosure of private data to untrusted hosts.

Occasionally free-software communities have been exposed to trojaned-binary attacks, including from SourceForge, in which the compiled binary has malicious code inserted into it that is not present in the source code. The most extreme form of this attack, demonstrated by Ken Thompson and previously hypothesized during the Multics security audit, involves inserting self-reproducing backdoor code into the compiler binary, so that even recompiling the compiler from source will not solve the problem. Currently, if any Debian Developer's development machine is compromised, the attacker can nearly undetectably compromise binaries built on that machine by inserting malicious code into them, and they will later be distributed to all Debian users who install that package.

The defense against these binary-poisoning attacks is reproducible builds, pioneered by the Tor project and now widely adopted, including by nearly all Debian packages. Kogluktualuk makes all builds reproducible, and it can be used to automatically detect such attacks.

In most existing computing systems, all computations have full access to read and write every file accessible by the user that launches them, as well as sending that data over the internet and manipulating and measuring the CPU load in order to communicate data through covert channels. In Kogluktualuk, each computation only has access to the data it is explicitly provided, and almost no computations have access to the clock or to the ability to pause themselves at will. This dramatically reduces users' vulnerability to malicious code.

Incrementalization of computation can often provide enormous performance increases, often three or more orders of magnitude; in the form of `make`, this was crucial, for example, to enabling the original development of UNIX in a high-level language on shared 0.4-MIPS computers in the 1970s. Current work in automatic

incrementalization, under the name "self-adjusting computation", is producing very promising results, but involves a slowdown of a factor of about five when the computation does not benefit from incrementality. Kogluktualuk should provide most of the performance benefit of full incrementality with no detectable overhead in the non-incremental case.

I've written before about the inefficiencies and bugs due to the numerous ad-hoc levels of caching in our existing computer systems. Kogluktualuk's unified caching system obviates every level of caching of greater granularity than a few million instructions, and it has the necessary global information to optimize caching globally. Even without any incrementality or parallelization benefit, this should produce better-performing computer systems, and they should be much simpler.

Unresolved questions

What does the user interface look like?

How do we index the computation cache so that it depends only on the dynamic dependencies of a computation, yet is fast to retrieve from? Maybe it would be better to spawn subtransactions in a very restricted namespace with only the things they depend on, even if that requires duplicating the information of which things they depend on in their caller?

What should the virtual machine look like? In particular, how should it handle floating-point math, and should it be vector-oriented in order to get better performance from available GPU (SIMT) and SIMD (e.g. SSE) resources?

How low can we get the overhead of forking off a subtransaction? Am I being too optimistic to think that 128 kiB and a few million instructions is big enough to amortize that overhead into insignificance? Or am I even being pessimistic? Does it depend on whether the subtransaction is going to get run on a different cluster node?

What about data that can potentially be computed in more than one way, depending on what data is available in cache? For example, in an OLAP system, you might want to see total sales by region (4 rows); if you have an existing materialized view of total sales by region and product category (40 rows) or total sales by region and customer type (16 rows), you can calculate the desired result very quickly, much more quickly than if you have to trawl over the entire dataset of, say, 50,000,000 rows. But maybe you only have one of those two views already computed, and it would be nice to use the one that is. Is there a way to fit this into the Kogluktualuk paradigm?

Guaranteeing responsiveness probably requires updating some cached items preemptively, rather than waiting for them to be needed.

What's the story on publicly-accessible caches? If you choose to trust a publicly-accessible binary cache for your Nix packages, you can avoid having to recompile anything yourself. But with Kogluktualuk, you might be at risk not only of getting malicious code from that cache, but also of sending that cache the hashes of private data in order to find out whether they are present there. You could reduce this risk with something like Bloom filters or compressed

Golomb rulers; is that enough, and if not, how can this risk be eliminated or made acceptable?

Anytime algorithms, which can be stopped at any point when time is running out, are important for real-time computation, for example in robotics and in some user interfaces, because they can guarantee real-time responsiveness without the heavy restrictions that are needed to deterministically bound the halting time of an algorithm. Many mathematical optimization metaheuristics automatically produce anytime algorithms. Can they be fit into Kogluktualuk's framework?

Truly random numbers are important for security. Enormously many secret keys have been compromised by compromised sources of randomness, including the Debian OpenSSL debacle and the Dual EC DRBG backdoor. One of the problems introduced by transparent checkpoint-and-restart systems like those in VirtualBox is that they can result in reuse of randomness, which can compromise that randomness. (One-time pads and DSA have notoriously bad problems here, but RSA keys with common moduli, for example, have also been broken en masse.) Kogluktualuk seems to allow checkpoint-and-restart functionality without this problem, because the granularity of the checkpoint is transactional, and you can probably avoid allowing the truly random data to escape a transaction. Does that really work?

What about resources like RAM? What do you do if a transaction wants to allocate ten gigabytes of RAM? Could that provide a covert signaling channel for exfiltrating private information to other transactions? Is there a way to ensure that some (or most?) transactions are small and light enough to run on small embedded processors?

Is there a way to get out of writing a compiler for a virtual machine instruction set, at least for prototyping Kogluktualuk?

Should we store everything in the same blob store — non-derived and thus irreplaceable source data, transaction outputs, native code for a particular processor, the cache database? Relatedly, how does Kogluktualuk relate to version control with Git?

In the cluster case, is the blob store implemented as a DHT, or what?

What interface does a transaction use to tell Kogluktualuk what its outputs are?

Is it really practical to do pipelining by having a pipeline of transactions that each read from the previous transaction's output, without having each one pause until the previous one is done? Is there a way we can usually avoid storing the data that flowed through the pipeline?

How do we modularize namespaces so that we can make a new namespace that differs from an existing one by changing one file (like /dev/mouse or /src/grep.c), without generating many megabytes of data traffic?

What exactly are the algorithms the cache service uses to figure out what to evict from cache?

In a cluster implementation of Kogluktualuk, how do we decide which transactions to run on which nodes? Does the hierarchical transaction structure provide enough information about the communication patterns to do a good job of this? If large blobs (like

my example /data/july-trades above) are sharded across nodes, how much of that do we expose to the transactions running on top in order to allow them to optimally divide up their subtransactions, and how do those subtransactions end up on the best node — do we migrate or restart them on a new node if they start accessing large data blocks, or what?

How do we deal with laggard nodes in a cluster implementation?

How do you fit reactive computation into this framework, if at all — how do you do, for example, a chat system? (Is there anything interesting in Urbit's implementation of chat?)

For things like what people use SPARK for, can we get adequate performance with, say, binary floating-point data in one column per file? (Kogluktualuk is a lot like SPARK.)

Topics

A type-inferred dialect of JS

Kragen Javier Sitaker, 2016-04-22 (4 minutes)

Suppose we want to do type inference on JS code so that we can run it faster or statically debug it or some shit like that.

Now, some things are really easy to infer a type for:

```
var projectionMatrix = [ [1, 0, 0, 0]
                       , [0, 1, 0, 0]
                       , [0, 0, 1, 0]
                       , [0, 0, 0, 1]
                       ]
```

That's clearly an Array of Arrays of Numbers, although you get into some weird subtyping stuff if you start caring about whether they're ints or not. (In this case they're not, because it gets mutated later on.) Also it happens that it's specifically a 4x4 array of arrays, and that matters in this case.

There are things that are intermediate:

```
function evToPoint(ev) {
  var ref = cvs.getBoundingClientRect();
  // This is clearly wrong, but my theory is that maybe this will make
  // it return SOME point within the canvas on old iOS.  Probably I
  // should use pageX there instead.
  return { x: (ev.clientX - ref.left) % cvs.width
         , y: (ev.clientY - ref.top) % cvs.height
         };
}
```

This takes an object `ev`, which needs to have `clientX` and `clientY` properties (but might have other properties too), and they need to be Numbers (except that in JS `"fuck" - "you"` is actually valid and returns a NaN), and returns an object with exactly the properties `x` and `y`, which are also Numbers, presupposing that `cvs` has a `getBoundingClientRect` property that is a function of no arguments returning a thing with `left` and `top` Number properties, and also `cvs` needs to have Number properties called `width` and `height`.

Now if it happens that all of these things are true, then we could probably use like a super optimized version of `evToPoint`. But if some of them aren't true, we might still want to remain JS-compliant, and just skip the optimization part. But if you can find all the calls to `evToPoint`, maybe you can prove that those properties are always true, so you can always use the optimized version and not even compile an unoptimized version.

Other things are really hard to infer a type for. Like `["x", 3]`, or `function(obj, prop, val) { obj[prop] = val; }`.

How about an RJS, though? Like RPython, but for JS. A restricted subset of JS that doesn't allow you to do anything that interferes with type inference. This seems feasible to me, although it's not going to be as simple as type inference for ML, because you unavoidably have some subtyping.

In particular, I think you need OCaml's upper-bound and

lower-bound types for objects with properties if you're going to have heterogeneous data structures: it's fine to pass {x:3, y:4, z:5} to a function that needs x and y to be numbers, but not fine to pass {x:3, y:4} to a function that needs x, y, and z to be numbers. Along the same lines, integers are a subtype of numbers in general, and probably an important one (although JS normally does gradual overflow, I don't think I've ever seen a JS program where this would matter). Once you're dealing with subtyping, maybe you could also deal with arrays of known size as a subtype of arrays of unknown size.

I'm not sure how exactly to deal with mutability, although subtyping might help. Immutable-array-of-integers is covariantly a subtype of immutable-array-of-numbers, and mutable-array-of-integers is a subtype of immutable-array-of-integers, but mutable-array-of-integers is not a subtype of mutable-array-of-numbers, because you can jam 1.5 into the latter. Unless I'm confused about the subtyping relation.

You probably also need to distinguish Objects used as hash tables (mapping strings to objects of some type) from Objects used as structs. Given the RJS niche, you could simply forbid using an object in both ways; it's trivial to rewrite `x.y = 3` into `x['y'] = 3`.

RJS wouldn't need to support inheritance.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- JS (p. 3533) (12 notes)
- Type inference
- Debugging

Notes from a Buenos Aires blackout, summer 2013–2014

Kragen Javier Sitaker, 2014-04-24 (15 minutes)

Scale: 53 minutes

It's 2013-12-24, about 0:30. Sweat is running down my face as I lounge naked on my bed, soaking into the sheet below me. The only light I can see in the room is from the netbook balanced on my thigh, which dimly illuminates the button-down shirt hanging from the inert light fixture above the bed on a hanger. My mouth is burning from the tallarines verde with hot sauce I bought from a street vendor on the way home.

Upon arriving home, I took about 0.3mg of melatonin, an antioxidant secreted by our pineal glands in the absence of exposure to blue light. It's dissolved in a reused bottled-water bottle in my silent refrigerator, which is still cool from the plastic coke-bottles of ice I stocked it with earlier. Soon the melatonin will probably induce drowsiness, and I'll drop off to sleep.

The neighborhood is quiet from this vantage point, nestled inside an apartment building with my tiny 2mx1.5m courtyard open to the sky. The occasional motorcycle or shouting person is audible in the distance. My neighbors have mostly gone to bed, since there's not much power. (At least one of the three phases feeding the building is still present.)

Before I sleep, I'll probably read half a chapter or so of *The Grammar of Graphics* by candlelight.

Scale: 160 minutes

Earlier, I walked home, enjoying the cooler outdoor air, from a restaurant near where I used to live, one which has Wi-Fi, an upper story where your laptop isn't visible from the street, and even outlets, although that last would have helped me more if I'd brought my laptop's bulky power supply. I argued with an atheist friend of mine who's being dragooned into leading prayers at a church in Korea, sympathized with another friend who's uncertain about her boyfriend's commitment to their long-distance relationship, wrote down phone numbers, and worked on a Bicicleta-to-JS compiler, which work is visible at https://github.com/kragen/bicicleta.py.

I also carried out a simple migration of data to a new AWS EC2 volume, something which needed to be done probably by tomorrow or possibly Thursday.

Mostly, of course, I read things. Articles, tweets, messageboards. Am I learning things? Probably.

Eventually my laptop ran out of battery, and having already paid, I stuffed it into my backpack, waited a while, and then speedwalked out of the restaurant.

The street vendor asked $27 for her tallarines verdes, which I didn't notice had a chicken wing on top. Not having $27, I offered her $18, which she accepted. The food was kind of lukewarm, having sat in a cooler in the hot Buenos Aires summer all afternoon. I hope I don't

get food poisoning.

As I sleep, I will probably be bitten by mosquitoes. I killed six last night, and I've already heard one tonight, though they're hard to see without lights. I'm glad dengue hasn't yet returned to Buenos Aires. I don't think the power outages are helping with that.

Scale: 8 hours

Darius and I went to a café with Wi-Fi when we discovered power was back out at my house, hoping to collaborate on the Bicicleta implementation. We didn't end up collaborating much, and I'm not sure what the reason was. Maybe Darius was tired of making the effort to communicate with me, or maybe he felt he'd be more productive working on his own on the table next to me.

Addled by the heat, I couldn't find the café I was looking for, but fortunately Buenos Aires is full of cafés with Wi-Fi.

As the night proceeds, it should cool off and become more comfortable. I'm keeping a bottle of cold water by the bed just in case.

Scale: 24 hours

Tomorrow morning I will wake up and go to the rose garden in Palermo for a date with a friend of mine, where I can paddle around the lake in a paddleboat. Fortunately I've refound the sun-shading hat I bought in Chinatown for my birthday party last month.

This afternoon I worked on visualizing foreign-exchange market data with d3.js with my coworker Ruth, who is just learning to program but alongside whom I am still much more productive. I was planning to wake up to start work earlier, but my phone ran out of battery, and I thought I'd wake up on time without it, but I didn't, probably because I didn't go to sleep last night until 5 AM.

Fortunately Ruth's charger was able to charge my phone.

I was also planning to work on this fractal calendar thing http://canonical.org/~kragen/sw/dev3/siercal.html with my friend Ganesha, but that fell through. I have some ideas.

Scale: 3 days

I washed a bunch of laundry in a bucket in the bathroom yesterday, making a big dent in the pile of laundry needing doing. My broken hot-water heater has ceased to be a handicap at the moment; cold-water showers are quite comfortable, and most days I take three or four of them. But washing clothes by hand, or by foot, is still a lot of work. It's amazing how much grime comes out of clothes. I think most of it is soot from diesel engines. I imagine my house air would be a lot healthier with heavy filtering.

I also ended up helping a friend of mine in California with a visualization project she's doing with d3.js, which I'm only slightly more experienced with than she is.

After my date tomorrow, I may end up working on the client project some more. This depends, in part, on the availability of electrical energy in my dwelling. Later, I'm going to a pluralistic holiday gathering, where five people with six nationalities and at least three faiths will celebrate together.

It won't be nearly as quiet tomorrow night. The traditional

Christmas fireworks are due.

Scale: 9 days

I got paid by my client, which enabled me to pay Ruth. I'm frustrated with how slowly the project is progressing. I also spent a bunch of time working on a fun little project, a lightweight web server --- currently 1928 bytes of 386 machine code running on Linux. http://canonical.org/~kragen/sw/dev3/server.s.

Power was out in my apartment, and nearby --- like the traffic light on the corner --- from Tuesday to Friday of last week, the days it was hot. It returned on Friday (at which point Darius had booked other accommodations with less mosquitoes and sweat) and lasted until today. It'll probably be out much of this week, too.

On Friday I went to a homemade pizza party at a friend's house nearby. I expected it to be quick to travel there, but due to the power outages, there were protestors cutting off streets a few blocks from my house, and the bus lines were rerouted and, when I found the stops, infrequent.

Saturday I went to the hacklab, where some people are scanning books with a homemade laser-cut apparatus similar to the Internet Archive's Scribe. We saw off a friend of ours who is leaving the country, and Violeta and I talked for a while.

My friend Jake Appelbaum's apartment in Berlin was broken into, presumably by US spies. Nothing was taken. He's a political dissident in exile from the US as a result of persecution, a result of his work on behalf of Edward Snowden and WikiLeaks. (None of this is private information; it's been written about extensively in Der Spiegel and other newspapers.)

Snowden, for his part, has suggested that Brazil ought to grant him asylum. Former KGB executive Vladimir Putin, who has granted him temporary asylum, said he wishes Russia could get away with what Obama is doing in the US. Snowden is still unable to make his way to any country that hass granted him permanent asylum.

In the next few days, I should reinflate my bicycle, clean up my patio, bedroom, and kitchen, try to keep the fridge stocked with ice bottles, and see if I can get the hot-water heater working again. If power comes back on, maybe I can get an air-conditioner dude to come see if the air conditioner is leaky or just undersized.

Also, I need to get an Android phone, both to develop for it and for the other useful aspects of Android. Being able to write on the bus at 40wpm instead of 17, having a portable wi-fi hotspot for larger computers, showing people videos, and consulting bus directions are all desirable features.

Scale: 27 days

I went to see a gorgeous circus performance by some friends of mine and a dozen or so of their classmates.

My friend David Kendal has still not had his server returned by the police, but has replaced it; it was seized in an investigation of our mutual friend Lauri Love, who has been indicted for breaking into computers remotely for activist purposes, but is presently at liberty.

My vocal folds have been irritated and unhappy since my birthday party a month ago, to greater or lesser extents. Possible contributing factors include air pollution, eating too much food seasoned with

pepper spray (a gift from Ruth after my mugging, which I will write about at greater length soon), talking all the time (and loudly, so Darius can possibly understand me), and reverse glottal fries.

I think I should probably take a short vacation, somewhere rural and maybe more polar. A friend of mine has suggested we hitchhike to Misiones together, but is still uncertain.

My aunt has a detached retina and needs help. I cannot travel to help her.

Scale: 81 days

I've gone to see a couple of immigration lawyers, but so far don't have a signed contract for either one to represent me. I'm still living in the same apartment I have been since February, but I think shortly will want to move. I spent some time looking at an eight-bedroom house with friends, but so far we haven't been able to put together a deal to actually move in there.

I've connected with a polyamory group here in Buenos Aires. So far, not much has come of it in terms of developing norms or sharing useful advice. Most of the participants are in their early 20s, and so are more interested in drinking beer and trying to hook up than in talking about strategies for managing jealousy, starting cohousing projects, or passing along info on poly-friendly lawyers and therapists. While I have no problem with people hooking up, and I at least tolerate people drinking, those aren't the things that are scarce in my life right now.

Beatrice has a paid job again, for almost the first time since 2006. And I've been working on this client project, and so we're on a path to pay off our shared debt, one of the few things we haven't yet unshared from the time we were married.

My sister is pregnant with twins.

My friend Stig Hackvan died unexpectedly, apparently from stomach cancer.

I was robbed on the train on November 9th, on the way to the memorial hackathon for my friend Aaron Swartz, who committed suicide in January after years of political prosecution by the US government for his activism. More details on the robbery later. I'm fine, it was just money.

In the next month and a half, I need to close a deal with an immigration lawyer to represent me and regularize my immigration status. This will make it possible for me to travel, and also start moving me toward Argentine citizenship again, which I've been working towards for seven years now.

A friend of mine is moving to Buenos Aires with her retired dad in mid-January, and I'm going to help them find a place to live.

I celebrated my birthday in the park on November 23rd. It was a potluck. A friend and I made polenta-based sushi. About 20 people showed up in all over the course of the 9 hours we were in the park, although a couple tried and didn't make it, because they couldn't figure out which direction from the duck pond was east, I guess, and because I let my cellphone run out of battery.

I gave how-to books on misoprostol abortions to everyone who came to the birthday party, although a few people refused. Abortion is almost entirely illegal in Argentina, as is mifepristone, but misoprostol is legal, safe, and widely available. It's slightly less

effective than mifepristone, but produces abortion in about 90% of cases. And it is legal to distribute information on how to perform abortions, although if the police had come by to hassle us, I'm not sure I could have persuaded them of that.

I've been single since September 27th, which is almost three months. The last time I was single this long was when I was 17. I think this is a good indication that finding a partner for a polyamorous relationship in Buenos Aires is going to be a lot more difficult than it would have been to find a partner for a theoretically monogamous relationship that isn't really, which seems to be the usual arrangement here. But I'm sure I'll fall in love with somebody again, in the next month or two. There's one person in particular I've had my eye on for a long time who *might* be interested.

And, if I manage to stay productive, I'll be able to pay off my debts.

On September 28th, I gave a talk on privacy software at the 30th anniversary celebration of the GNU project, which we celebrated at the hacklab here in Buenos Aires. Due in part to Snowden's revelations, interest in privacy software has increased substantially since then.

Time

It's now 02:00. It's still hot. I'm going to try to sleep.

PS

Power returned 07:00, went out again 10:00. My date didn't show up to the rose garden, but it was closed anyway.

Topics

Diode logic

Kragen Javier Sitaker, 2018-06-17 (16 minutes)

I saw on Hackaday that Ted Yapo made a digital clock out of diodes and oscillators. This ought to be impossible, since diodes can't invert or amplify, but it turns out that it isn't, because of reverse recovery time.

Yapo is using regular power rectifier diodes as RF switches, like PIN diodes. When they're forward-biased or even zero-biased, they pass RF AC with no problem. When they're reverse-biased, the depletion region grows, the junction capacitance drops, and they block RF AC (up to some rolloff frequency, anyway). So you can use DC voltage biases to get them to block or pass RF. Then you can use a faster small-signal diode to rectify the RF AC into DC, which you can use for further control signals.

Essentially, you carry two signals from two different power supplies on the same wire at the same time, allowing you to do something thought impossible for decades.

You could argue that this isn't "really" diode logic because you need an external oscillator to drive it. Yapo says this is a bogus criticism — the external oscillator is just another power supply, like how you need regulated 5VDC for TTL and regulated 3V3 or 1V8 for many modern CMOS chips. You don't need more oscillators as your logic gets more complicated.

(Incidentally, I've seen an avalanche relaxation oscillator LED flasher that works by back-biasing the collector-base diode junction of a transistor until reverse breakdown, so I suspect you can get an RF oscillator out of relatively ordinary diodes too, not just exotics like Gunn diodes and tunnel diodes.)

It occurred to me that maybe you don't even need two different kinds of diodes for this trick. You can use different regions of the response curve of a single kind of diode. You can use a capacitor (with more capacitance than your back-biased diodes) to move an AC signal from riding on one DC voltage level to another. If the DC level is close to the diode's forward drop, it rectifies the RF signal. If it's above the diode's forward drop, it passes the RF signal; if it's below it, it blocks it.

This is kind of tricky because it seems like it could be hard to get amplification with just one kind of diode this way. Unless you're going to use transformers, the RF voltage needs to be high enough to, when rectified, switch a diode from blocking RF to passing RF. I was thinking that this wasn't really a problem because you can use an arbitrarily small DC voltage to build up an arbitrarily large charge as you back-bias a diode, but it's actually a bit trickier than that — the small DC voltage will stop building up the large charge once the voltage created by that charge gets high enough. But I think you can use a standard voltage multiplier circuit (Cockcroft-Walton generator) made of diodes and capacitors to solve this problem. (I say "I think" because I'm not sure if this would require a different kind of diode.)

A standard 1N4148/1N914 handles 10 mA at .75 V forward and can peak at 100 mA at 1.1 V forward, withstands up to 75 V reverse, and

has .9 pF junction capacitance and 4 ns reverse recovery time. I think this means it can usefully rectify signals up to about 60 MHz. Common LEDs can also pulse up into the tens or hundreds of MHz. Schottky diodes don't have a reverse recovery time but have more capacitance; a 1N5819 has 150pF and can handle 1 A at .4 V forward. That capacitance will pass 1 A at .4 VAC at 2.7 GHz, although presumably their lead inductance will start to be a problem well before that, and you probably don't really want to run your logic circuits at 400 milliwatts per diode anyway, and lower currents will make the capacitance proportionally more important — 27 MHz already passes 10 mA.

(2.7 GHz? Is that really true? To charge 150 pF up to a .6 volt peak, I guess you need 90 pC, which an amp would deliver in 90 picoseconds, which gives you a waveform period of about four times that, which is indeed about a third of a nanosecond. Wow.)

There are smaller Schottky diodes with correspondingly smaller capacitances. The 1N6263 offers 0.1 pF and can carry 1 mA, which hits the corresponding transition at 3.9 GHz. Amusingly, at the other end of the scale, things look better: the MBRP40045 can carry 400 amps (!) and claims only 3500 pF, which at an AC RMS voltage equal to its 540 mV forward drop, would only start carrying 400 amps RMS at 34 GHz. But 3500 pF would resonate in series with its own leads at 2.7 GHz with only one picohenry of parasitic lead inductance.

Using higher radio frequencies allows you to use smaller capacitors to pass them or smaller inductors to block them. At 10 MHz, for example, 22 pF is 700 Ω, and 100 μH is 6000 Ω; at 20 MHz, 10 pF is 800 Ω and 47 μH is 6000 Ω.

The effective switching speed of logic circuits built this way might be able to reach a tenth of the carrier frequency of the RF power supply. The rectifier output needs to be smoothed over intervals in that ballpark, and the "DC" signals need to be separated from the "RF" by at least that much.

(Speaking of inductors, square-hysteresis-loop magnetic logic had somewhat similar characteristics, using a DC signal to push a magnetic core into saturation and thus pass or block an AC signal, also restoring the signal edges with the sharp hysteresis. It was much slower, though, because magnetic domains take a while to move around, like nearly a millisecond. This approach seems like it might have the potential to reach into the MHz.)

The AC current produced by a small AC voltage across a diode riding on some DC bias is an exponential function of that DC bias, which will then produce a nicely exponential AC voltage if fed into, say, a resistor or capacitor. This suggests a sort of alternative approach: rather than trying to suppress the RF entirely by back-biasing the diode, just move down closer to the threshold voltage to attenuate it more, but not so close that you're effectively rectifying. I am not sure if this will work. Let's try to work it out.

Let's see if we can fit a curve to the 1N4148 figures I gave earlier. The current should grow by a factor of e every 25.3 mV, according to my understanding of the Ebers-Moll equation; in fact, though, it grows only by a factor of 10 over the 0.35 volts from .75 to 1.1 volts, when actually it should be growing by a factor of a million. Horowitz & Hill has some diode curves, including LEDs, in figure 2.8; it gives the 1N914 (supposedly the same as the 1N4148) as ramping up to

about 30 mA at something like 0.7 volts.

I measured some points on the curve in the Gimp. 0V is 97 pix and 1 V is 240 pix and 2 V is 384 pix, for 143 or 144 pixels per volt; 5 mA is 374 pix, 10 mA is 301 pix, and 15 mA is 230 pix, for about 144 pixels per 10 mA. So using the formula ((x - 97) / 143.0, (301+144-y) / 14.4) we can transform points measured on the curve with Gimp's crosshairs; we get 0.48 V, 0.1 mA; 0.59 V, 0.6 mA; 0.66 V, 1.4 mA; 0.71 V, 2.5 mA; 0.75 V, 5 mA; 0.79 V, 7.8 mA; 0.80 V, 10.3 mA; 0.85 V, 20 mA; 0.86 V, 30 mA.

This suggests that, at some temperature, a voltage wiggle between 0.59 V and 0.66 V (70 mV) will result in a current wiggle of 0.8 mA (11.4 mʊ̄); a voltage wiggle between 0.66 V and 0.71 V (60 mV) will result in a current wiggle of 1.1 mA (18.3 mʊ̄); a voltage wiggle between .71 and .75 V (40 mV) will result in a current wiggle of 2.5 mA (63 mʊ̄); a voltage wiggle between .75 and .79 V 2.8 mA (70 mʊ̄); between .79 and .85 V 12.2 mA (200 mʊ̄); and presumably continuing from there.

(This amounts to a factor of 5 increase in current over 270 mV, from 590 mV to 860 mV, which would be an increase by a factor of e every 168 mV, not evey 25.3 mV.)

Let's leave aside the delicate question of biasing the circuit to 0.48 V or 0.59 V or whatever rather than 0.45 V or 0.65 V, which are points that shift with temperature. Let's suppose it's feasible.

The thing I'm not sure about here is that I feel like we don't really get a place with more or less rectification along this curve. As long as dI/dV keeps growing exponentially, an AC waveform of some fixed voltage will always have less conductance on its lower half, in fact by always exactly the same ratio, so it will always be the same percent rectified, leaving aside frequency-dependent effects like reverse recovery time. But I guess that if we put a resistor in series with the diode, the resistor-diode series unit will gradually become more symmetric as more of the voltage is across the rectifier. And of course diodes have their own internal resistance, which has the same effect, even if it isn't visible from the curve there yet.

Again I am feeling skeptical about amplification, though. At 800 mV DC and 10.3 mA, the control signal costs 8.2 mW. If we are switching a 120 mV p-p RF waveform, which turns out to be roughly 17 mA p-p, are we really getting amplification? It's dissipating only about 0.25 µW in that diode... but maybe we can rectify that to about 6 or 7 mA DC and run it through two or three volts of other diodes in series (three or four diodes) as a control signal? Or pump it up to 1600 mV in a 13-stage Cockcroft-Walton generator?

The thing that's confusing me here is that I confuse the energy *dissipated within* the device with the energy *controlled by* the device. In theory we could be driving those 17 mA ac (rms) through any number of other diodes in series; but I don't have a clear idea as to how.

By contrast, it's totally clear to me that the mechanism of reverse bias and reverse recovery time exploited by Yapo provides amplification: the energy to maintain a diode in reverse bias is just the energy to replenish the leakage current — for the 1N4148 that's 10 nA at 20V reverse bias, which would be 200 nW, and less if you're only reverse-biasing it to 2V or so — and the energy to maintain it at DC ground is zero. But if you put a 125MHz sinewave at 1V RMS across it, it will pass it almost without resistance when it's not back-biased,

but block it when it's reverse-biased. This gives you near-infinite power gain, like a MOSFET.

(Yapo is actually using the 1N4148 for his high-speed rectifier and a 1N4007, a hefty power rectifier diode, as his switch, because he's using only 4.5 MHz as his RF frequency.)

I think I have a clearer idea of how much power you can control with such a switch now. Consider a switch with some resistance in general. When its output is shorted, it's controlling no power, because there is no voltage across the load; similarly when it's open-circuited, because there is no current through the load. As we add conductance starting from an open circuit, the current starts to creep up, but since the current causes some voltage to be dropped across the switch, the voltage across the load starts to fall. Load power is at its maximum when precisely half the voltage is across the load, and as we move further toward a short circuit, although current continues to increase linearly with the conductance, power starts to drop. The overall shape is a parabola; this is easily visualized as being the product of two linear ramps, one from zero current at open circuit up to the current from the switch's conductance at short circuit, the other from zero voltage at short circuit up to all the voltage at open circuit. Those are the roots of the quadratic, from which it immediately follows that its extremum is halfway in between.

Things get a little more complex when the switch impedance is partly imaginary, but I think the difference is relatively small, not orders of magnitude — except of course that the switch doesn't dissipate power from the imaginary part of its impedance.

So the power that a switch can control to a load is at or near its maximum when the switch itself is dropping the same amount of voltage. So, for our example earlier, if the part of our waveform across the switching diode is 120 mV p-p, it could have another part across the load of another 120 mV p-p for optimal power, which means that at 17 mA p-p it can be delivering uh 0.354^2 120 mV 17 mA = 0.255 mW of power to that load, just as it's dissipating in the switching diode, about 3% of the DC bias current. So, basically, no, that part of the curve will not work for amplification.

Yapo's "diode-diode logic" design, by contrast, reverse-biases the switching diodes to empty out a generous depletion region around the junction in order to block the RF energy. This is essentially static — a 1N4007 can, according to Vishay's datasheet, carry 1000 mA forward and presumably an even larger amount as RF, and leaks 5 μA when back-biased to 1000 V. At the -15 V of Yapo's design, according to Yapo's "DDL01 DDL Hex NOR Gate" datasheet, it should presumably be more like 75 nA of bias current and 1.13 μW, while the 12 Vpp RF ends up as 4.2V RMS at something like 8 mA, which works out to 34 mW, about 30 000 times larger. Nevertheless, the circuit barely works — he had to resort to two stages of amplification per gate and over 20 components, including inductors, and supports fanout of up to about 5.

Topics

• Electronics (p. 3430) (138 notes)

Notes on Óscar Toledo G.'s bootOS

Kragen Javier Sitaker, 2019-10-07 (updated 2019-10-08) (28 minutes)

Recently Óscar Toledo G. released bootOS, an MS-DOS-like operating system with a built-in filesystem and hex editor; it provides some basic OS services and can launch programs, and fits entirely into the 512-byte boot sector of a floppy disk.

Background

The author is a wizard from a secretive underground society of wizards known as the Familia Toledo; he and his family (it is a family) have been designing and building their own computers (and ancillary equipment like reflow ovens) and writing their own operating systems and web browsers for some 40 years now. Unfortunately, they live on the outskirts of Mexico City, not Sunnyvale or Boston, so the public accounts of their achievements have been mostly written by vulgar journalists without even rudimentary knowledge of programming or electronics.

And they have maintained their achievements mostly private, perhaps because whenever they've talked about their details publicly, the commentary has mostly been of the form "This isn't possible" and "This is obviously a fraud" from the sorts of ignorant people who make a living installing virus scanners and pirate copies of Windows and thus imagine themselves to be computer experts. (All of this happened entirely in Spanish, except I think for a small amount which happened in Zapotec, which I don't speak; the family counts the authorship of a Zapotec dictionary among their public achievements.) In particular, they've never published the source or even binary code of their operating systems and web browsers, as far as I know.

This changed a few years back when Óscar Toledo G., the son of the founder (Óscar Toledo E.), won the IOCCC with his Nanochess program and four more times as well. His obvious achievements put to rest — at least for me — the uncertainty about whether they were underground genius hackers or merely running some kind of con job. Clearly Óscar Toledo G. is a hacker of the first rank, and we can take his word about the abilities of the rest of his family, even if they do not want to publish their code for public criticism.

I look forward to grokking BootOS in fullness and learning the brilliant tricks contained within! Getting a full CLI and minimalist filesystem into a 512-byte floppy-disk boot sector is no small achievement.

It's licensed under the two-clause BSD license.

The significance of bootOS

If you have a PC with BIOS and a floppy disk, but no software and no other computer, you have a paperweight, because you have no way to program it. bootOS is sufficient to program it, albeit in hexadecimal machine code, and to store your programs on the disk so

that you don't have to start from scratch every time the machine loses power. And it's close to the bare minimum amount of software to provide a programming environment you could actually use.

A brief listing of bootOS in octal

Here's `od -b os.img`:

```
0000000 061 300 216 330 216 300 216 320 274 000 167 374 276 000 174 277
0000020 000 172 271 000 002 363 244 276 354 173 277 200 000 261 006 245
0000040 253 342 374 276 274 173 350 106 001 315 040 374 016 016 016 037
0000060 007 027 274 000 167 260 044 350 006 001 200 074 000 164 354 277
0000100 310 173 212 005 107 045 377 000 164 021 221 126 363 246 165 004
0000120 377 025 353 327 001 317 107 107 136 353 347 211 363 277 000 174
0000140 315 043 162 002 377 343 276 303 173 350 003 001 315 040 211 363
0000160 254 074 040 164 371 315 045 162 355 303 350 240 000 211 337 200
0000200 075 000 164 005 211 376 350 346 000 350 156 000 165 361 303 126
0000220 061 311 254 101 074 000 165 372 136 277 000 170 303 127 006 350
0000240 100 000 264 002 007 133 162 003 350 204 000 211 345 320 126 004
0000260 317 127 006 123 315 045 133 350 325 377 046 200 075 000 164 007
0000300 350 067 000 165 365 353 335 127 363 244 350 132 000 137 350 062
0000320 000 264 003 353 317 350 012 000 162 321 271 020 000 350 103 000
0000340 353 311 123 350 067 000 136 350 245 377 126 127 121 363 246 131
0000360 137 136 164 017 350 003 000 165 361 303 203 307 020 201 377 000
0000400 172 371 303 215 205 020 210 261 004 323 340 100 221 303 277 000
0000420 170 271 000 002 350 014 000 273 000 172 111 353 022 016 007 264
0000440 002 353 006 260 000 363 252 264 003 273 000 170 271 002 000 120
0000460 123 121 006 260 001 061 322 315 023 007 131 133 130 162 360 303
0000500 315 042 276 200 167 211 367 074 010 165 002 117 117 315 041 074
0000520 015 165 002 260 000 252 165 357 303 264 000 315 026 074 015 165
0000540 006 260 012 315 042 260 015 264 016 273 007 000 315 020 317 254
0000560 315 042 074 000 165 371 260 015 315 042 303 277 000 174 127 260
0000600 150 350 274 377 137 200 074 000 164 022 350 034 000 163 357 261
0000620 004 322 340 221 350 022 000 010 310 252 353 356 260 052 350 237
0000640 377 126 133 277 000 174 315 044 303 254 074 000 164 015 054 060
0000660 162 367 074 012 162 005 054 007 044 017 371 303 142 157 157 164
0000700 117 123 000 117 157 160 163 000 003 144 151 162 172 172 006 146
0000720 157 162 155 141 164 016 173 005 145 156 164 145 162 173 173 003
0000740 144 145 154 156 172 003 166 145 162 043 172 000 053 172 131 173
0000760 135 173 235 172 261 172 325 172 117 117 117 117 117 117 125 252
```

Here's a version formatted for readability, insofar as that is possible for an octal dump of machine code:

```
061 300 216 330 216 300 216 320 274 000 167
374 276 000 174 277 000 172 271 000 002 363 244
276 354 173 277 200 000 261 006
245 253 342 374
276 274 173 350 106 001 315 040
374 016 016 016 037 007 027 274 000 167
260 044 350 006 001
200 074 000 164 354
277 310 173
212 005 107 045 377 000 164 021 221 126 363 246 165 004 377 025 353 327
001 317 107 107 136 353 347
211 363 277 000 174 315 043 162 002 377 343
```

```
276 303 173 350 003 001 315 040
211 363 254 074 040 164 371 315 045 162 355 303
350 240 000 211 337  200 075 000 164 005 211 376 350 346 000
    350 156 000 165 361 303
126 061 311  254 101 074 000 165 372  136 277 000 170 303
127 006 350 100 000 264 002  007 133 162 003 350 204 000
    211 345 320 126 004 317
127 006 123 315 045 133 350 325 377
    046 200 075 000 164 007 350 067 000 165 365 353 335
    127 363 244 350 132 000 137 350 062 000 264 003 353 317
350 012 000 162 321 271 020 000 350 103 000 353 311
123 350 067 000 136 350 245 377  126 127 121 363 246 131 137 136 164 017
    350 003 000 165 361 303
203 307 020 201 377 000 172 371 303
215 205 020 210 261 004 323 340 100 221 303
277 000 170 271 000 002 350 014 000 273 000 172 111 353 022
016 007 264 002 353 006
260 000 363 252  264 003  273 000 170 271 002 000
    120 123 121 006 260 001 061 322 315 023 007 131 133 130 162 360 303
315 042 276 200 167 211 367  074 010 165 002 117 117
    315 041 074 015 165 002 260 000  252 165 357 303
264 000 315 026  074 015 165 006  260 012 315 042 260 015
    264 016 273 007 000 315 020 317
254 315 042 074 000 165 371  260 015 315 042 303
277 000 174  127 260 150 350 274 377 137 200 074 000 164 022
    350 034 000 163 357 261 004 322 340 221 350 022 000 010 310 252 353 356
    260 052 350 237 377 126 133 277 000 174 315 044 303
254 074 000 164 015 054 060 162 367 074 012 162 005 054 007 044 017 371  303
142 157 157 164 117 123 000
117 157 160 163 000
003 144 151 162  172 172
006 146 157 162 155 141 164  016 173
005 145 156 164 145 162  173 173
003 144 145 154  156 172
003 166 145 162  043 172
000
053 172  131 173  135 173  235 172  261 172  325 172
117 117 117 117 117 117  125 252
```

Some commentary

It would be remiss not to mention that Toledo has written and self-published a book about this kind of programming.

Cold-boot code

```
061 300  216 330  216 300  216 320  274 000 167
```

This is `ax ^= ax;  ds ← ax;  es ← ax;  ss ← ax;  sp ← 167 000`. Presumably this means the BIOS doesn't reliably set the segment registers or stack pointer to sensible values, and that it loads the boot sector with a CS of 0. Also presumably there should be a cli/sti pair protecting the last couple of instructions.

```
374  276 000 174  277 000 172  271 000 002  363 244
```

This is memcpy (`rep movsb` — 363 244) of 002 000 bytes (in cx, register 1) from 174 000 (in si, register 6) to 172 000 (in di, register 2) with ascending addresses (374, `cld`); this copies bootOS itself, as explained in the comments in the source. BIOS loads programs at 174 000 (0x7c00) and perhaps bootOS wants to load programs in the same place so that if a program is written to run as a boot sector it will still run under bootOS.

After this point, everything can assume bootOS is in place at 172 000.

```
276 354 173   277 200 000   261 006   245   253   342 374
```

This loop sets up the interrupt vectors, each of which is 4 bytes, IP then CS, starting at address 000 200. bootOS uses a separate interrupt vector for each system call, unlike MS-DOS and the BIOS; MS-DOS does provide a deprecated int 0x20 (040) to exit a program, and bootOS uses this same number for its own warm-boot system call. (bootOS programs exit by warm-booting, like CP/M programs.)

This loads register 6 (SI) with the address of the six interrupt vectors toward the end of the boot sector, at 173 354, then register 7 (DI) with the interrupt vectors, and CL with the number 6. (We're justified in assuming CH is 0 because we just got here from the rep movsb above.) Then 245 (movsw) copies 16 bits from [SI] to [DI], incrementing both pointers, and 253 (stosw) stores 16 bits from AX, which is still 0 from the 061 300 instruction at the beginning of the program, so all the CS fields of the interrupt vectors will be 0, as they should be.

`ver` command

```
276 274 173   350 106 001   315 040
```

This is actually the code for the "ver" command; it's placed here so that, now that we're done with the 35 bytes of system initialization code that set up the interrupt vectors, we display the version banner before displaying the prompt (by warm-booting with 315 040, int 0x20). 173 274 is the address of the ASCIZ string "bootOS" below (142 157 157 164 117 123 000), and 001 106 is a PC-relative call offset to the `output_string` routine below (254 315 042 074 000 165 371 260 015 315 042 303).

CLI

```
374   016 016 016   037 007 027   274 000 167
```

That's the warm-boot code, invoked by 315 040 — it runs CLD (374) and copies CS (loaded from the interrupt vector if necessary) into DS, ES, and SS via the stack, and then resets the stack pointer. This seems a little hazardous to me — if you suspect SS or SP may be pointing somewhere random, I'd think you wouldn't want to write to the stack. Also I'd think you'd want to have interrupts disabled.

Interestingly, this sequence is two bytes shorter than the cold-boot sequence that does the same thing using AX, but that sequence also

clears AX. At this point AX may have crap in it, which has consequences for the loop over built-in commands below.

Even though it's usually invoked by an interrupt instruction, it never returns from the interrupt, and since it forgets where the stack pointer was pointing, it can't.

```
260 044  350 006 001
```

This sets AL to 044 '$' and calls `input_line` with a PC-relative call. So now we are well and truly into command-line handling.

```
200 074 000  164 354
```

This is a special case for empty commands so you can hit Enter at the prompt and not get an error message or run a file whose name is the empty string; it's checking to see if the first byte in the buffer pointed to by SI is a NUL.

```
277 310 173
```

This sets register 7 (DI) to 173 310, where there's a list of built-in commands, each followed by its address.

```
212 005 107 045 377 000 164 021  221 126 363 246 165 004  377 025 353 327
    001 317 107 107 136  353 347
```

This is a loop over the built-in command names. We have DI pointing to a length-prefixed command name and SI pointing to the (non-length-prefixed!) command line. First we load AL (low register 5?) indexing 0 with DI, then increment DI (107: register 7). To see if the length byte was 0, we use 045 377 000, ANDing AX (register 5) with 000 377, which would seem like a strange way to write `test al, al`, but remember that AH may still have crap in it, and shortly we are going to use all of AX as the count for a `rep` prefix. In the case that the length byte was 0, indicating we've come to the end of the command list, we jump 021 bytes (164 021) and out of the loop, to the next code following; but if not, we 221 (`xchg ax, cx`), save SI with 126 (`push si`) and use `rep cmpsb` (363 246) to compare the command name. (The comparison uses ascending addresses because we ran `cld` at the top of the warm-boot code, and nothing clears it.)

If the command name didn't match (and note that this is a prefix match!) we jump 4 bytes to the second line of the loop above with 165 004. But if it did, it does an indirect call via DI (377 025) which has now been incremented off the end of the command name, and then does an unconditional jump up to the warm-boot code above.

On the second line of the loop, DI is pointing somewhere into the middle of the command name, where we found a mismatch. To correct this, we add the leftover counts in CX to it with 001 317 (`add di, cx`, CX being register 1 and DI being register 7), increment DI twice (107 107) to step over the command handler address, 136 (`pop si`, register 6) to recover from the earlier 126, and jump 031 (=25) bytes backwards to the beginning of the loop to try another command.

Note that if we do invoke a built-in command due to this prefix match, we do so without restoring SI — so SI points to the text after the command, so it can take arguments.

You'd think that maybe switching the roles of SI and DI would make sense here, since the beginning of the loop is basically `lodsb` but with DI, but it might not save you enough to be worth the swizzling.

```
211 363  277 000 174  315 043  162 002  377 343
```

So now that we've exhausted the possibilities of built-in commands, let's try to load a file to handle the user's command. We have an ASCIZ string at SI, which we transfer to BX (211 363), then load a buffer address (174 000, the boot sector loading address) into DI, and then invoke the load-file interrupt (315 043). This routine indicates errors by setting the carry flag, so if that's set, we jump forward two bytes (162 002) to the next line; otherwise, we do an indirect jump through BX (377 343).

Although this has the same opcode byte 377 as the indirect call above, it really is a jump and not a call; nothing is left on the stack. So the newly launched program must return control to bootOS via the warm-boot interrupt or not at all.

```
276 303 173  350 003 001  315 040
```

This puts 173 303, the address of ASCIZ "Oops", into SI, then calls the `output_string` routine with a PC-relative call; then it warm-boots.

And that's the whole CLI loop: 67 bytes, including the warm-boot code. Everything that follows is subroutines and data; the bulk of it is the filesystem.

Commands, system calls, and the filesystem

These layers are kind of mixed together, which is probably somewhat unavoidable within the 512-byte limit; some orderings can allow you to use short jumps, too, or (as with the `ver` command above) omit jumps and just use fallthrough.

`del` command

```
211 363  254  074 040  164 371  315 045  162 355  303
```

That's the "del" command; it just invokes the deletion interrupt routine. It starts out with 211 363 `bx ← si`, which is presumably because the `delete_file` call takes its argument in BX, not SI.

Then it has a five-byte loop 254 `lodsb` 074 040 `al == ' '` 164 371 `jz $-7` which skips over spaces, overwriting BX again if necessary. This arrangement was somewhat puzzling to me at first, since you'd think it would make just as much sense to set BX once, outside the loop, once we really knew what we wanted to set it to (the position after we've skipped over the spaces). This doesn't really matter for efficiency, but it puzzled me that it was done in this non-obvious way. Eventually I realized that this repeat-until loop topology leaves BX with the *unincremented* value of SI. (This of course means that there would be no advantage to making `delete_file` take its argument in SI.)

Once that is achieved, it invokes the `delete_file` system call 315 045, reports any errors by jumping into the CLI loop's error handler if the carry flag is set 162 355, and then 303 returns.

`dir` command

```
350 240 000  211 337   200 075 000  164 005  211 376  350 346 000
    350 156 000  165 361  303
```

First this calls `read_dir` below with a PC-relative call. `read_dir` overwrites bx with the pointer to the disk sector in memory, and the next instruction 211 337 is `di ← bx`.

The rest of the function is a loop followed by 303 `ret`; the main body of the loop is 211 376 `si ← di`, 350 346 000 `call output_string`, and 350 156 000 `call next_entry`. `next_entry` advances di to the next directory entry and sets the carry flag if it's passed over the whole directory — and, implicitly, the zero flag if it has. 165 361 is a jump conditional on the *zero* flag, back to the beginning of the loop. (This suggests that the `stc` instruction in `next_entry` could be eliminated by making all of its callers use the zero flag instead of the carry flag — unless there's someplace where the *BIOS* disk routine's use of the carry flag to indicate errors is propagated to the caller.)

`next_entry` does not skip empty directory entries, so before the loop's main payload, `dir` does a 200 075 000 `byte[di] == 0` and a 164 005 to jump past the loop's main payload to the `next_entry` call in that case.

This compact code is enabled by the fact that `output_string` and the directory entries both use ASCIZ strings; it would be even more compact if `output_string` were to use `di` rather than `si` to point to its string.

opendir (filesystem routine)

```
126  061 311   254  101  074 000  165 372   136  277 000 170  303
```

In the source code this is called `filename_length`, but I thought that was somewhat misleading. It does count the length of the ASCIZ filename at SI (in CX), using 254 `lodsb` 101 `cx++` 074 000 `al == 0` 165 372 `jnz .loop`, rather than using `scasb`, for reasons I don't yet understand; around this it wraps a 126/136 pair to preserve SI (register 6). But it *also* sticks the pointer to the disk sector buffer `read_dir` uses into di: `di ← 170 000` before 303 returning, presumably because that value in di was useful in more than one filesystem routine.

load_file

```
127  006  350 100 000  264 002
    007  133  162 003  350 204 000
    211 345  320 126 004  317
```

This is an interrupt routine, so it returns with 317 `iret` rather than the usual 303. Moreover, it actually has three separate entry points: `save_file` merges with it at the beginning of the second line (`shared_file`), while `delete_file` merges with it at the beginning of the third line (`ret_cf`).

`iret` restores not only CS and PC (uh, "IP") off the stack, but also the flags? This means that `load_file` and its brethren need to mutate the saved flags in order to achieve this with `rcl byte[bp+4], 1` (320 126 004), having previously pointed bp at the stack with 211 345 `bp ← sp`. This presumably has the bizarre effect of shifting one byte of the flags register by 1.

XXX

save_file

```
127 006 123 315 045 133 350 325 377
    046 200 075 000 164 007 350 067 000 165 365 353 335
    127 363 244 350 132 000 137 350 062 000 264 003 353 317
```

delete_file

```
350 012 000 162 321 271 020 000 350 103 000 353 311
```

find_file

```
123 350 067 000 136 350 245 377  126 127 121 363 246 131 137 136 164 017
    350 003 000 165 361 303
```

next_entry

```
203 307 020 201 377 000 172 371 303
```

get_location

```
215 205 020 210 261 004 323 340 100 221 303
```

format command

```
277 000 170 271 000 002 350 014 000 273 000 172 111 353 022
```

This has a short jump at the end of it (a tail call, say) that would have had to be long if it had been more than 128 bytes from the target.

read_dir

```
016 007 264 002 353 006
```

write_zero_dir/write_dir/disk_dir/disk

These four entry points are connected by fallthrough to save space.

```
260 000 363 252  264 003  273 000 170 271 002 000
    120 123 121 006 260 001 061 322 315 023 007 131 133 130 162 360 303
```

input_line

```
315 042 276 200 167 211 367  074 010 165 002 117 117
    315 041 074 015 165 002 260 000  252 165 357 303
```

input_key/output_char

```
264 000 315 026  074 015 165 006  260 012 315 042 260 015
    264 016 273 007 000 315 020 317
```

output_string

```
254 315 042 074 000 165 371  260 015 315 042 303
```

`enter` command

This is the hex editor. For some reason I don't seem to be getting it to work in QEMU; I don't know if I'm using it wrong and it's not giving me an error message, if it's buggy, or if QEMU is buggy.

```
277 000 174  127 260 150 350 274 377 137 200 074 000 164 022
    350 034 000 163 357 261 004 322 340 221 350 022 000 010 310 252 353 356
    260 052 350 237 377 126 133 277 000 174 315 044 303
```

xdigit

This function is partly necessitated by the choice of hexadecimal, but it also skips spaces:

```
254 074 000 164 015 054 060 162 367 074 012 162 005 054 007 044 017 371  303
```

Data tables and the trailer

A couple of ASCIZ strings, "BootOS" and "Oops":

```
142 157 157 164 117 123 000
117 157 160 163 000
```

Then the table of built-in commands and their absolute addresses:

```
003  144 151 162               172 172
006  146 157 162 155 141 164   016 173
005  145 156 164 145 162       173 173
003  144 145 154               156 172
003  166 145 162               043 172
000
```

Finally, the table of interrupt vectors:

```
053 172  131 173  135 173  235 172  261 172  325 172
```

Then filler and the boot-sector signature 125 252:

```
117 117 117 117 117 117  125 252
```

Topics

Examination of a shitty USB car charger

Kragen Javier Sitaker, 2019-10-24 (13 minutes)

I found a car-cigarette-lighter USB charger on the street with a missing contact spring and some broken plastic. It's labeled as having one 1-amp USB port and one 2.1-amp USB port, although in fact these are wired strictly in parallel. It's fairly transparently a simple buck converter on a single-sided PCB (silkscreened "HT-668"), built around a buck-converter through-hole 8-pin chip labeled "LC51", followed by some kind of fraction-like sigil. There is some residual flux from sloppy hand soldering on the bottom of the board.

"LC51" is not a useful marking for finding the correct datasheet on search engines; it turns up many datasheets for chips used in this application but they all have the wrong pinout.

Circuit netlist and pinout reverse engineering

C1 is a 22-µF 25-V electrolytic across the nominally-12-volt input, with its - terminal on pin 4 of the IC and its + terminal on pin 6. L1 is an unmarked ferrite-core inductor across pins 2 and 3, and "D1" (with the "1" silkscreened upside down) is what looks like an ⅛-watt rectifier (labeled, I think, "N5 MI") across pin 4 (ground) and pin 2 (the stripey end of the diode package). "CS" is a 100-µF 10-V electrolytic on the output, which is connected to pins 4 (on its - terminal) and pin 3. There is a blue LED connected across pins 4 and 5. Pins 1 and 7 are shorted to pin 6. The USB connector shield is tied to pin 4.

Amusingly, there are no resistors.

So I infer the following pinout for the chip:

- 1: Vin
- 2: switched Vout
- 3: feedback sense
- 4: ground (common between input and output)
- 5: current-regulated? power LED indicator output
- 6: Vin
- 7: Vin
- 8: no connection

Except for the indicator LED, this is a textbook buck converter circuit. It seems strange to me that there are 3 pins for Vin and only 1 pin for Vout, but I guess most of the output current (7/12 at nominal 12V input) really flows through the diode and not the chip. Still, one would hope that the amount of current flowing to ground through the chip's internal circuitry is at most a few milliamps, not hundreds of them, so you'd want the same number of input and output pins.

Observed behavior

None of the components have any visible damage, but on connecting it to a 12-volt power supply sourcing an amp or two to a

LED illumination panel connected in parallel, the onboard blue LED lights up, but there is no 5-volt output. None of the components heat up noticeably.

After unplugging, some residual voltage is visible on both capacitors, tens to hundreds of mV, so they are not shorted.

The diode reads as a 170-mV voltage drop with this multimeter a friend lent me, but I'm not sure how correct that is. That seems too low to be a working diode. But in the other direction it looks like an open circuit, so I suspect the meter. However, unless I'm reading this wrong, the meter is telling me not only that the diode has an implausibly low forward voltage drop but also that it is installed backwards, i.e., passing current in the +5V-to-ground direction. Maybe the diode is blown and the voltage I'm seeing is actually from reverse-biasing the output electrolytic, or some kind of input protection circuit in the SMPS chip itself?

Design commentary

If we assume that the ⅛-W rectifier diode is a 300-mV Schottky, then it can carry an average of 417 mA, which means that the whole board can only supply 12/7 of that — 714 mA — without overheating the diode. So the "2.1 A" output rating molded into the plastic is probably unreasonably optimistic. There is also nothing connected to the data pins (two of them may be shorted together, but this looks like an accident) so no smart current negotiation can be going on here, but even without current negotiation, you could plug two 500-mA devices in at once, which would mildly overheat the diode.

It probably isn't really okay to use a 25-volt electrolytic on the input of this charger, either, because car electrical systems are reputed to have inductive spikes that go higher than that. You'd probably be okay if you never left it plugged into the cigarette lighter when the car was turning on and off.

So the design seems a bit dubious.

Most of the chips that come up in the "LC51" datasheet search use fixed-frequency oscillations in the tens to hundreds of kHz. Maybe if I hook this up to a voltage source I could observe its oscillation frequency.

I don't know the inductance of the inductor, so I don't know how fast its voltage would ramp up and down, but at 1000 mA difference between input and output you would drain the output 100-µF capacitor from 5 V down to 0 in 500 µs, and from 5 V down to 4.5 V (which I am inferring is probably about the limit of the acceptable voltage deviation, although I don't remember from the USB spec) in 50 µs. So it seems like a fair bet that the chip's designed oscillation frequency is at least a few tens of kHz to keep the thing from falling over without a larger output cap.

The inductor presumably needs to be able to ramp its current up and down between 0 and 1000 mA over a similar time period — otherwise it wouldn't be able to respond fast enough to things being plugged and unplugged. So a sensible inductor size would be in the neighborhood of 50–100 µH. When the chip's switched output pin is floating, the inductor has to maintain some significant fraction of its ≈1000 mA against ≈4.7 volts (the 5 volts of the output capacitor minus the presumably 300 mV of the rectifier) for ≈50 µs, which would require at least 250 µH or so. When the

chip's switched output pin is high, the inductor's current would have to be able to ramp up from 0 to ≈1000 mA when driven by ≈7 V (12 V - 5 V) in ≈50 µs, which would require no more than 350 µH. In practice the frequency is probably considerably higher, so the time period is probably more like 5–10 µs, with a correspondingly smaller inductance.

(But there's no guarantee that they did in fact use a sensible inductor size.)

If the inductor were 100 µH, the LC time constant $\sqrt{(LC)}$ would be 100 µs. I think the resonant frequency is $1/(2\pi\sqrt{(LC)}) \approx 1.6$ kHz, so it is surely much lower than the frequency being used.

Presumably its feedback pin is divided down internally and compared to a temperature-compensated internal bandgap reference; this feedback (plus rectifier and capacitor leakage currents) is the only open-circuit load on the power supply, and so it might be designed to be a heavy enough load to maintain regulation when nothing is plugged into it. The chip, though it's a through-hole DIP, is a super tiny through-hole DIP, and although its package seems like ceramic rather than epoxy, it's probably not capable of dissipating more than ¼ W, which implies a maximum current of 50 mA into the feedback pin, and probably more like 5 mA.

Prospects for hacking

These are probably obvious, and maybe a bit goofy given that the thing is apparently broken.

Presumably the duty cycle of the chip's switched output is variable from under 10% to at least 50% (probably 100%), and this is controlled by the voltage detected on the feedback pin. By cutting a trace on the board and hooking up the feedback pin to the center tap of a pot between the output and ground, it should be straightforward to get a more or less regulated, temperature-compensated output from 5 V up to at least 50% and probably 95+% of whatever the input voltage is; I've inferred that the current through the internal feedback divider is probably several milliamps in order to satisfy the minimum-load requirement of the internal circuitry, so the current through the pot would probably need to be tens of milliamps if you want it to be very precise. But that would imply a pot < 1kΩ, which would be hard to find.

Lower output voltages would require pulling the feedback output up toward some higher voltage. The simplest approach might be to totally fake the feedback — hook up a variable voltage divider across the input that lets you generate a voltage somewhere around 5 V to feed to the feedback pin, and use an external resistor to fulfill whatever minimum-load requirements the switcher has. That might work, but depending on the feedback scheme, it might always rail at Vout = 0 or Vout = Vin. Including some of the real output voltage in the summing junction should make it possible to stabilize it in such cases. Any of these schemes depend on a regulated input voltage, since the divided input voltage is going to be compared against its internal bandgap.

You might be able to get a regulated output current rather than voltage by the usual trick of floating the circuit's ground at a distance below its filtered output determined by a current-sensing resistor in series with that filtered output. For example, if you want to set the

output to 100 mA, you'd use a 50-Ω resistor. I'm not sure the circuit will start up properly in this situation, though, and because there's 5 volts across the resistor, it's potentially going to burn up a lot of power.

Soldering in a bigger diode would probably enable higher output powers, although without current negotiation, compliant USB devices will not really take advantage of that. Amplifying the output with a large output transistor to get tens of watts of output power might be a fun thing to try, too, though at the cost of losing the overcurrent protection presumably included in the chip.

If the output frequency is fixed, a resonant LC notch filter or two across the output should be able to attenuate the EM noise from the switcher by 20 dB or so.

Presumably you should be able to use the chip as a decent class-D audio amplifier by adding the audio input signal into its feedback pin in order to modulate its output voltage, which, as I said above, needs to be able to respond across the whole audible spectrum in order to avoid output overvoltage when you unplug USB devices. It should be able to output 5 watts or more of output at efficiencies of around 90% into impedances in the usual 4–8 Ω range, but you will need an ac-coupling capacitor on the output to avoid a humongous 5-V dc bias on the speaker output.

Depending on what the output frequency actually is, and whether it's fixed or variable, you might also be able to use it as a micropower AM or FM radio transmitter if you bias the audio input into the right range (and add in the appropriate fraction of the filtered output voltage, if necessary) on the feedback pin. Even if the baseband output frequency isn't in the right band, one of its harmonics may be, and by finely adjusting the duty cycle you can finely adjust the harmonics' amplitudes. (See Processing halftoning (p. 915) for some notes on this in the spatial domain.)

Topics

Giving Golang a second look for writing a mailreader (in 2012)

Kragen Javier Sitaker, 2012-12-17 (updated 2013-05-17) (2 minutes)

In 2012, starting again with golang.

• The email address parsing functionality is a little tied into the email system; you apparently have to make a fake header.
• go doc is pretty cool, but looking at the actual library source is even better in most cases.
• Type equivalence is by name, so even if a mail.Header *is* just a map[string][]string, you still can't mutate it. But you can construct a new one.
• for x := range somearray makes x range over the indices of somearray, not its values. You want _, x :=.
• How do you find the length of an array? Oh, Pythonwise, len(array).
• Okay, so suppose I want to do the equivalent of this Python:

```
def main(outbox, inbox):
    important_senders = recipients(outbox)
```

I guess I need to be able to open the outbox file and accumulate stuff in a map. Hmm, isn't there a thing about named return values? How does that work? Oh, (name type, name type, name type) after the argument list.
• How do I open a file? os.Open.
• How do I print messages to stderr and return failure from main? Looks like log.Fatal does that, and even log.Printf prints to stderr. (Presumably fmt.Printf doesn't.)
• Now that I have the file open, how do I get messages from it? Looks like the file is already an io.Reader, so I'm good.
• Hmm, now it seems like I'm suffering from email parsing problems, from apparently trying to parse a header starting in the middle of a subject line. I guess I need to read that message body.
• But hmm, io.Reader.Read(p) reads up to len(p) bytes. How do I make a slice (?) that I can pass in there? (Looks like make([]byte, 16).) Oh, there's an ioutil.ReadAll? That solves the problem. But now I still have no addresses! ... Also related to problems like these: bufio.NewReader(os.Stdin) and r.ReadString("\n").
• Oh, hmm, this is just message parsing, not mailbox parsing. I still have to do mailbox parsing myself, and provide an io.Reader interface? Well, there's an io.LimitReader in the library.
• Cool, the I/O 2011 talk mentions an SDL binding for Go.
• Looks like the email message parsing chokes on this:

```
To: Kragen Sitaker <kragen>
```

Topics

Running your regular desktop in QEMU?

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

So I've been playing with QEMU, which lets you run a virtual computer inside your normal computer. At the moment I'm using it to create a reproducible development environment on a project I'm working on.

Among QEMU's features is the ability to save a virtual machine snapshot, which includes the entire state of the virtual computer: memory, CPU, even disk. This seems similar to KeyKOS's checkpointing facility, although it seems to be a bit slower, maybe to the point of being less useful. (It seems to do all of its I/O before continuing to run, rather than doing some kind of copy-on-write. It seems like fork() might be sufficient to get good copy-on-write performance.)

(In one case, a VM snapshot of a 128MB VM took 53MB of space.)

But suppose you ran your normal GUI session inside of QEMU. Maybe every few minutes, you could do a backup of your live session to a server somewhere nearby.

Benefits

If you do this, you can transport your GUI session from one machine to another --- something the term "VNC", an abbreviation for "Virtual Network Computer", promised but never delivered. If your machine ever crashes or gets stolen, you can restore from the previous checkpoint; sometimes this might be worth doing even if only a single application within it has crashed. And you can have a large number of GUI sessions for different users in suspended animation on your disk.

This kind of thing could give users in, for example, an internet cafe, the freedom to really customize their environment. Rather than storing all of their state on a web site, they could store much of it on the servers at the internet cafe itself, as if they owned it. They could install software, keep their files, and so on; and whenever they came into the cafe, their session would be waiting for them, just as it was when they left it.

Problems

There are a lot of times these days where you'll want to run stuff that doesn't run very well inside QEMU: MPlayer, Art of Illusion, anything with SSE, MMX, or 3-D acceleration. I think that's kind of a minor problem, since the data involved in that part of the system (the latest frame of a movie, say) is usually quite transient and easy to recreate.

Also, it's not uncommon these days for a GUI session to fill up gigabytes of RAM, and all of that RAM could in theory change its contents about once a second. So you still might end up copying all or the vast majority of the memory pages during a snapshot.

Topics

Schimmler parallelism asymptotic gain

Kragen Javier Sitaker, 2007 to 2009 (1 minute)

In Dan Bernstein's grant proposal to the NSF from 2001, Circuits for Integer Factorization: A proposal, he writes:

A philosophical note. I always thought that common general-purpose computers were the pinnacle of realistic computational power. Special-purpose computer architectures, such as Lehmer's bicycle chain sieve or Pomerance's Cracker or Shamir's TWINKLE, were at best a constant factor faster. Quantum computers are asymptotically faster for many computations, but it is unclear whether they can actually be built.

I also thought that parallel computing reduced the *time*, not the *cost*, of computations. Ten processors might perform a computation in one tenth the time of a single processor, but they are ten times as expensive, so the cost of the conmputation remains the same.

I was wrong. Schimmler's machine, with m^2 processors, can be built for $m^{2+o(1)}$ dollars, just like a single-processor computer with $m^{2+o(1)}$ bits of memory. It can sort m^2 numbers in time $m^{1+o(1)}$, while the single-processor machine needs time $m^{2+o(1)}$. The cost of the computation has dropped from $m^{4+o(1)}$ to $m^{3+o(1)}$.

I have a hard time believing this, but Bernstein is pretty reliable; maybe I should go back and read more about Schimmler's machine.

Topics

- Performance (p. 3621) (149 notes)
- Parallelism (p. 3616) (8 notes)

Passive ultrasound sonar

Kragen Javier Sitaker, 2016-12-28 (1 minute)

Up to a quarter megahertz, ultrasound attenuates in air at about 20dB/ft/MHz, which is 66dB/m/MHz. The Airy limit is $\sin\theta = 1.220\lambda/D$ for a circular aperture. Mach 1 is 331 m/s.

Could you do passive ultrasound sonar at a reasonable resolution? At 100kHz you have 6.6dB/m of attenuation, so you only have about 20 meters of range where a detectable amount of sound is going to reach you. λ = 3.3 mm, so the Airy limit for a 100mm aperture is 0.040 radians, about 2.3 degrees. You'd need a much larger aperture to get close to the theoretical far-field limit of what 100kHz could resolve.

At a higher frequency, like 1MHz, you only have a couple of meters of range, so you'd have to get super lucky to have an ultrasound source in range. 1MHz in air has a 330-micron wavelength, so your limit would be 4.2 milliradians, about a quarter of a degree, and then you really could see stuff with subcentimeter resolution at a meter of distance or so. But in practice you probably need an active ultrasound source for that.

Topics

Constant-space grep

Kragen Javier Sitaker, 2014-02-24 (3 minutes)

Grep reads an input file consisting of arbitrary-sized lines and outputs those lines which contain a given pattern. Related utilities like mboxgrep and glimpse do a similar job. One thing they all have in common is that when they reach the beginning of a new line (or other record), they do not yet know whether that line will be output or discarded; and they may not know until reaching the end of the line.

Two simple approaches to this problem are to limit line length and to allocate memory dynamically. Limiting line length is practical, but the arbitrary limit produces problems later on when you run into it. Allocating memory dynamically is also practical, but can use arbitrarily large amounts of memory.

But the problem can be solved in constant space, for different kinds of constant space, with different degradations of the usual grep features.

Sequential-access read-only rewindable input

If your only writable memory is small, but the input file is stored and seekable, the best you can do is to keep track of how many bytes you are into the current line, and rewind the input back to the beginning of the line ("record") when you find a match. This requires, in the worst case, one seek for every record in the input file, and two reads of every byte.

If the input file isn't even stored, grep is impossible, because computing the first output line can require an arbitrarily large amount of data to be stored.

Two sequential-access bidirectional temporary files

With two tape drives, you can solve the grep problem as follows, although you lose grep's normal pipeline promptness. Each record from the input is copied into a temporary file T1, followed by its length and a boolean carefully whether it contains a match or not. Then, we read T1 backwards, copying the matching records (in reverse order) onto a second temporary file T2. Since we're reading it backwards, we read the boolean before each record, so we know whether to copy it or not. Finally, we read T2 backwards, copying its contents to the output.

Under some circumstances, the file T2 itself (the backwards grep output) might be adequate as output to the next stage; in those cases, we can make do with a single sequential-access bidirectional temporary file.

Two sequential-access unidirectional temporary files

In this case, we copy the input records into a temporary file T1,

while writing match booleans to a second, much smaller, temporary file T2. Then, we rewind both and read them sequentially and in parallel, using the match booleans from T2 to tell us which records to output from T1.

Topics

Ice pants

Kragen Javier Sitaker, 2017-04-04 (updated 2019-01-22) (17 minutes)

Some people have serious difficulty with some naturally occurring levels of heat. The CEO of Singapore, Lee Kuan Yew, has credited air conditioning with allowing his (tropical) country to develop economically; and in recent years, as I age, I find myself suffering increasing levels of impairment and ill-being from the Buenos Aires summer. But air conditioning has some disadvantages. It requires large, capital-intensive machinery; it consumes a lot of energy to cool large spaces, and a lot of power to cool spaces with a lot of surface area to a hot outdoors; and it isn't portable to many places people would like to go.

Years ago I read a 2007 Mother Jones article by Dennis Gaffney about hypermilers in which the protagonist, Wayne Gerdes, sometimes wears an "ice vest", "which he uses at the nuclear plant when he has to work in really hot rooms," because, says Wayne, "You can drive at 95 degrees [Fahrenheit; 35°] with an ice vest, and it doesn't feel like 95 [35°].... No electricity, no air, no fans."

Also in 2007, my then wife, Beatrice, was suffering from the Buenos Aires summer heat in our house. I don't remember if we didn't yet have an air conditioner in the bedroom, or if it wasn't powerful enough, or if the power was going out, but she took to freezing plastic bottles in the freezer and wrapping them in a towel to snuggle up to at night, a trick which I've used to great effect ever since.

What if you had an optimized, portable version of this? Could you eliminate the need for air conditioning? What would such a design look like?

Wikipedia tells me:

> However, in situations demanding one is exposed to a hot environment for a prolonged period or must wear protective equipment, a personal cooling system is required as a matter of health and safety. There is a variety of active or passive personal cooling systems;[14] these can be categorized by their power sources and whether they are person- or vehicle-mounted.
>
> Because of the broad variety of operating conditions, these devices must meet specific requirements concerning their rate and duration of cooling, their power source, and their adherence to health and safety regulations. Among other criteria are the user's need for physical mobility and autonomy. For example, active-liquid systems operate by chilling water and circulating it through a garment; the skin surface area is thereby cooled through conduction. This type of system has proven successful in certain military, law enforcement, and industrial applications. Bomb-disposal technicians wearing special suits to protect against improvised explosive devices (IEDs) use a small, ice-based chiller unit that is strapped to one leg; a liquid-circulating garment, usually a vest, is worn over the torso to maintain a safe core body temperature. By contrast, soldiers traveling in combat vehicles can face microclimate temperatures in excess of 65 °C and require a multiple-user, vehicle-powered cooling system with rapid connection capabilities.

I think I want one of these for daily use.

Rough calculations and background

An adult human normally must consume about 2000 to 2500 kcal per day (8.5–10.5 MJ) to remain normally active without losing weight. Only athletes or people with very extremely demanding jobs

eat significantly more than this. Essentially all of this energy is converted into heat inside our bodies — either by bacteria in our intestines, by the various metabolic activities of our brain, liver, muscles, and other tissues, or by damping work done on our bodies. And there are no other significant thermal influxes under normal circumstances.

These last two points deserve some elucidation. The other day, I walked up 12 flights of stairs because the power was out in the elevator, about 50 meters. I weigh about 110 kg, so this amounted to about 54 kJ of mechanical work. Roughly speaking, as I walked up the stairs, my leg and butt muscles converted about 220 kJ of glucose and other metabolic fuels to exhaust — carbon dioxide and water — and converted about three-quarters of this directly into heat. The other one-quarter, however, was converted into the gravitational potential energy of my body.

So, you could argue, if I kept walking up stairs all day long, only about three-fourths of the calories I had consumed would be converted to heat. I could probably walk up about 900 flights of stairs in a 12-hour day of walking up stairs, at which point I would have done about 4 MJ of mechanical work and gained about 4 MJ of potential energy, and I would have expended about another 12 MJ of heat. I would be very hungry and my knees would hurt a lot.

I would also have reached an altitude of 3750 m above where I started, a bit less than climbing Mauna Kea or Mount Ranier. At some point, I would probably have to come back down, which would convert my gravitational potential energy back into mechanical energy, in my leg and butt muscles. Since my leg and butt muscles are not equipped to convert mechanical energy back into glucose, they would convert it back into heat.

(There are cases where the mechanical energy is eventually dissipated in something that isn't your body — when you go swimming or kayaking, for example, or if you're climbing a stair climber instead of actual stairs. Or if the power comes back on and you take the elevator down instead of the stairs. But these are unusual cases.)

I say there are no other significant thermal influxes under normal circumstances because when you put people in an environment where they cannot reject body heat to the environment, they die, usually in minutes to hours. So the net heat flux in a survivable situation is always from your body into the environment.

"Literature" "review"

Porticool is a brand name for a liquid-CO_2-cooled vest marketed to HAZMAT specialists and other emergency responders, developed on a DHS SBIR Phase I grant, using an open-loop system that releases 500psi gaseous CO_2 from some of the tubes running through the vest, after running it through non-porous tubes as a liquid.

While other cooling solutions (ice vests, cooled air vests and liquid circulated garments) tested have shown favorable responses, none provide the flexibility and mobility afforded by the Porticool PCS. Nor could they compete with the light weight and thin garment size that the Porticool PCS achieved.

This doesn't sound like something you could safely wear all day, if at all ever, because you'd have to be replacing the liquid CO_2 and you'd be constantly at risk of being poisoned by it. (I guess I should

calculate how much CO_2 is emitted, but I think the answer is in the range of a few kilograms or cubic meters per day, therefore liters per minute.)

The DHS published a superficial TechNote on personal cooling systems in 2013. It divides them into "passive systems" (like ice vests) which contain no moving parts and "active systems" which do, for example because they involve a circulating fluid, and therefore need a power source. (I suppose directly applying Peltier coolers would also be "active" and without moving parts, but that wouldn't be practical, so probably nobody does it.)

It mentions evaporative cooling systems with "water absorption crystals"; vests with phase-change material pockets, usually with paraffin, which last up to 2 hours; "gel or ice pack vests", erroneously implied to not be phase-change materials; "ambient air systems", which blow ambient air under your clothes; and "liquid circulating products" with vapor-compression or thermoelectric cooling, or ice, to chill the liquid. It shows a Veskimo ice-backpack-powered cooling vest as its example of this last category. Apparently ASTM F2300 is the testing standard.

In 2014, Adam Savage of Mythbusters and some other guys talked about building a cooling suit. The background is that, when he went to ComiCon in a hot costume, he was trying a gel-vest system made of "Polysorb... like probably diaper crystals" and although "it's a brilliant design" he nearly passed out from heat exhaustion because he didn't have a freezer to freeze it in previously. He said it should have been able to keep him cool for an hour if he had frozen it. He also mentions a liquid-circulating cooling shirt called "CoolShirt" which is sold for (auto?) racing, with "recirculating pumps that are too big"; he didn't think it was up to the job.

The video is a waste of time; it's just three guys talking on microphones for half an hour, and they haven't even built the suit.

The Veskimo cooling vest mentioned in the DHS report runs microtubing through a thin vest. It explains:

> NASA pioneered the use of garments employing circulating chilled liquid in the 1960's to keep astronauts cool during space walks. The design and construction of these systems are well documented. Systems of similar construction are currently in use by military personnel in aircraft and armored land vehicles. These systems are very expensive because they use compressor-based refrigeration units to chill the circulating liquid coolant. By substituting ordinary ice to chill the water, the cost and complexity of the system is greatly reduced, making Veskimo Personal Cooling Systems affordable, yet still truly effective.
>
> ...The Veskimo Personal Microclimate Body Cooling Vest is made from lightweight breathable mesh fabric that will not inhibit the evaporation of perspiration, and has a thickness of less than one-quarter inch, so it fits easily under any close fitting garment or protective gear. Its zippered front and adjustable-length elastic side straps make it easy to take on and off and adjust for best fit, comfort and the desired degree of tube-to-skin contact.

Their page also mentions "evaporative garments" usually use sodium polyacrylate crystals, like diapers, flowerpots, and maxi pads. Because apparently the Veskimo people aren't scientifically illiterate like DHS employees, they class ice vests with other "phase-change garments". They say their ice "backpack or cooler" is "typically enough for 4 hours or more".

More details on the Veskimo system:

> Approximately 8 pounds of ice can fit in our 4.4 Quart Hydration Backpack, and as much as 16 pounds in the 9 Quart Cooler. The useful heat capacity of 8 pounds of

ice is approximately 400 Watt-hours, and 800 Watt-hours for 16 pounds of ice. Studies performed by NASA and the US Military conclude that 100 Watts of cooling power is effective in maintaining body core temperature in all but the most extreme heat conditions. The Veskimo Personal Microclimate Body Cooling Vest is capable of providing over 100 Watts of body cooling. If the system and the user are well insulated from the external environment, the useful cooling duration is approximately 4 hours (400 Watt-hours / 100 Watts) for the Backpack and 8 hours for the 9 Quart Cooler System at 100 Watts cooling output. If the user wears a lightweight windbreaker-type jacket over the vest to minimize loss of cold to the atmosphere (which we recommend), these are reasonable estimates for the system's cooling performance. Some users have reported even longer cooling duration because they had their system adjusted to provide less than 100 Watts of cooling.

They run their pump on 4 watts on a 12-volt Li-ion battery. Their list price is US$1116 for the Veskimo vest and backpack, and on top of that they're out of stock, so a homemade substitute would be worth considerable work.

Pants or shorts

A possible alternative to an ice vest would be ice pants, which would be less of a weight burden (since only your legs would carry the weight, not also your back) but perhaps more cumbersome. Your legs are similar in surface area to your torso and have very substantial blood flow to them.

Peltier devices

A 100-W Peltier device is typically 10 A at 12 V and costs about US$15 retail, but you need to run Peltier elements at substantially less than their maximum rating to get reasonable efficiency; at $\Delta T = 0$ and $I/I_{max} \approx 0.3$, $Q/Q_{max} \approx 0.5$ and CoP ≈ 3.0 (that is, 3 joules of heat are removed from the cold side of the reservoir for every joule of electrical energy dissipated). So maybe the 100-W device can give you 50 W of heat rejection at 4 A (and, I suppose, very close to 12 V, since thermocouples are closely approximated as constant-voltage devices.) As ΔT rises, the CoP falls, eventually past 0 as the resistive heating effect of the current outweighs the Peltier effect.

In the case of a human in an environment that is only mildly hostile — say, 30°–40° — $\Delta T \approx 0$, so this might be a reasonable region to work in. (XXX is it actually going to be adequate to keep your skin at 30° or 35°, or is that going to result in dangerously low heat transfer rates from the body core? What skin temperature is adequate? Presumably it's about the same as the bearable air dewpoint.) CoP ≈ 3.0 means that you only need ≈30 W of battery power to reject 100 W of heat to the environment, which means that each hour of cooling consumes about 110 kJ of battery. (The energy cost of pumps and fans was 4 W in the Veskimo case, which is small compared to the Peltier devices, though not insignificant.)

An 18650 cell typically weighs 47 g and costs about US$10, and is commonly 2000 mAh and 3.7 V, for an energy capacity of 27 kJ, so you'd be emptying on the order of 4 18650s per hour, weighing 190 g. This compares very favorably to the 1080 g of ice (100 W / (333 kJ/kg) = 300 mg/s = 1081 g/hour) that you would need to carry to absorb the same amount of heat by melting. Ice is, of course, substantially cheaper; equaling the four-hour 100-W capacity of the 4-kg-of-ice Veskimo unit described above would require 16 18650s at

a price of about US$160, plus another US$100 or more for the Peltier modules and other materials. But weighing only 750 g rather than 4 kg could be a decisive advantage for the battery-driven unit under normal weather conditions.

However, the humans require for comfort not just a low temperature, but a low dewpoint; cooling the air next to the skin reduces the temperature but not the dewpoint, at least until condensation begins. A dewpoint of 15–20° is required, which I think means that the cooling vest or pants actually need to reach a *temperature* of 15–20°. With an external temperature of 35°, that's a ΔT as high as 20°, so according to Meerstetter's guide we can't expect a CoP of better than about 1.2, and that at (again) I/Imax ≈ 0.30, at which point Qh/Qmax ≈ 0.35 and Qh/Qc ≈ 1.75, so Qc/Qmax ≈ 0.20. So we'd need 5 "100W" Peltier elements to reject 100W of heat, and we need 83 W of battery power, emptying 11.1 18650s per hour, for a weight of 520 g of batteries emptied per hour. This is still almost twice as dense as the ice, but much pricier; your 4-hour unit now needs US$440 of batteries in it. Also, you have the potential safety issue of carrying 1.2 megajoules of highly volatile lithium batteries strapped to your body.

Under extreme conditions like those described in the Wikipedia article (as high as 65°) the ice-based system would continue to function exactly as well, while the Peltier-based system's efficiency would degrade enormously, and it might cease to work entirely. But perhaps under normal weather conditions the Peltier approach might work better.

Topics

Compressing a screen update with a tree of dirty bits

Kragen Javier Sitaker, 2017-06-21 (1 minute)

Suppose we build up dirty bitmaps of screen areas starting from some leaf size?

If we use the 8×8 pixel tile used in JPEG as our basic unit, and the 64-bit size of an amd64 register as our treenode size, then the next level up the tree is 64×64 pixels, then 512×512. My screen is 1920x1080 pixels, so to fill it, you need just under four such tiles from left to right and just over two from top to bottom.

So to describe a screen update, you could use twelve 64-bit words to list which parts of the screen you want to update, then another 64-bit word for each of those up-to-768 areas of 64×64 pixels you want to update, and then probably just pixel data for each 8×8 area. The worst case for small updates is 13 64-bit words of overhead plus an entire 8×8 area (128, 192, or 256 bytes); for large updates, the 6240 bytes of 1 bits at the beginning don't amount to much, one bit for every 41 pixels. (And in RAM of course you can keep those 6240 bytes always allocated.)

I'm not sure how useful this is now that we're often repainting the whole screen every frame.

Topics

Time series data type

Kragen Javier Sitaker, 2016-08-26 (3 minutes)

In work with financial market data, I often want to do computations on time series data. I'd like to be able to specify the computations in a way that isn't, for example, coupled to a sampling rate. I'd like to be able to specify a computation once and then efficiently perform it on any of the following, without editing the code:

- Real-time data as it becomes available.
- Regularly sampled historical data.
- Irregularly sampled historical data on my laptop.
- Irregularly sampled historical data on some Amazon EC2 nodes.
- Conservatively approximated historical data.
- Specific randomly generated data.
- Specified mostly continuous (though not necessarily piecewise-constant) functions.
- Probability distributions of data.
- An incremental variation of a previous dataset.

Also, I want to be able to interactively create models and visualize the results.

I'm not sure I will be able to achieve this, but I can achieve most of it.

There are three main time-series data types I am dealing with:

- Partial functions, which map some subset of the timeline to values such as $301.50, Wednesday, false, or 3%, with at most a countable number of discontinuities.
- Finite mostly-continuous subsets of the timeline, which can be thought of as boolean partial functions that happen to be defined everywhere.
- Events, which are countable numbers of points on the timeline.

There are functions between these data types. The discontinuities of a function or the beginnings and endings of a subset are events; given start events and end events you can create a subset; a boolean function can be converted to a subset of the timeline, and you can extract the domain of any function; you can restrict a function to be valid only within a subset; and you can discard events outside a subset.

Operations on the values mapped to by partial functions can be lifted to operate pointwise over those functions, intersecting their domains.

Finally, you can coalesce partial functions, SQL-style.

All of the above operations are nearly memoryless. However, there are also a set of *causal* operations available. The simplest is simply a lag, applicable to functions, subsets, and events; but others are also available. They are defined as a generalization of integration (in some sense numerical integration rather than symbolic integration), with an arbitrary semigroup operation used in place of addition, and another arbitrary lifting operation used in place of multiplication by a step size. This permits, for example, computing a window of the last ten

minutes of updates.

Requiring the "multiplication" operation to be a semigroup operation (i.e. associative) permits efficient parallelization and incrementalization.

Topics

Compressed sensing microscope

Kragen Javier Sitaker, 2016-10-06 (7 minutes)

By using a fairly rigid aperture grille with sparse randomly placed holes of different sizes, including a large number that are very small, it should be possible to do very high resolution microscopy, including subwavelength near-field light microscopy of flat objects, using only low-resolution, low-quality lenses or mirrors to focus light.

You put the grille on top of the sample, in contact with it, and shine light through the sample and the grille, taking a picture of the starfield pattern that comes through. Where a hole is over a clear spot in the sample, a white point of light will show; where it's over a transparent red spot, a red point of light will show; where it's over an opaque part, no light will show. The holes are much smaller than the camera pixels, but they are sparse enough that typically only one hole or less makes a significant contribution to each camera pixel. The consequence is that moving the grille by a hole diameter or so scans many such high-resolution pixels over the sample, and by taking many such frames, eventually the whole sample can be covered redundantly by many holes.

It's necessary to estimate the relative positions of the holes to deep subpixel resolution, and it may be necessary to do this simultaneously with estimating the image of the surface, a problem similar to the simultaneous localization and mapping (SLAM) problem in robotics. Alternatively, if the grille is rigid enough, it may be possible to estimate the hole positions ahead of time, using, for example, a known microscopic image or many high-resolution photographs of the grille.

Using a variety of different sizes of holes in such a grille is a way to reduce the difficulty of this estimation problem. The larger holes provide a blurrier image, but their positions are much easier to estimate to the desired precision, and the blurry image helps in estimating the position of the smaller holes.

A very practical way to produce such a grille is by perforating a thin metal surface, such as gold leaf; aluminum foil peeled from a gum wrapper; the aluminum coating on metallized Mylar/boPET or polyimide, such as a discarded potato chip bag; or the silvering of a first-surface mirror. A very practical way to perforate such a metal sheet is with a short-lived arc, triggered by bringing an electrode close to the surface; the hole diameter, if not its position, can be controlled fairly precisely by controlling the energy released in the arc, which can be measured fairly precisely by measuring the charge and voltage loaded onto a capacitor before the discharge.

Gold leaf is typically 0.2 microns thick; Mylar is commonly 10 microns thick, but only about 0.5 microns of that is the metallized film, sometimes as little as 0.1 microns. Regular kitchen aluminum foil is on the order of 20 microns thick. Vaporizing a 1-micron-diameter hole in an 0.5-micron-thick metal film requires vaporizing about 0.4 $(\mu m)^3$ of metal, which at 2.7 g/cc for aluminum works out to about a picogram. Solid aluminum's specific heat is 24.20 J/mol/K, its heat of fusion is 10.71 kJ/mol, its heat of vaporization is 284 kJ/mol, and it boils at 2470°, so we're looking at

(2470 - 20) K 24.20 J/mol/K + 10.71 kJ/mol + 284 kJ/mol = 354 kJ/mol, and its atomic weight is about 27.0 g/mol, so that's 14 kJ/g = 14 nJ/pg = 14 nJ per hole. That's the energy of a 280 pF capacitor at 10 V.

If the capacitor's capacitance were to increase or decrease by 1% (for example due to temperature, ferroelectric, or soak effects), which is a typical precision for low-capacitance capacitors like this, that would increase or decrease the diameter of the hole by about ½%. It should be straightforward to measure the charge deposited on the capacitor during the charging process to within about 0.1%.

I say it's probably not practical to control the location of the hole precisely because you need to bring the electrode close enough to the foil to provoke a dielectric breakdown of the air; for 10 V to be adequate, for example, the distance needs to be about 3 microns. But if the tip has a spherical radius of 3 microns, then a corona discharge will be occurring around the tip at the same time. This seems undesirable to me for a variety of reasons (uncontrolled loss of energy, ionic erosion) so it would be better to make the tip diameter much larger than this. But this means that the position of the arc will be uncertain to within several microns, controlled by small asperities on the tip or on the foil or by stray ions that wander by. You're depending on the positive feedback of the arc itself to channel all the energy into a small area, but that same positive feedback creates unpredictability.

Using lower voltages would help if it were possible, but it probably is not practical; Paschen's law has its minimum for nitrogen at about one torr cm and about 300 V. An atmosphere is 760 torr, so we reach one torr cm at about 13 microns. Using higher voltages makes the lateral uncertainty larger.

XXX now that I know about Paschen's law I need to rethink the above for a higher voltage.

Coloring the grille black, for example with a layer of carbon black deposited by smoke, should reduce stray light contamination which will damage SNR and require algorithmic rejection after the fact. This should be especially helpful for imaging using reflected light rather than transmitted light.

This approach doesn't seem like it would offer any advantages for electron microscopy, and without very exotic materials, it probably doesn't extend very far into the ultraviolet.

Doing telescopy rather than microscopy with this approach probably will not work very well, because diffracting light through small holes like that will lose a lot of information about which direction it came from. But in telescopy, the whole point is to determine which direction it came from. A more purely diffractive approach, where the incoming light diffracts through a random aperture grille and then falls on a focal plane with no intervening lens, might work better.

Topics

- Opacity holograms (p. 3607) (5 notes)
- Sparks (p. 3724) (4 notes)
- Microscopy (p. 3583) (3 notes)
- Slam

Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias

Kragen Javier Sitaker, 2019-04-10 (updated 2019-04-16) (2 minutes)

The Apollo missions left six nylon flags on the moon, which are probably intact today. They're about 1.5 meters long by 0.9 meters tall, meaning that the stripes are 69 mm tall. The first flag was blown over during the takeoff of the lunar lander. The LRO in 2012 showed that the three flags it imaged remain intact and erect.

What would it take to get 500-mm resolution from a terrestrial telescope? Suppose we can use 300-nm blue light. The Airy limit is $\sin\theta = 1.220\lambda/d$ for a circular aperture, and the moon is 384 megameters away, so our angular resolution needs to be about 1.3 nanoradians. This gives the telescope aperture diameter $d = 280$ m.

The largest single-aperture optical telescope currently is the Gran Telescopio Canarias, which is 10.4 meters in diameter.

In the last 30 years, substantial progress has been made on long-baseline optical interferometry, and there is one such telescope in the US with a sufficient baseline: the Navy Prototype Optical Interferometer with its 437-meter baseline, which has a resolution of a few milliarcseconds. This is unfortunately not quite good enough: 1.3 nanoradians is about 0.27 milliarcseconds. The Very Large Telescope in the Atacama, the first telescope to image an etrasolar planet, has comparable resolution, down to a single milliarcsecond. But these telescopes are designed to image bright objects such as stars, not dim objects such as points on the moon.

Topics

Resistor assortment

Kragen Javier Sitaker, 2018-06-17 (4 minutes)

The electronics shop around the corner from GM Electronics sells resistors in quantity 10 and up, for AR$8. https://www.digikey.com/product-detail/en/stackpole-electronics-inc/CF14JT10K0/CF14JT10K0CT-ND/1830374 is a typical resistor: a 10kΩ ¼W 5% axial through-hole resistor. Its price is 10¢ in quantity 1, 4¢ in quantity 10, down to 0.7¢ in quantity 1000 (US$7 for the 1000).

40¢ is AR$10, so AR$8 is a reasonable price in quantity 10, slightly cheaper than Digi-Key actually.

I think I should ask for:

```
10 1Ω   $8
10 2.2Ω $8
10 4.7Ω $8
10 10Ω  $8
10 22Ω  $8
10 47Ω  $8
10 100Ω $8
10 220Ω $8
10 470Ω $8
10 1kΩ  $8
10 2.2kΩ $8
10 4.7kΩ $8
10 10kΩ  $8
10 22kΩ  $8
10 47kΩ  $8
10 100kΩ $8
10 220kΩ $8
10 470kΩ $8
10 1MΩ   $8
10 2.2MΩ $8
10 4.7MΩ $8
10 10MΩ  $8
```

That's 22 separate values, totaling AR$176.

I should probably also include some other basic components: opamps, microcontrollers, voltage regulators, capacitors, transistors.

Opamps: the most popular opamps on Digi-Key are the TI TSV321RILT (US$0.58), the TI LM2904DR (US$0.39), and the TI TL072CDR (US$0.62). I don't recognize any of those names. The most popular non-TI opamp is the ST LMV321ILT (US$0.51), which I assume is a TI LM321 clone. So maybe I should ask for an LM321 and expect to pay AR$15 for it. The LM741 is the one Electrocomponentes says is standard.

Digi-Key's most popular µC is STILL a PIC12, a 74¢ one. But the top three PIC12s are followed by a $4.34 Cypress CY8C4245AXI-483, whatever the fuck that is (apparently a Cortex-M0), then an $3.56 STM32 (also a M0), and then finally a $12.35 ATMega2560. All the cheapest µCs are AVRs, though — an ATTiny5 for US$0.17, an ATTiny102 for US$0.30, an ATTiny10 for

US$0.34. So if I don't want to hassle with PICs, I should probably ask for AVRs. Electrocomponentes does have ATTiny11s, 13s, 2313s, and ATMega32s (which I guess are ATMega328s). ATtiny13s cost US$0.40 at Digi-Key.

Voltage regulators: probably I want programmable ones, plus maybe some 7805s. The top pick among linear regulators is the US$0.72 LM317 with 350k in stock. Even more popular among switchers is the US$0.70 EZBuck AOZ1280CI from a company I've never heard of. No boost regulators seem to be super popular.

Capacitors: I should probably get a bunch of 0.1µF ceramic bypass caps. These seem to cost about 10¢ apiece in quantity 10 and 5¢ apiece in quantity 100.

For MOSFETs, I should probably get some 2N7000s or similar. These seem to cost about US$0.51 in quantity 1 (9.4¢ in quantity 1000), and they're good to 60V, 115mA. Also, for higher power, maybe some IRF530s (100V) and IRF630s (200V) and their P-channel partners IRF9530 and IRF9630.

For BJTs, the most popular and cheapest are currently apparently something called an MMBT3904, a 10¢ part good to 40V, 200 mA, with a PNP counterpart called the MMBT3906 or PMBT3906. An order of magnitude faster are the 47¢ Rohm 2SC5662T2LPs, with fT of 3.2 GHz, good to 11V, 50 mA. There are also the 12¢ BC846 and BC849, NPN 65V and 30V 100mA transistors, and their PNP counterparts the BC858 and BC856, more or less; but these are only good to 100MHz. The traditional BJT is I think the 2N3904 (200 mA, 40V) and its PNP partner the 2N3906.

Oh, I guess I should get some diodes: 1N4007 (the big 1N4001) and 1N4148 at least, and maybe some Schottky 1N5819s. Maybe 100 of each. Maybe not from the place that doesn't guarantee their semiconductors though.

Connectors for wires are called "borneras"; for pins are "conectores".

Topics

• Electronics (p. 3430) (138 notes)
• Pricing (p. 3646) (89 notes)

Some thoughts on SDF raymarching

Kragen Javier Sitaker, 2019-11-11 (updated 2019-12-10) (31 minutes)

After writing Interval raymarching (p. 1342), I just wrote my first raymarcher with signed distance functions ("SDFs"), intentionally a pretty minimal affair. It was a lot quicker to write than My Very First Raytracer, taking about an hour and a page of Lua to initially get working, rather than all night and four pages of C. Some of that difference is being able to build it on top of Yeso, so it doesn't have to include code for image file output; some of it is that Lua is a bit terser than C; some is that the raymarcher doesn't support color; half a page of it is the input file parsing in the C raytracer; but most of the reason is that a minimal SDF raymarcher is simpler than a minimal Whitted-style raytracer.

Disappointingly, although it does manage to do full-motion video, it's a great deal slower than the precise raytracer, although I haven't added specular reflections, lighting, color, or texture to it yet. I don't yet know if that's because it's an iterative approximation method or because LuaJIT is producing somewhat suboptimal code. In this very simple scene, it's only doing an average of around 9.5 SDF evaluations per pixel, which is comparable to the number of intersection tests the precise raytracer needed per ray; this weighs on the side of blaming LuaJIT.

Of course it's a bit silly to be rendering real-time 3-D graphics on the CPU in 2019, even if your GPU *is* just a shitty little Intel Gen8 (see Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033)), but it's still enough experience to provoke me to write a bunch of things.

Rounding corners and inexact SDFs

In iq/rgba's notes on the topic, he frequently mentions the importance of using exact SDFs, which provide you with the exact distance to the nearest scene object, rather than inexact SDFs (I think the original sphere-tracing paper calls these "distance underestimate functions"), which merely provide you with a lower bound. He explains that this is important because using correct Euclidean SDFs will speed up the raymarching process dramatically.

Now, this may be true, but unfortunately many of the attractive modeling features of implicit surfaces involve transforming functions in ways that do not preserve SDFs' exactness. In particular, union (minimum of SDFs) is exact, but intersection and difference (maximum of SDFs and of an SDF and a negated SDF) are not exact. The SDFs generated by other very appealing features, like surface perturbations, bounding volume hierarchies (including IFS rendering), smooth union, and twisting space, are also usually inexact.

More unfortunately, other attractive modeling features of SDFs — such as the ability to round corners by subtracting a constant, or the ability to compute a constant-thickness shell by taking the absolute value and subtracting a constant — depend on the SDFs'

exactness. Consider a sphere whose SDF is $\lambda\vec{p}.|\vec{p}-\vec{c}_0|-r_0$ and another whose SDF is $\lambda\vec{p}.|\vec{p}-\vec{c}_1|-r_1$. Each of these is exact. If we take the CSG difference as $\lambda\vec{p}.|\vec{p}-\vec{c}_0|-r_0 \vee r_1-|\vec{p}-\vec{c}_1|$ (where $x \vee y$ is the maximum of x and y) such that the first sphere has a second-sphere-shaped bite taken out of it, we get an inexact, lower-bound SDF. It's straightforward to see that attempting to round the sharp corner where the cut penetrates the surface by adding a constant will not work; addition distributes over maximum and minimum, so $(a-k_0 \vee k_1-b)+r = a-(k_0-r) \vee (k_1+r)-b$. That is, all the level sets of this SDF can be reached by increasing one radius while decreasing the other. So the hoped-for rounded edge at the intersection line never appears.

(I say "it's straightforward" but it surprised me when I saw it on the screen.)

A brute-force solution to this problem is to resample the inexact-SDF-generated geometry into some kind of numerical approximation that admits efficient exact SDF computation.

Automatic differentiation

I mentioned autodiff briefly in Interval raymarching (p. 1342), but now that I've actually written an SDF raytracer, I feel like I have a better understanding of the situation.

With SDFs you don't easily know which object you've hit. So if you want to do something that depends on the surface normal, such as Lambertian lighting, you don't have a straightforward way to find it. Moreover, even if you do have a way to find which underlying primitive you've collided with, its SDF $d(\vec{p})$ doesn't immediately tell you how to find its normal. You need the *gradient* of the SDF, $\nabla d(\vec{p})$. Autodiff can give you that, at the cost of only doubling the computational cost, rather than quadrupling it, which is the cost of the standard approach.

However, as I said in Interval raymarching (p. 1342), using affine arithmetic might actually work better than the instantaneous derivative in this case, in order to avoid aliasing artifacts. iq/rgba has shown compelling demonstrations of how using a too-small box to estimate the gradient can cause aliasing.

(If your final top-level SDF is a tree of $\wedge$ and $\vee$ operations, where $a \wedge b$ gives you the lesser of a and b while $a \vee b$ gives you the greater (since real numbers have a total order), you could very reasonably trace the final result of the $\wedge/\vee$ back down the tree to its origin, finding the non-CSG surface it originated from. If you retained the computed values, you can do this in linear time in the length of the path you trace, which is probably faster than annotating each value with its origin on the way through this tree, effectively applying the $\wedge/\vee$ operations to (d, id) pairs rather than just d values.)

To me, one of the appealing aspects of successive-approximation algorithms like SDF raymarching and any kind of Monte Carlo simulation is that they can be used as "anytime algorithms" in order to guarantee responsiveness at the expense, if necessary, of quality (see Anytime realtime (p. 803) and Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) for more on this). My current implementation does not do this. How can I get this in practice?

Using autodiff, you could perhaps "subsample" the SDF

evaluations of the final image, for example tracing a single ray through the center of a 5×5 pixel square; at the final point of contact, rather than just extracting the gradient of the SDF with respect to the (x, y, z) scene coordinates, you could carry the autodiff computation all the way through to the (r, g, b) color value and then compute its Jacobian with respect to the (u, v) pixel coordinates on the screen. This Jacobian gives you a color gradient value at the center of that 5×5 square, and so you could perhaps get an image that looks like a badly compressed YouTube video frame, which is considerably better than you would get by just downsampling an image from 640×360 to 128×72 (same number of samples) or 221×125 (same amount of data).

However, running on the CPU, you can *adaptively* subsample. If you reached the surface in a small number of SDF evaluations, you didn't pass close to any other surfaces, and the surface isn't sharply angled to the ray where you're hitting it, so you can probably get by with fewer samples.

Propagating SDF values to neighboring rays

Even without autodiff, computing an *exact* SDF $d(\vec{p})$ at some point $\vec{p}$ tells you a great deal about its values in the neighborhood of $\vec{p}$. At any displacement $\Delta\vec{p}$ in any direction, we know $d(\vec{p} + \Delta\vec{p}) \in [d(\vec{p}) - |\Delta\vec{p}|, d(\vec{p}) + |\Delta\vec{p}|]$, because the highest it can be is if $\Delta\vec{p}$ follows the gradient away from the object (and the gradient remains constant over that distance), and the lowest it can be is in the opposite direction. (No such pleasant bound holds for inexact SDFs, neither the upper nor the lower bound.)

(Here by $|\cdot|$ I mean the Euclidean L_2 norm, as before.)

Still, though, even with an inexact SDF, we know that an entire sphere of radius $d(\vec{p})$ around $\vec{p}$ is devoid of, as they say, "geometry". This means that if you compute a non-tiny value from an *exact or inexact* SDF for a ray shot from the camera, you have computed a bound for a whole view frustum around that point; a single SDF evaluation is enough to advance the whole wavefront around that point up to the bound of a sphere around that point, at least the part of the wavefront that has successfully *entered* that sphere. It may be worthwhile to use a conservative approximation of the sphere, such as a ball made from a weighted sum of the L_1 norm (whose balls are octahedra) and the L_∞ norm (whose balls are cubes). I think there is a cheap weighted-sum norm whose balls are these irregular polyhedra with 32 triangular faces — "frequency-2 geodesic spheres" in Buckminster Fuller's terminology, except that the underlying polyhedron whose triangular faces were subdivided into four triangles is an octahedron, not a dodecahedron.

You can build a Z-buffer and pick arbitrary points in it and evaluate the SDF at them; each SDF evaluation digs a big crater in the neighboring Z-buffer values, but only those values that are initially within the ball defined by that SDF value.

A fun way to do this might be to start with a very-low-resolution Z-buffer (3×2, say), then repeatedly double its resolution, perhaps using pairwise min to compute the newly interpolated pixels. Each doubling, you tighten the termination threshold for SDF iterations so that it's comparable to the new pixel resolution, and iterate over all the Z-buffer pixels until you've hit that termination condition on

each of them. This might be able to keep the total number of (primary) SDF evaluations per pixel down to 1–3, instead of the 9.5 I'm seeing or the 100–1000 commonly seen in the demoscene. If this can be combined with the adaptive subsampling mentioned above, it should be possible to get in the neighborhood of 0.1 SDF evaluations per pixel.

I understand that on the GPU it's dumb to try to do things like that because communication between pixels is super expensive. On the CPU, though, it might be a sensible thing to do.

A *much simpler* way to propagate SDF values to neighboring rays

The above algorithm with its various resolutions of Z-buffers and so on would seem to require a lot of memory management complexity. But one of the appealing things about both raymarching by sphere tracing and Whitted-style raytracers is the simplicity of their memory management: every ray is traced totally independently, though the process of tracing a ray may result in tracing some more rays, recursively.

It occurred to me that this recursive structure is potentially a good way to handle the multiple resolutions. Initially trace a cone that's big enough to encompass, say, a 16x16-pixel region, by marching a ray down its center. When this ray encounters a low enough SDF that the corners of that region are, say, closer to objects than they are to the ray you're marching, split it up into four rays, one marching down the center of each 4x4 quadrant, with a narrower cone; trace one of them at a time. When one of those rays encounters a low enough SDF that the corners of its region are maybe closer to geometry than they are to the ray, break down into 2x2 quadrants, which break down into pixels. And when one of those rays reaches its destination, you invoke a pixel-found callback, or store a value in the single, full-resolution Z-buffer, and backtrack to complete the recursion.

And of course you don't really start at 16x16. You start at 512x512 or 1024x1024 or 2048x2048, whatever your screen is.

The amount you advance is less than with standard SDF sphere tracing, because you want to make sure that you aren't smashing any of the corners of your current pixel region into an object. The simplest bound is to advance from point p by $d(p) - r$, where d is the SDF and r is the distance from p to the furthest corner of the projection of the current pixel region onto a sphere of radius $|p|$ (assuming the eye is at 0). But this is a pessimistic bound; the precise bound is the positive solution for e of the quadratic equation $(e + |p| - |p| \cos\theta)^2 + (|p| \sin\theta)^2 = d(p)^2$, where θ is the angle from p to the corner of the region, as seen from the eye. When the angle is large, this could give a significantly better bound, but when the angle and the SDF are small, the improvement in the bound is also small in absolute terms, and the angle is almost always small. However, this small improvement is probably usually close to a factor of 2, and my guess is that it might cut the recursion depth needed by a factor of 3 or so in typical cases. So the tighter bound might be worth the extra complexity, or it might not.

This doesn't fit well into a fragment shader, which may be the reason I haven't seen it discussed even though it seems obvious in retrospect, but it seems like it should fit beautifully into a CPU. It's

somewhat less optimal than the more complex approach described earlier with the multiple Z-buffers, but it's also enormously simpler, and it's embarrassingly parallel.

Multithreading and SIMD

On this three-torus scene, I'm getting 5.5 frames per second on my laptop.

I'm not attempting to use SIMD operations like SSE and AVX, and the algorithm is expressed in such a way (with data-dependent iteration bailouts) that I would be very surprised if LuaJIT were finding a way to take advantage of them. I don't think SSE 4.1 (which LuaJIT has) or even SSE 4.2 (which my CPU has) support half-precision 16-bit floats, so probably a 4× speedup for reasonably coherent vectors is the most I can expect from SIMD. (Again, see Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033).)

The CPU also has four cores, so in theory I should be able to get an additional 4× speedup from multithreading, as long as the pixels aren't too interdependent.

So in theory I should be able to get a 16× speedup just by applying brute force, working out to 88 fps (at 320×240).

Automated fabrication

There's another major reason I'm interested in SDFs that's actually not real-time graphics at all: automatic fabrication. Historically manufacturing has largely been concerned with the *geometry* of parts, because steel, fired clay, concrete, glass, and polyethylene — the best materials for many purposes — are sufficiently uniform and isotropic that their geometry is most of what you need to know. (Heat treatment is important, but commonly applied to an entire article; similarly painting, galvanizing, etc. Surface finish is sometimes important, but that's in some sense a question of geometry too. Work hardening, for example from cold forging, is extremely important.) Moreover, for metals in particular, merely achieving a desired geometry is often a difficult and expensive proposition.

So algorithms that make contending with geometry tractable are of great interest.

(I think one of the really interesting possibilities of automated fabrication is actually that we can make things out of nonuniform and anisotropic materials.)

The existing libraries, algorithms, and user interfaces for dealing with 3-D geometry in computers are utter shit. "Sketching", "lofting", "extrusion", "press/pull", "pocketing", "filleting", and so on, are terribly unexpressive; achieving even the most basic compound curves is often beyond their capabilities. The interactive sculpting user interfaces in things like Blender and ZBrush are more expressive, but incapable of handling any demands for precision. The triangular bounding surface meshes and NURBS commonly used as the internal data representation are humongous, bug-prone, and — for any given level of humongousness — terribly imprecise; furthermore, doing topological optimization is basically impossible with them. The available libraries are buggy as shit and crash all the time. Half the time when you export a mesh from one program to import into another it turns out not to be "manifold", which is to say, it fails to

represent a set of solid objects. Voxel representations are, if anything, even worse, but at least they can handle topopt and don't have the fucking "manifoldness" problem.

Christopher Olah's "ImplicitCAD" is an attempt to remedy this situation by using SDFs (and Haskell). I'd like to play with the approach and see what I can get working, but without Haskell.

So, what would it take to import an STL file into an SDF world? How about a thresholded voxel volumetric dataset?

SDFs in two dimensions

Before I ever heard about 3-D raymarching using 3-D SDFs, I read a now-lost Valve white paper about rendering text "decals" in games using 2-D SDFs. The idea is that you precompute a texture containing a sampled SDF for the letterforms you want to use, and in your shader, you sample from that texture, with the built-in bilinear interpolation the GPU gives you. Then, instead of just using the sampled value as a pixel color, *you threshold it to get an alpha value*. This allows you to use the bilinear interpolation to interpolate sharp letterform boundaries in between texels.

Moreover, by thresholding it *softly*, you can get antialiasing. A similar effect comes into play with 3-D SDFs if you stop marching the ray when the SDF falls below about the scale of pixel spacing projected on the surface: the SDF sphere smooths over surface detail smaller than a pixel or so, preventing its high spatial frequencies from aliasing down into lower frequencies on the screen.

A problem with this technique as described is that the bilinear interpolation inevitably kind of rounds off sharp corners where you would want them on the letterforms; to deal with this problem, you can use two or more color channels to approximate different parts of the letterform boundary with smooth curves, which cross at the desired sharp corners. There's an open-source "mSDF" software package for generating these "multichannel SDFs".

I wonder if there's a way to carry the analogy through to 3-D raymarching with SDFs. Perhaps, for example, it could somehow provide a solution to my problem with rounding off edges.

Use in Dercuano

I'm pretty sure now that I can implement SDF-based sphere-tracing raymarching in JS on `<canvas>` to get reasonable 3-D diagrams (see Dercuano drawings (p. 64) and Dercuano rendering (p. 2300)). `<canvas>` has a typed array interface for raw pixel access I used in Aikidraw, so you don't need a DOM call for every pixel you draw. I think I can render raster graphics into that interface pretty easily, fast enough at least for still images.

Specifically, you can call `ctx.putImageData(data, x, y)` on a 2d `<canvas>` drawing context, where `data` is an `ImageData` object made out of a width `.width`, a height `.height`, and a `Uint8ClampedArray` in RGBA order `.data`; there are also `ctx.createImageData(w, h)` and `ctx.getImageData(x, y, w, h)` methods. The `ImageData` constructor is more experimental but available in Web Workers.

Computing normals from a Z-buffer

The raycasting process from any given pixel produces, at least, a z

-coordinate or distance at which the first intersection was found. (Maybe it also tells you things like how close the ray came to other objects, or how many iterations it took to converge, or which object it hit, and what the x and y coordinates were too, but at least it has a z-coordinate.) So this gives us a map from screen coordinates (u, v) -> z, which is pretty much exactly a Z-buffer.

Suppose we have nothing more than such a Z-buffer and we want to know the surface normals. A first crude approximation comes from the first backward differences: Z[6, 2] - Z[5, 2] gives us some kind of approximation of dz/du at (5.5, 2), and Z[5, 3] - Z[5, 2] gives us some kind of approximation of dz/dv at (5, 2.5).

These two sets of first differences have three problems:

• They can't distinguish discontinuities (from running off the edge of an object, for example) from smooth differences. If we're rendering smooth objects, this is an important difference.
• They're expressed in screen space. What we have is (roughly) dz/du and we want dz/dx. $x = uz$, so $dx = u\,dz$, so we need to divide by z. (Analogously for y.)
• They're offset by half a pixel from the pixels we want.

I think we can remedy problems #1 and #3 by, from the two adjacent gradient pixels at (5, 1.5) and (5, 2.5), choosing the one with the *smallest absolute value*. So, for example, if Z[5, 1] = 14, Z[5, 2] = 16, and Z[5, 3] = 15, we have deltas of +2 and -1 on each side of (5, 2), so we pick -1. If Z[5, 4] = 13, that gives us a delta of -2, so from -1 and -2, we pick -1 for the gradient y-element at (5, 3).

This will jitter some of the normals by half a pixel one way and others by half a pixel the other way, and it eliminates discontinuities unless there are discontinuities on both sides of the pixel, in which case it will pick the smaller discontinuity.

(This jitter is sort of like morphological erosion of these depth-gradient values, but by half a pixel rather than an integer number of pixels, and using the absolute value rather than the signed value. Maybe there's some kind of morphological dilation of depth-gradient values that could compensate, but I don't know how to dilate by fractional pixels.)

So this gives us an estimate of the gradient of the z-coordinate with respect to x and y. But what we want is the surface normal. Let's say $dz/dx = d$ and $dz/dy = e$. Now vectors tangent to the surface include $(1, 0, d)$ and $(0, 1, e)$; their cross-product is a normal, which if I'm not confused is $(-d, -e, 1)$, but that one points the wrong way, so we want $(d, e, -1)$, which points toward the camera. So you can normalize that $(\div \mathrm{sqrt}(d^2 + e^2 + 1))$ and you have some kind of reasonable approximation of the surface normal.

Per pixel, this approach requires two subtractions, two binary-minimum-by-absolute-value operations, a reciprocal (of z), two multiplications by it, two squarings, two additions, a reciprocal square root, its negation, and two multiplications by it, 15 operations in all, two of which (the reciprocal and reciprocal square root) are slow. This is probably faster than automatic differentiation of the SDF, and it gains the benefit of antialiasing that the centered-differences approximation has.

A simpler and cheaper function that might still provide a useful illusion of three-dimensionality is to take the gradient of z with

respect to (u, v) thus estimated and use it directly for, say, red and cyan color channels, as if the object were illuminated from the right with a red light and from above with a cyan one. Unfortunately, it's signed and may have a large dynamic range. The dynamic-range problem can maybe be handled by running it through some kind of sigmoid, like tanh or erf or something, although those only expand the dynamic range by like a factor of 2 or 3 or something; signedness can then be handled by truncating it with ReLU, i.e., $s \vee 0$. To get a third light coming from the camera, a simple approach would be to take the un-truncated sigmoid values and subtract the average of their absolute values from 1, although of course the actually correct thing would be to take the square root of 1 minus the sum of their squares.

Super cheap non-photo-realistic rendering from a Z-buffer

Suppose that instead we want to outline objects at discontinuities of depth and discontinuities of slope. The above gradient estimate attempts to smooth over single-pixel discontinuities by eroding them away, but if we skip that step, we can just use the total absolute gradient value (the L_1 norm; its Euclidean norm would be more correct) to select where to draw lines. You could use quickselect to find, say, the 96th and 97th percentiles of absolute gradient values, then use a piecewise-linear "smoothstep" to highlight the pixels with those values.

But that only gives us discontinuities *of depth*. If we want discontinuities *of slope*, we need an additional level of derivation, to calculate the second-order derivatives of the z-value with respect to (u, v), which are $\partial^2 z/\partial u^2$, $\partial^2 z/\partial u \partial v$, and $\partial^2 z/\partial v^2$, each of which is a single subtraction if we're satisfied with backward differences, as we should be. These values will be very large for two pixels at depth discontinuities, and somewhat large for one pixel at slope discontinuities; this may be a useful way to approximate the "thick inked lines outline figures, thin inked lines show detail within figures" rule from cartoon drawing.

In this case, although we have three values, we probably shouldn't map them to R, G, and B; it makes some visual sense to have an object diffusely illuminated from different directions with red and cyan lights, but it would make no visual sense to draw vertical lines in red, horizontal lines in blue, and saddle points and edge intersections in green. Instead we should take some kind of norm of this four-element Hessian matrix (which has only three distinct elements because it's symmetric).

Some kind of contouring may be a useful way to help make curved surfaces legible in diagrams; the simplest might be a checkerboard pattern in x, y, and z, formed by frobbing the pixel's brightness a bit with the XOR of some bit from each of the three coordinates. Which bit is chosen determines the scale of the checkerboard pattern along that dimension; if the bits are too significant, the whole image will be within a single cube, while if they're too insignificant, they're much smaller than a pixel, so it amounts to aliasing noise. So ideally you want them to be on the order of 32 pixels in size at the relevant distance.

A drawback of that kind of contouring is that one of the spatial

depth cues traditionally used in the visual arts is the amount of detail: foreground objects are drawn in more detail, as if the background objects were out of focus. (Also, they're often reduced in contrast and shifted toward the background color, which makes their details harder to see.) But the suggested checkerboard does just the opposite, putting the most detail in background objects. The standard XOR texture, the three-dimensional version of the Hadamard–Walsh matrix (the parity of the bitwise AND of the three coordinates), and similar fractal volumetric textures seen in a million 1024-byte demos and in My Very First Raytracer are one way to escape from that: they provide detail at a wide range of scales, or even all scales. But I'm not sure how to adjust the level of detail of such a volumetric texture to be coarser at further distances.

Another way to reduce the detail of background objects, related to the sphere-tracing algorithm rather than shading, is to roughen the sphere-tracing termination threshold with distance. If you do this merely proportional to distance, you just get cone-tracing antialiasing, but if you include a faster-growing term such as a quadratic term, you could maybe get background objects to lose surface detail faster than perspective shrinks that surface detail.

A third possible way to add more detail to foreground objects is to perturb the scene SDFs slightly with some kind of noise field to give the surface some kind of texture, but make the noise function drop off at larger z-coordinates, or possibly even middle-clip it ($a - (-b \wedge a \vee b)$, where a is the original noise function and b is the middle-clipping threshold) with a threshold that grows with distance.

In *Happy Jumping*, iq/rgba demonstrated a technique for perturbing SDFs slightly with a roughly isotropic volumetric texture made from a wavefunction with a feature size that was small relative to the objects but large relative to nearby pixels; the SDF perturbation was tightly clipped, so only its zero-crossings produced a relief on the surface. Thus, it formed smoothly curved random blobby contours on the surface whose foreshortening helped greatly to visually indicate the orientation of the surface. If you were doing such a thing I think you could reduce the level of detail with distance by reducing the amplitude of the higher-frequency components of the wavefunction.

Topics

Evaluating DSP operations in minimal buffer space by pipelining

Kragen Javier Sitaker, 2018-12-18 (updated 2018-12-19) (20 minutes)

I was thinking about how to implement a cascade of short-memory filters in C, how to manage all of their circular buffers, and it occurred to me that I can actually use the same circular buffer for all of them. Furthermore, especially if I'm willing to accept a few samples of extra latency, this allows me to move samples from one stage of the pipeline to the next with zero cost.

Suppose, for example, that I'm cascading the three FIR filters y(n) = x(n) - x(n-1), y(n) = x(n) - x(n-5), and y(n) = x(n) + x(n-15). Let's suppose our circular buffer b, indexed by index bi, has a size that is a power of 2 with bitmask m, so the current input sample is at `b[bi & m]`. It's convenient to invert the polarity of our first filter: y(n) = x(n-1) - x(n). We can implement this as follows:

```
b[(bi - 1) & m] -= b[bi & m];
```

This converts the sample at bi-1 from being x(n-1) to being y(n).

Now our second-stage filter, y(n) = x(n) - x(n-5), is being fed an inverted signal. So if it, too, is inverted, to y(n) = x(n-5) - x(n), we can implement it as follows:

```
b[(bi - 6) & m] -= b[(bi - 1) & m];
```

This converts the sample at bi-6 from being x(n-5) to being y(n). Note that this is the value we computed in the previous step, so it doesn't need to be loaded from RAM. The inversion of sign in this step cancels the inversion of sign in the previous step.

Then, our final stage, y(n) = x(n) + x(n-15), can be implemented similarly:

```
b[(bi - 21) & m] += b[(bi - 6) & m];
```

This converts the sample at bi-21 from being x(n-15) to being y(n).

Finally, you need to delay the output from the pipeline for 21 samples. Here I'm also dividing it by 4 to compensate for the amplification by the above filters.

```
if (bi >= 21) buf[bi - 21] = b[(bi - 21) & m] >> 2;
```

Here's a loop doing the above pipeline, although I haven't verified that it performs the desired DSP function:

```
for (bi = 0; bi != n; bi++) {
  b[bi & m] = buf[bi];
  b[(bi - 1) & m] -= b[bi & m];
  b[(bi - 6) & m] -= b[(bi - 1) & m];
  b[(bi - 21) & m] += b[(bi - 6) & m];
  if (bi >= 21) buf[bi - 21] = b[(bi - 21) & m] >> 2;
}
```

amd64 assembly

Here's what it looks like compiled with GCC, 92 bytes:

```
  400653:   74 5a                 je     4006af <filter+0x89>
  400655:   4d 89 c1              mov    %r8,%r9
  400658:   ba 00 00 00 00        mov    $0x0,%edx
        ; LOOP START
  40065d:   41 0f be 01           movsbl (%r9),%eax            ; load buf[bi]
  400661:   0f b6 ca              movzbl %dl,%ecx              ; bi & m

  400664:   89 04 8c              mov    %eax,(%rsp,%rcx,4)    ; b[bi & m] = .o
o..
  400667:   8d 4a ff              lea    -0x1(%rdx),%ecx       ; bi - 1
  40066a:   0f b6 c9              movzbl %cl,%ecx              ; & m

  40066d:   8b 1c 8c              mov    (%rsp,%rcx,4),%ebx    ; load b[(bi-1)o
o & m]
  400670:   29 c3                 sub    %eax,%ebx             ; - b[bi & m]
  400672:   89 d8                 mov    %ebx,%eax

  400674:   89 1c 8c              mov    %ebx,(%rsp,%rcx,4)    ; store b[(bi-1o
o) & m]
  400677:   8d 4a fa              lea    -0x6(%rdx),%ecx       ; bi - 6
  40067a:   0f b6 c9              movzbl %cl,%ecx              ; & m

  40067d:   8b 1c 8c              mov    (%rsp,%rcx,4),%ebx    ; load b[(bi-6)o
o & m]

  400680:   29 c3                 sub    %eax,%ebx             ; - b[(bi-1) & o
om]
  400682:   89 d8                 mov    %ebx,%eax

  400684:   89 1c 8c              mov    %ebx,(%rsp,%rcx,4)    ; store b[(bi-6o
o) & m]
  400687:   8d 4a eb              lea    -0x15(%rdx),%ecx      ; bi - 21

  40068a:   44 0f b6 d1           movzbl %cl,%r10d             ; & m (saving e
ontire)

  40068e:   42 03 04 94           add    (%rsp,%r10,4),%eax    ; b[(bi-21) & mo
o] + ...
  400692:   42 89 04 94           mov    %eax,(%rsp,%r10,4)    ; store it
  400696:   83 fa 14              cmp    $0x14,%edx            ; bi >= 21?
  400699:   76 09                 jbe    4006a4 <filter+0x7e>
  40069b:   89 c9                 mov    %ecx,%ecx             ; 2-insn NOP
  40069d:   c1 f8 02              sar    $0x2,%eax             ; b[...] >> 2
  4006a0:   41 88 04 08           mov    %al,(%r8,%rcx,1)      ; buf[
  4006a4:   83 c2 01              add    $0x1,%edx             ; bi++
  4006a7:   49 83 c1 01           add    $0x1,%r9              ; ++&buf[bi]
  4006ab:   39 fa                 cmp    %edi,%edx             ; bi != n?
  4006ad:   75 ae                 jne    40065d <filter+0x37>
```

It seems like %edx is being used as our loop counter `bi`, %r9 is being

used as the input counter `&buf[bi]`, and %edi is being used for n, the limit. My index mask `m` is 255, so GCC is implementing the masking with `movzbl` instructions. Each stage of the pipeline requires 6 instructions, for whatever that's worth in 2018, and there's an additional 4 instructions or so of loop overhead, for a total of 28 instructions inside the loop.

ARM assembly

The Cortex-M4 equivalent is as follows, 80 bytes and 29 instructions, 25 inside the loop:

```
  22:   b334        cbz r4, 72 <filter+0x72>
  24:   f105 3eff   add.w   lr, r5, #4294967295 ; 0xffffffff
  28:   2300        movs    r3, #0
  2a:   aa01        add r2, sp, #4
    ; LOOP START
  2c:   f81e 1f01   ldrb.w  r1, [lr, #1]!        ; buf[bi]
  30:   b2d8        uxtb    r0, r3               ; bi & m?
  32:   f842 1020   str.w   r1, [r2, r0, lsl #2] ; b[bi & m] = ...
  36:   1e58        subs    r0, r3, #1           ; bi - 1
  38:   b2c0        uxtb    r0, r0               ; & m?
  3a:   f852 6020   ldr.w   r6, [r2, r0, lsl #2] ; load b[(bi-1) & m]
  3e:   1a76        subs    r6, r6, r1           ; subtract r1 from it
  40:   f842 6020   str.w   r6, [r2, r0, lsl #2] ; store b[(bi-1) & m]
  44:   1f98        subs    r0, r3, #6           ; bi - 6
  46:   b2c0        uxtb    r0, r0               ; & m?
  48:   f852 1020   ldr.w   r1, [r2, r0, lsl #2] ; load b[bi-6 & m]
  4c:   1b8e        subs    r6, r1, r6           ; subtract r6 from it
  4e:   f842 6020   str.w   r6, [r2, r0, lsl #2] ; store b[bi-6 & m]
  52:   f1a3 0015   sub.w   r0, r3, #21          ; bi - 21
  56:   b2c0        uxtb    r0, r0               ; & m?
  58:   f852 1020   ldr.w   r1, [r2, r0, lsl #2] ; load b[bi-21 & m]
  5c:   4431        add r1, r6               ; add r6 to it
  5e:   f842 1020   str.w   r1, [r2, r0, lsl #2] ; save it
  62:   2b14        cmp r3, #20              ; bi > 20?
  64:   bf84        itt hi                   ; conditional, unsigned greater
  66:   1089        asrhi   r1, r1, #2           ; r1 >>= 2
  68:   f80e 1c15   strbhi.w    r1, [lr, #-21]; store buf[bi-21]?
  6c:   3301        adds    r3, #1
  6e:   42bb        cmp r3, r7
  70:   d1dc        bne.n   2c <filter+0x2c>
```

This uses 5 instructions for each stage of the pipeline, against AMD64's 6, and unlike implementations of AMD64, I think that number actually does mean 5 clock cycles under normal circumstances.

Actually, on further thought, I think my concern about the 21 samples of latency was unfounded. There's a data path straight through the loop from buf[bi] to what ought to be buf[bi] but is instead buf[bi-21]. There's *additionally* a delayed path through the FIFOs, which is as it should be.

(I fixed that bug and removed the >> 2 and verified that the impulse response was as it should be.)

Recursive filters

Recursive filters can also be implemented with this structure. Suppose we have a cascade of two humble integrators, y(n) = x(n) + y(n-1), as we might for the beginning of a Hogenauer filter. Of course, we could implement this as follows:

```
y1 += x;
y2 += y1;
```

But if we stick to this staged-FIFO structure so that we can implement arbitrary lags, it looks like this:

```
void filter2(char *buf, int n)
{
  enum { bufsiz = 256, m = bufsiz-1 };
  int b[bufsiz] = {0};

  for (unsigned bi = 0; bi != n; bi++) {
    b[bi & m] = buf[bi];
    b[bi & m] += b[(bi - 1) & m];          // ∫1
    b[(bi - 1) & m] += b[(bi - 2) & m];    // ∫2
    b[(bi - 7) & m] -= b[(bi - 1) & m];
    b[(bi - 12) & m] -= b[(bi - 7) & m];
    buf[bi] = b[(bi - 12) & m] >> 3;       // compensate for 25× amplfier
  }
}
```

The ∫1 line computes the new y(n) value from the new x(n) value and the previous y(n) value; the ∫2 line does the same thing, but for the second integrator. Absent any further transmogrification, it leaves a trail of second-order integrator values behind in the buffer. Then, immediately below, we have two feedforward comb filter lines, which tame the wild integrators and convert them into a mild-mannered triangular-kernel FIR filter, completing the structure of a second-order (CIC) Hogenauer low-pass filter, although in this case without the usual decimation step.

In ∫1, because we don't need to preserve x(n), we can transform it into y(n), saving a memory load. But in the ∫2 stage, we do need to preserve x(n) (the previous stage's y(n)), so we just overwrite y(n-1) instead.

Again, in the comb filters, we're inverting the sign here for convenience — instead of computing y(n) = x(n) - x(n-5), we're computing x(n-5) - x(n), which requires no extra work because we have an even number of such inverting comb filters in the pipeline.

This filter function ends up as follows on amd64 with -Os:

```
  9b:   48 81 ec 18 04 00 00    sub    $0x418,%rsp
  a2:   48 89 fa                mov    %rdi,%rdx

  a5:   b9 00 01 00 00          mov    $0x100,%ecx             ; buffer size to
o zero
  aa:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  b1:   00 00
  b3:   48 89 84 24 08 04 00    mov    %rax,0x408(%rsp)
  ba:   00
```

```
  bb:   31 c0                   xor    %eax,%eax
  bd:   48 8d 7c 24 08          lea    0x8(%rsp),%rdi

  c2:   f3 ab                   rep stos %eax,%es:(%rdi)        ; zero the buffe
or
  c4:   48 89 d7                mov    %rdx,%rdi
    ;; LOOP START
  c7:   39 f0                   cmp    %esi,%eax                ; bi (%eax) != n
  c9:   74 64                   je     12f <filter2+0x94>
  cb:   0f be 0f                movsbl (%rdi),%ecx              ; buf[bi]
  ce:   8d 50 ff                lea    -0x1(%rax),%edx          ; bi - 1
  d1:   44 0f b6 c0             movzbl %al,%r8d                 ; bi & m
  d5:   48 ff c7                inc    %rdi                     ; ++&buf[bi]
  d8:   0f b6 d2                movzbl %dl,%edx                 ; (bi - 1) & m

  db:   42 89 4c 84 08          mov    %ecx,0x8(%rsp,%r8,4)     ; b[(bi-1) & m]
= …
  e0:   03 4c 94 08             add    0x8(%rsp,%rdx,4),%ecx    ; … + b[bi & m]
  e4:   42 89 4c 84 08          mov    %ecx,0x8(%rsp,%r8,4)     ; b[bi & m] = …
  e9:   8d 48 fe                lea    -0x2(%rax),%ecx          ; bi - 2

  ec:   44 8b 44 94 08          mov    0x8(%rsp,%rdx,4),%r8d  ; XXX redundant loa
d
  f1:   0f b6 c9                movzbl %cl,%ecx
  f4:   44 03 44 8c 08          add    0x8(%rsp,%rcx,4),%r8d  ; XXX WTF
  f9:   8d 48 f9                lea    -0x7(%rax),%ecx          ; bi - 7
  fc:   0f b6 c9                movzbl %cl,%ecx                 ; & m
  ff:   44 89 44 94 08          mov    %r8d,0x8(%rsp,%rdx,4)
 104:   8b 54 8c 08             mov    0x8(%rsp,%rcx,4),%edx
 108:   44 29 c2                sub    %r8d,%edx
 10b:   89 54 8c 08             mov    %edx,0x8(%rsp,%rcx,4)
 10f:   8d 48 f4                lea    -0xc(%rax),%ecx
 112:   ff c0                   inc    %eax
 114:   0f b6 c9                movzbl %cl,%ecx
 117:   44 8b 4c 8c 08          mov    0x8(%rsp,%rcx,4),%r9d
 11c:   41 29 d1                sub    %edx,%r9d
 11f:   44 89 ca                mov    %r9d,%edx
 122:   44 89 4c 8c 08          mov    %r9d,0x8(%rsp,%rcx,4)
 127:   c1 fa 03                sar    $0x3,%edx
 12a:   88 57 ff                mov    %dl,-0x1(%rdi)
 12d:   eb 98                   jmp    c7 <filter2+0x2c>
 12f:   48 8b 84 24 08 04 00    mov    0x408(%rsp),%rax
 136:   00
 137:   64 48 33 04 25 28 00    xor    %fs:0x28,%rax        ; stack canary?
 13e:   00 00
 140:   74 05                   je     147 <filter2+0xac>
 142:   e8 00 00 00 00          callq  147 <filter2+0xac>
 147:   48 81 c4 18 04 00 00    add    $0x418,%rsp
 14e:   c3                      retq
```

Looks like it has 6 stores, 5 fetches, and 30 instructions inside the loop to realize this four-stage pipeline, but again only 5 or 6 instructions per stage of the pipeline. I was hoping to reach some kind of conclusion here about whether recursive filtering was going to be slightly costlier, but I really have no idea.

It looks to me like GCC has given up on register allocation in part of this code and is just emitting redundant loads, which is weird.

However, this is definitely less costly:

```
y1 += x;
y2 += y1;
```

And if you're doing a Hogenauer filter, you should probably not only do that, but you should also decimate the stuff downstream from the integrators.

But you can use this approach for other kinds of recursive filtering, too.

On one core of my laptop, this code filters 100 mebibytes (of malloced, mostly uninitialized memory) in 785 ms. That's about 134 million samples per second, or about 12 clock cycles per sample, or about 3 clock cycles per sample per pipeline stage. Presumably the number on a Cortex-M4 would be more like 6 clock cycles per sample per pipeline stage, due to in-order execution.

This suggests that processing a signal at 44.1 ksps would require about 132 kHz of amd64 clock per stage or 266 kHz of Cortex-M4 clock per stage. This seems like it might come in at the sub-MIPS level I was hoping for in the Bleep modem.

Multirate processing

So here's a Hogenauer filter with a twist — it's tuned to detect one particular frequency (19110 Hz) and null another one (17640 Hz). At least, that was the intention. I haven't tested this C code, just code in Python that purports to be equivalent, except that it uses floating-point (!!).

```
unsigned filter3(signed char *buf, int n)
{
  enum { bufsiz = 256, m = bufsiz-1 };
  int b[bufsiz] = {0};
  uint64_t d1=0, d2=0, d3=0, d4=0, i1=0, i2=0, i3=0, tmp, tmp2, c2=0, c3=0;
  int bi5=0, bo=0, odd=0;

  for (unsigned bi = 0; bi != n; bi++, bi5++) {
    tmp = buf[bi];
    tmp2 = tmp - d1;            /* Differentiator 1 */
    d1 = tmp;
    tmp = tmp2 - d2;            /* Differentiator 2 */
    d2 = tmp2;
    tmp2 = tmp - d3;            /* Differentiator 3 */
    d3 = tmp;
    tmp = tmp2 - d4;            /* Differentiator 4 */
    d4 = tmp2;
    i1 += odd ? -tmp : tmp;     /* Integrator 1 */
    odd = ~odd;
    i2 += i1;                   /* Integrator 2 */
    i3 += i2;                   /* Integrator 3 */

    if (bi5 == 5) {
      bi5 = 0;
      /* First comb filter: y(n) = x(n-2) - x(n); note inversion */
```

```
      b[bi & m] = i3;
      b[(bi - 2) & m] -= i3;
      /* Second comb filter: y(n) = x(n-1) - x(n); canceling inversion */
      tmp = c2 - b[(bi - 2) & m];
      c2 = b[(bi - 2) & m];

      /* Third comb filter: y(n) = x(n) - x(n-1), this time not inverting; but no
ow into the output buffer */
      buf[bo++] = tmp - c3;
      c3 = tmp;
    }
  }
  return bo;
}
```

This rather alarming C comes out to this perhaps even more alarming assembly:

```
000000000000014f <filter3>:
 14f:   41 57                   push   %r15
 151:   41 56                   push   %r14
 153:   b9 00 01 00 00          mov    $0x100,%ecx
 158:   41 55                   push   %r13
 15a:   41 54                   push   %r12
 15c:   49 89 fc                mov    %rdi,%r12
 15f:   55                      push   %rbp
 160:   53                      push   %rbx
 161:   31 d2                   xor    %edx,%edx
 163:   45 31 db                xor    %r11d,%r11d
 166:   45 31 c0                xor    %r8d,%r8d
 169:   45 31 ff                xor    %r15d,%r15d
 16c:   48 81 ec 38 04 00 00    sub    $0x438,%rsp
 173:   45 31 d2                xor    %r10d,%r10d
 176:   45 31 c9                xor    %r9d,%r9d
 179:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
 180:   00 00
 182:   48 89 84 24 28 04 00    mov    %rax,0x428(%rsp)
 189:   00
 18a:   31 c0                   xor    %eax,%eax
 18c:   48 8d 7c 24 28          lea    0x28(%rsp),%rdi
 191:   89 74 24 1c             mov    %esi,0x1c(%rsp)
 195:   45 31 f6                xor    %r14d,%r14d
 198:   45 31 ed                xor    %r13d,%r13d
 19b:   31 ed                   xor    %ebp,%ebp
 19d:   31 db                   xor    %ebx,%ebx
 19f:   f3 ab                   rep stos %eax,%es:(%rdi)
 1a1:   31 ff                   xor    %edi,%edi
    ; LOOP START
 1a3:   3b 54 24 1c             cmp    0x1c(%rsp),%edx
 1a7:   89 54 24 18             mov    %edx,0x18(%rsp)
 1ab:   0f 84 a0 00 00 00       je     251 <filter3+0x102>

 1b1:   49 0f be 34 14          movsbq (%r12,%rdx,1),%rsi     ; sign-extending by
te to quad?
 1b6:   48 89 34 24             mov    %rsi,(%rsp)
```

```
1ba:   48 29 de                sub    %rbx,%rsi              ; d1
1bd:   48 89 74 24 08          mov    %rsi,0x8(%rsp)
1c2:   48 29 ee                sub    %rbp,%rsi              ; d2
1c5:   48 89 74 24 10          mov    %rsi,0x10(%rsp)
1ca:   4c 29 ee                sub    %r13,%rsi              ; d3
1cd:   48 89 f5                mov    %rsi,%rbp
1d0:   49 89 f5                mov    %rsi,%r13
1d3:   4c 29 f5                sub    %r14,%rbp              ; d4
1d6:   48 89 eb                mov    %rbp,%rbx
1d9:   48 f7 db                neg    %rbx
1dc:   45 85 db                test   %r11d,%r11d            ; odd
1df:   41 f7 d3                not    %r11d
1e2:   48 0f 44 dd             cmove  %rbp,%rbx
1e6:   49 01 d9                add    %rbx,%r9               ; i1
1e9:   4d 01 ca                add    %r9,%r10               ; i2
1ec:   4c 01 d7                add    %r10,%rdi              ; i3
1ef:   41 83 f8 05             cmp    $0x5,%r8d
1f3:   75 40                   jne    235 <filter3+0xe6>
   ; DECIMATED CONDITIONAL START
1f5:   8b 5c 24 18             mov    0x18(%rsp),%ebx
1f9:   44 0f b6 44 24 18       movzbl 0x18(%rsp),%r8d
1ff:   83 eb 02                sub    $0x2,%ebx              ; bi - 2
202:   0f b6 db                movzbl %bl,%ebx               ; & m
205:   42 89 7c 84 28          mov    %edi,0x28(%rsp,%r8,4)  ; b[bi & m] = i3
20a:   44 8b 44 9c 28          mov    0x28(%rsp,%rbx,4),%r8d ; b[bi-2 & m]
20f:   41 29 f8                sub    %edi,%r8d              ; x(n-2) - x(n)
212:   44 89 44 9c 28          mov    %r8d,0x28(%rsp,%rbx,4) ; ?
217:   4d 63 c0                movslq %r8d,%r8
21a:   48 63 d8                movslq %eax,%rbx
21d:   4c 29 c1                sub    %r8,%rcx
220:   ff c0                   inc    %eax
222:   40 88 cd                mov    %cl,%bpl
225:   44 29 fd                sub    %r15d,%ebp
228:   49 89 cf                mov    %rcx,%r15
22b:   4c 89 c1                mov    %r8,%rcx
22e:   41 88 2c 1c             mov    %bpl,(%r12,%rbx,1)
232:   45 31 c0                xor    %r8d,%r8d
   ; DECIMATED CONDITIONAL END
235:   4d 89 ee                mov    %r13,%r14
238:   41 ff c0                inc    %r8d
23b:   48 ff c2                inc    %rdx
23e:   4c 8b 6c 24 10          mov    0x10(%rsp),%r13
243:   48 8b 6c 24 08          mov    0x8(%rsp),%rbp
248:   48 8b 1c 24             mov    (%rsp),%rbx
24c:   e9 52 ff ff ff          jmpq   1a3 <filter3+0x54>
   ; LOOP END
251:   48 8b 94 24 28 04 00    mov    0x428(%rsp),%rdx
258:   00
259:   64 48 33 14 25 28 00    xor    %fs:0x28,%rdx
260:   00 00
262:   74 05                   je     269 <filter3+0x11a>
264:   e8 00 00 00 00          callq  269 <filter3+0x11a>
269:   48 81 c4 38 04 00 00    add    $0x438,%rsp
270:   5b                      pop    %rbx
271:   5d                      pop    %rbp
```

```
272:    41 5c                   pop    %r12
274:    41 5d                   pop    %r13
276:    41 5e                   pop    %r14
278:    41 5f                   pop    %r15
27a:    c3                      retq
```

So, this has an 8-stage per-sample pipeline (which will be shared with the other ultrasound signals I want to detect), followed by a decimator and a three-stage per-output-sample pipeline. The loop overhead is 7 instructions per sample at the end and 3 instructions per sample at the beginning; the 9-stage per-sample code is 20 instructions; then the decimated code is 18 instructions, averaging 3.6 instructions per loop. All in all this should run in 33.6 instructions per sample, which would be about 17 amd64 clock cycles per sample if the same ratio held as for the previous code whose performance I actually measured. Also that works out to 12 pipeline stages, which means about 3 instructions or 1.5 clock cycles per stage, definitely an improvement over the earlier numbers.

It probably isn't necessary to use 64-bit math in here, and definitely not for all the variables it's being used for. But I haven't done the analysis yet to be sure I understand the Hogenauer overflows.

This suggests that doing the full demodulation this way will require a bit over a Cortex-M4 MIPS.

The differentiators on the front end might not be necessary; they're there to filter out the (commonly much more powerful) low-frequency noise.

Topics

Notes on scraping the Codex Arundel to preserve it

Kragen Javier Sitaker, 2017-08-22 (1 minute)

The Codex Arundel is a codex of manuscripts from Leonardo da Vinci, made of 283 sheets of paper, each with a recto and a verso side. The British Library has scanned it at 8579×6250 pixels, divided into 256-pixel-square tiles with one pixel of overlap, each about 12 kilobytes, fetchable using wget. This works out to 34×25 tiles per side of a page, so the total number of tiles is (* 34 25 283 2) = 481,100, and the total weight of the work should be about 5.8 gigabytes.

Initially each of the 566 images loads as an image of about 1800×800, about 2.7% of the total size. It would make some sense to fetch these low-resolution images first (totaling 160 megabytes) before making the possibly doomed effort to fetch the whole dataset.

The page for the thing is at https://www.bl.uk/manuscripts/FullDisplay.aspx?ref=Arundel_MS_00263.

Topics

- Archival (p. 3322) (34 notes)

Incremental MapReduce for Abelian-group reduction functions

Kragen Javier Sitaker, 2015-09-03 (4 minutes)

MapReduce is a batch-processing framework. Manuel Simoni wrote about incremental MapReduce in 2008 and is working on it still. He explains that in Damien Katz's proposed solution, "you have to store intermediate results and do recomputation when the inputs change," but you can avoid this if "map() stays the same, but reduce() is extended so that it takes a diff of the map outputs from before and after a document update." Simoni's design has an additional benefit over Katz's: Katz's will nearly always involve re-executing the entire reduce() stage. Simoni's design does, however, involve storing the final reduce() output.

This is a very interesting idea. Simoni comments that it has a drawback:

> [R]educe functions get more complex in this scheme, but the Google papers on MapReduce and Sawzall suggest that map functions are much more often user defined than reduce functions.

There's actually a whole algebraic landscape associated with this statement.

To summarize algebra in a few lines, a magma that's associative is a semigroup, a semigroup that's commutative and idempotent is a semilattice, a semigroup with identity is a monoid, a monoid with inverses is a group, and a commutative group is an Abelian group.

In many cases, you can construct your reduce-function by composing an Abelian group operation with some other operation. Undoing an Abelian group operation is relatively trivial: because the operation is associative (because semigroup) it doesn't matter what order the operations were done in, and because it's commutative, it doesn't even matter where in the sequence the removed element was. So you can just apply the inverse of the removed element to the previous reduce-function result.

As one broadly applicable example, if we extend bags over some set A, $A \to \mathbb{N}$, to allow negative multiplicities, $A \to \mathbb{Z}$, then any bag has its antibag which is its inverse under extended bag sum $\uplus$.

So if you store the result of the Abelian group operation, you can apply the diff to it, then redo the final operation.

For example, to maintain the set of unique words present in a set of documents, you can maintain in storage an extended bag that counts the number of times each word occurs, and then just discard the multiplicities to produce the final result.

As another example, if you want to know the largest one-second price jump from a large collection of stock-market data, you can divide your input data into shingles that overlap by a second, use a map function that outputs just the largest jump in each shingle, and then use a max-by-jump-size as your reduce function. Unfortunately, max, like semilattice operations generally, does not admit element inverses. A general incremental solution for this at the point of reduce would involve maintaining, at least, a max-heap of existing records, to which you could add and remove records.

Max-heaps, then, are the Abelian group needed here. But this has the same space cost (modulo overhead) of Katz's solution of storing previous map results and re-executing affected reductions.

If you cannot decompose your reduce-function into an Abelian group operation, the problem is more complex; but in fact reduce-functions in MapReduce are almost invariably intended to be commutative and associative, because the order in which the different map-results are presented to them is more or less arbitrary.

The max-heap example points up an inadequacy in the group-based, or even magma-based, analysis of incrementally updatable reduce-functions: it requires that the things coming from the map-functions be of the same type as the internal state of the reduce-function. In Haskell, `foldl` has type `(a → b → a) → a → [b] → a` — it takes an initial `a`, produces a final `a`, and in the middle it folds a list of `b`s into it using an `a → b → a` function. For example, `a` could be a max-heap of stock-market price moves, and `b` could be a single stock-market price move.

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- Systems architecture (p. 3691) (48 notes)
- Incremental computation (p. 3517) (24 notes)
- Algebra (p. 3309) (11 notes)
- Parallelism (p. 3616) (8 notes)
- Trading (p. 3754) (4 notes)
- Mapreduce
- Haskell

Replacing fractional-reserve banking with a bond market disintermediated with a blockchain

Kragen Javier Sitaker, 2019-07-03 (6 minutes)

I was reading Jonathan Stray's account of Jon Hicks's possibly fictional 1989 account of the how banking and modern money came to be, and it occurred to me that the bond market might be an alternative to the borrow-short lend-long model of fractional-reserve demand-deposit banking.

That is, if you have some money that you want to keep liquid but also earn interest on, one possibility is to deposit it in a bank as demand deposits. The bank lends (most of) it out at higher interest, keeping a small ("fractional") reserve on hand in case some depositors show up one day to close their accounts, and pays some (lower) interest to you and the other depositors for the use of your money. (Modern banks have been able to mostly eliminate depositor interest, as their oligopoly on payment systems, and in some case government regulations, obliges people to deposit their money in banks even if they don't earn interest.) The risk you take is that there might be a bank run — if too many depositors show up demanding their demand deposits at once, the first ones will get paid in full and the last ones will get their share of whatever is left over from the bankruptcy.

Given the existence of a liquid trading market, bonds, such as corporate bonds, might be a reasonable alternative. Instead of depositing your money in a bank account, you buy some bonds. Instead of withdrawing it, you sell the bonds on the market. Bond prices go up and down a bit — usually measured in basis points — and occasionally companies will go bankrupt (or countries will go into sovereign default) and again you're just a bankruptcy creditor, but you can diversify to minimize this risk. Lending to companies is overall mostly profitable, and the interest rates are higher than you get with a checking account.

This is why money-market accounts are popular, but there's no fundamental reason that retail investment in the corporate-bond market needs to be intermediated by banks or brokers; that's just an artifact of the limits of 20th-century information-processing technology. Intermediating retail access to corporate bonds in this way just adds risk — your bank or broker is far more likely to go bankrupt than the entire basket of companies you lend to, and additionally they're in a position to cheat you in a variety of ways, including bucket-shop tactics (that is, fractional-reserve banking, but in a way that is illegal in the US), front-running, and paying bonuses to their executives just before bankruptcy.

How would this differ in practice from a bank account? Blue-chip corporate and sovereign bonds typically fluctuate in value only by basis points, so the chances of a significant loss of capital in this way is fairly small, and you can diversify the risk across many different

debtors. The big difference is what happens when there's a "bank run": in the money market, those lenders who are most eager to liquidate their holdings are often the ones who take the losses, while in a demand-deposit bank run, the most eager lenders are the only ones who get paid. A temporary selloff in the bond market doesn't directly affect the companies whose bonds are being sold, unless they are floating a new issue of bonds at that moment, which they usually aren't. If the selloff is indeed temporary, rather than a result of the debtor's impending insolvency, those lenders who held during the selloff will preserve their capital (the precise opposite of a bank run), and those who risked buying the bargain-priced bonds are rewarded.

In the modern commercial paper market, though, clearing of trades is not instantaneous; the US corporate and municipal debt markets mostly clear via DTC, the Depository Trust Company, which acts as a counterparty to most transactions in corporate and municipal debt, permitting settlement to be delayed for two days or longer. As I understand it, the London Clearing House's EquityClear SA, despite its name, plays a similar role in Europe. Such clearing houses centralize debt-transaction counterparty risk in much the same way that banks and bank clearing houses centralize counterparty risk for cash transactions; without clearing houses, buyers of debt would be taking the risk that sellers would take the money and run without ever handing over the bonds they were ostensibly selling.

Clearing house insolvency is a likely outcome of the next world war, however, and represents the kind of systemic risk that it's very difficult to diversify away.

The obvious solution is to clear the trades with the Ethereum blockchain or something similar, so that the transfer of debt title from the seller to the buyer is a single atomic transaction with the transfer of funds from the buyer to the seller. This eliminates counterparty risk from the bond markets as long as the blockchain's integrity remains secure.

The actual coupon payments on the bonds, or title to collateral to secure those payments, *could* be incorporated into the same smart contract, but this is much less essential — the lenders are at least theoretically aware of who they're lending to and assuming the risk that that counterparty (the borrower, that is, the issuer of the bonds) will default. The whole purpose of the borrower issuing debt in the first place is so that they can invest the money thus raised, so you would expect that for most of the lifetime of the bond, they won't have an account with enough money in it to pay off the bond. So there's little point in making the coupon payments nominally automatic via a smart contract.

Topics

- Ethereum
- Banking

Installing Debian GNU/Linux on an ASUS E403S

Kragen Javier Sitaker, 2016-10-23 (10 minutes)

I have this new Asus laptop that says it's model E403S, and unfortunately it comes with Microsoft Windows 10 installed. These are my notes on trying to install Linux (specifically Debian GNU/Linux 8.6, a version of "jessie") on it.

So far I have gotten Debian to boot on it, despite UEFI, but I have not been successful installing it to the disk. Also, X is running at 800×600 (on a 1920×1080 screen), and the touchpad doesn't work.

By contrast, the current Linux Mint installs fine and everything works (encrypted install including swap, 1920×1080, touchpad, Wi-Fi, sound input and output, access to the 100GB SSD, webcam) except hibernation. I may return to Debian at some point.

Downloading the Debian live CD image

This step can take a while; for me it was a bit under an hour. Start it first unless you're on a very-high-speed internet connection. Some of the other steps can be done in parallel.

The best way to download Debian CD images is using zsync, and the best image to download is typically one of the "+nonfree" ones which have the drivers you need in order to bring up Wi-Fi successfully. These are "unofficial" because Debian does not condone the use of non-free software.

I did this with this command:

```
time zsync http://cdimage.debian.org/cdimage/unofficial/non-free/cd-including-firo
omware/8.6.0-live+nonfree/amd64/iso-hybrid/debian-live-8.6.0-amd64-standard+nonfreo
oe.iso.zsync
```

The resulting ISO file was 494927872 bytes and had the following cryptographic hashes:

```
sha256sum 496bef6cba35d5348d5cb2662f649f1949b0d9ee4a4b4ea7addf46b369a70507 debiano
o-live-8.6.0-amd64-standard+nonfree.iso

sha1sum   60e52866683d7205adabc6c41d2f0ce04d159b77 debian-live-8.6.0-amd64-standao
ord+nonfree.iso

md5sum    d79befa133725bb07251feb81a980ccc debian-live-8.6.0-amd64-standard+nonfro
oee.iso
```

The "standard" live "CD" doesn't bring up a GUI environment, but it's half the size of the alternatives that do (472 megabytes vs. 1.1+ gigabytes).

zsync is the best option because it detects and can recover from data transmission errors, and can restart after an interruption after taking only about 20 to 200 seconds to resynchronize. If I understand

correctly, it checks the truncated MD4 of each block of the file and the SHA-1 of the whole file. It's like a less demanding version of rsync.

Creating a UEFI-bootable USB stick for Debian

The Debian Live images are in "hybrid ISO" format, which means they are bootable under ancient BIOS from either USB or CD/DVD. Unfortunately, although the official instructions don't mention this, they are not bootable via UEFI, so I followed the following steps to get Debian to boot from USB under UEFI, as explained by Desmond86 in a Debian User Forums thread.

I formatted the USB stick as FAT32 (so UEFI would understand it) using `sudo cfdisk /dev/sdc`, creating a type-0c partition filling the entire disk, and then running `sudo mkfs -t vfat /dev/sdc1`. Then I mounted it with `sudo mount /dev/sdc1 /mnt` and copied `Shell.efi` from https://svn.code.sf.net/p/edk2/code/trunk/edk2/ShellBinPkg/UefiShell/X64/Shell.efi into `/mnt`.

In SVN revision 22855, `Shell.efi` is 909280 bytes and has the following checksums:

```
sha256sum 889a1f28051955fc33a9512901b2d0f5a5d500750e09fb7caf21defb1fd3b657 Shell.efi
sha1sum   6621d657f470c3902ab1bc2423e45e74d5c286cc Shell.efi
md5sum    8f2922f6d148c5a5776cf16c8952a1f4 Shell.efi
```

Then, I mounted the Debian ISO with `sudo mkdir /iso; sudo mount -o loop debian-live-8.6.0-amd64-standard+nonfree.iso /iso` and copied its contents onto the USB stick.

This was tricky; here's the command that finally worked:

```
time sudo cp -iLr /iso/{a*,.d*,di*,[f-z]*} /mnt/
```

The weird filename specification is to avoid copying the circular symlink called `debian` (which, fortunately, isn't strictly necessary) or the `..` directory entry. I used `-r` rather than `-a` to avoid errors about not being able to set ownerships on FAT32. The `-L` flag is to avoid errors about not being able to create symbolic links on FAT32. The missing symlinks might prevent some firmware files from being found (although it wasn't necessary to get my Wi-Fi working the first time around). The `-i` flag is to stop and tell me what's going on if there are clashes with files already there (like if I forgot to delete this stuff previously). This copying takes about five minutes (with one of the larger images, the Mate one) on my USB stick, most of which is copying `/iso/live/filesystem.squashfs`, although that depends on the speed of the USB stick in question. Mine is only writing at 5 megabytes per second, as shown by `iostat 10` in another window.

The following command was useful for watching the progress of the copying and figuring out what went wrong; it will go wrong itself if you have another `cp` command running somewhere.

```
watch -d -n .1 'sudo lsof -p $(pidof cp) | tail'
```

Things I tried for copying that didn't quite work:

• `cp -a …`, which spews lots of error messages about failing to create symlinks (because FAT32 doesn't support them) and failing to set file ownership (because FAT32 doesn't support that either).
• `cp -iLr /iso /mnt`, which puts everything in `/mnt/iso`.
• `cp -iLr /iso/{.,}* /mnt`, but it ends up trying to copy my entire filesystem onto the USB stick, including `/usr` and whatnot.
• `cp -iLr /iso/{*,.[^.],.??*} /mnt/`, which almost works, but copies everything into both `/mnt` and `/mnt/debian`, doubling the space usage.

Then I created a text file called `/mnt/liveboot.nsh` containing just this line:

```
live\vmlinuz initrd=live\initrd.img append boot=live components
```

Disabling hiberboot in Microsoft Windows 10

This isn't actually necessary for what I'm doing, since I'm going to erase Windows completely. I thought I had to do this to get into the BIOS setup, but I was wrong. It is necessary, however, to avoid filesystem corruption if you want to dual-boot and have any filesystems available to both Windows and Linux.

In Windows 10, I got to this via Settings (Windows → Settings) in System → Power & Sleep → Additional Power Settings, which takes me to Control Panel → Hardware and Sound → Power Options, from which I choose "Choose what the power buttons do" in the left sidebar, followed by "Change settings that are currently unavailable", which makes a "Shutdown settings" section appear, containing a checkbox for "Turn on fast startup (recommended)", which I unchecked, followed by clicking the "Save changes" button. The rumors that a locked filing cabinet guarded by a leopard is involved are unfounded.

Then I shut Windows down from the Windows → Power menu.

Entering BIOS setup

During the ASUS splash screen, before the blue-background Windows booting screen starts, pressing Esc gives a "Please select boot device" menu, one of whose options is "Enter setup".

Booting from USB

This part is tricky and dangerous, because the BIOS

Once I had the UEFI-bootable USB stick and disabled secure boot (in BIOS setup, under "Security", "Secure Boot menu", set "Secure Boot Control" to "Disabled") I was able to add the EFI shell as a boot option as follows. In BIOS setup, in "Boot", with the USB stick plugged in, I selected "Add New Boot Option", whose resulting menu contains "Add boot option" (prompting for an arbitrary but mandatory name, such as "walnuts") and "Path for boot option", which allowed me to navigate to "PCI(14|0)\USB(0,09)\HD(Part1,Sig0D30060D)" and then select the "Shell.efi" file I'd put on the USB stick in the previous section. Then I selected "Create", used Esc to get back to the previous menu

and see that my new boot option was selected. Upon pressing F10 to reboot, I got the UEFI Interactive Shell, which has the USB stick mapped as FS1: and apparently the internal disk mapped as FS0:; I can `dir FS0:\EFI` and see Microsoft in there, while `dir FS1:` shows me the contents of the USB stick.

Typing `liveboot` and hitting "enter" runs the liveboot.nsh batch file mentioned in the previous section, booting Debian.

Updating the ASUS E403S BIOS

The BIOS version on this machine is 213, which ASUS's page (see below) tells me is from 2015-11-10.

The Debian kernel isn't seeing the disk, and AskUbuntu suggests that this may be fixed by upgrading the BIOS.

So I loaded version 301 of the BIOS, date 2016-10-21, from ASUS's page. This page says it's for model E403SA, but the tag on the laptop says the model is both "E403S" and "E403SA-US21". The file I got is 2847438 bytes and has the following cryptographic checksums:

```
sha256sum a9f49a64fae2d915a2b9b6f7c515bfd4473a50f1ba4db071fa6e554397045113  E403SAAS301zip.zip
md5sum    3204c1bb9527bba7777cd89a4a528dbb  E403SAAS301zip.zip
sha1sum   a8f6ffc513d14e47b2771e737c753ecc40c5a55b  E403SAAS301zip.zip
```

The option to update the BIOS is in the BIOS setup under "Advanced": "Start Easy Flash". This is supposed to be able to read the above zip file from USB media. Unfortunately it insists on being plugged in to AC power, and I left the adaptor elsewhere tonight. So for now I'm kind of stuck at this point, although I can run a live Debian system from USB.

Launching the Debian installer from the login prompt

For whatever reason, the Debian Live USB image I booted (`debian-live-8.6.0-amd64-standard+nonfree.iso`) gives me a login prompt instead of automatically launching a desktop or an installer. I logged in with the login name `user` and the password `live` (these are documented on Stack Overflow) and ran `sudo debian-installer-launcher`.

A few steps into the installer, it queried me for the wireless network configuration, bringing the network up successfully. From this point on, things basically worked, except that it couldn't find the disk to install on, as documented above. I had to tell it my network was "WPA/WPA2 PSK", as most are these days, rather than "WEP/Open Network", the poorly-chosen default.

At this point I'm stuck, as mentioned above, because I can't get Linux to see the disk I want it to install on.

I did manage to get Mate running (from a different Debian live image), but the touchpad mouse doesn't work and the display only supports a resolution of 800×600, which is 23% of its native resolution of 1920×1080 and also blurry and smushed. Alt-F1 brings up the main menu despite the touchpad not working.

Nested inheritance

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

From "Scalable Extensibility via Nested Inheritance", by Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers:

In our work on the Polyglot extensible compiler framework [27], we found that ordinary object-oriented inheritance and method dispatch do not adequately support extensibility. Because inheritance operates on one class at a time, some kinds of code reuse are difficult or impossible. For example, inheritance does not support extension of an existing class library by adding a given field or method to all subclasses of a given class. Inheritance is also inadequate for extending a set of classes whose objects interact according to some protocol, a pattern that occurs in many domains ranging from compilers to user interface toolkits. It can be difficult to use inheritance to reuse and extend interdependent classes.

Nested inheritance is a language mechanism designed to support scalable extensibility. Nested inheritance creates an interaction between containment and inheritance. When a container (a namespace such as a class or package) is inherited, all of its components — even nested containers — are inherited too. In addition, inheritance and subtyping relationships among these components are preserved in the derived container. By deriving one container from another, inheritance relationships may be concisely constructed among many contained classes.

I've thought for a while that one of the advantages of Bicicleta's language is that it supports this kind of extensibility. However, their approach causes a class that "overrides" a class from another container to inherit from that other class, unlike Bicicleta's approach, which seems to be more similar to the "virtual class" mechanism of BETA and "Genericity in Java with virtual types".

Topics

Predictions for future technological development (2008)

Kragen Javier Sitaker, 2008-04-19 (11 minutes)

Here I'm going to try to write down some of my predictions for future technological development, so that I can check them later. I'm assigning things I'm more sure about a higher "certainty level or 'CL'", which I'm thinking will be in a range of 0–9.

Inspired by http://kk.org/ct2/2008/04/digital-things-ive-been-wrong.php.

2008-04-19

Second Life (or equivalent): going to be the basis of a lot of socialization in 10 years, if they can make it decentralized (CL 2); primarily among groups of people who know each other in real life, but aren't physically co-located at the moment (CL 2).

Spore: going to be a huge seller (CL 3) and even bigger influence on newly designed games (CL 4) (and maybe other programs (CL 2)) over the next 5 years.

Automated fabrication (one-off production of goods without direct human intervention): starting around 2010 and by 2013, it's going to replace more than 5% of traditional manufacturing, measured by value (CL 4). Price-sensitive goods will continue to be made by traditional manufacturing unless and until automated fabrication gets cheaper than traditional manufacturing plus transportation (CL 5), which I don't think will happen in 5 years and probably not in 10 (CL 3).

Version control: in five years, the mainstream version control systems will include SVN (CL 5), Git (CL 5), and Mercurial (CL 4), but not Darcs (CL 3), Bazaar (CL 3), any other arch variant (CL 4), CVS (CL 2), or Perforce (CL 4). They already don't include RCS, PVCS, SCCS, CSSC, Aegis, Vesta, or BitKeeper. There will be important new alternatives that I haven't heard of yet (CL 4) but none of them will be as popular as Git (CL 4). A lot of things that aren't traditionally considered source code will be stored in these version control systems, like textual documents.

Perl 5 will still be in wider use than Perl 6 five years from now (CL 2), but neither will be in as wide use as Ruby or Python (CL 3).

DiSo will have a lot of users in 5 years (CL 1) but still won't be as popular as Facebook (CL 3) or MSN Messenger (CL 5) and won't be as popular as some social networking system I haven't heard of yet (CL 3).

General-purpose consumer microcomputers that are popular in the EU will cost between one and four barrels of light sweet crude (CL 3), and computers at that price will still execute single-threaded code at under 5 billion basic blocks per second (CL 5). Today's equivalents (which cost a bit more) can typically do about 8 threads at full speed; five years of Moore's law would make that 32, but I think that computers at that price five years from now are likely to be able to run at least 128 threads at this 5G-basic-blocks-per-second speed (CL 4). Multithreading will provide a decisive performance advantage to

software written in a variety of ways that aren't currently mainstream, such as Erlang's model, Verilog, and array-processing languages (CL 3), rather than providing an equally large performance boost to software written with shared-everything threads and locks, as Herb Sutter predicted. High-performance software being written in this way, combined with a lot of experience with server virtualization, will create an opportunity for new CPU architectures that don't directly support x86, ARM, or MIPS instruction sets --- whether a relatively traditional design like the Tera MTA or something wild like FPGAs, the SeaForth, tagged dataflow machines, or concurrent linear graph reduction machines --- so at least one such architecture will have total market sales 1% or more of the x86-compatibles' market sales (CL 3).

In five years, devices like cell phones, portable computers with embedded radios in which the user does not have ultimate control over the software, will be the way that most people access the internet most of the time (CL 5; this is already true except that most people don't access the internet yet) and will sell ten times as many units per year as the traditional kind of computer in which the user is ultimately responsible (CL 3).

In five years, electoral campaign success in the US at every level except presidential campaigns will be largely determined by word of "mouth" (using electronic communications whose receivers solicit them, rather than paid advertisements) rather than paid advertisements (CL 3). Presidential campaigns will be too (CL 2). In ten years, both of these will be true (CL 4).

In five years, intellectual property restrictions, not terrorism, other aspects of human rights, or agriculture, will be the most controversial issue in international negotiations. (CL 2)

I don't know what to predict about peer-to-peer systems like Vipul's Razor and BitTorrent. I suppose that they will continue to exist, and the kind of people who thought jazz and tango music were scandalous will continue to oppose them (CL 6). And fragile centralized systems like DNS and Google will continue to exist too (CL 6). Whether the mix will change, and how it will affect society, I have no idea.

In the last few years, there have been several hobbyist UAVs and other robots for remote sensing, of which Art van den Berg's glider is probably the most impressive. In the next five years, there will be more of them (CL 5) and some of them will be even more impressive than van den Berg's project (CL 2). Governments will worry and some will make new laws restricting model aircraft (CL 4). Some people will commercialize the technology (CL 3), use it to advance the state of knowledge in some scientific field (CL 3), or do things with it that benefit s lot of people (CL 3). Possible applications include inexpensive high-quality aerial remote sensing, deep-sea exploration, substituting for radio towers and communication satellites, and rapid, inexpensive delivery of small, light items, especially to inaccessible places.

Five years from now, solar energy will be a cost-effective source of electricity in countries that currently derive much of their electrical energy from natural gas or oil (CL 5), because the prices of those commodities will have risen (CL 6). I mean to say that it will be cheaper than those fossil fuels without any government subsidies. It

will probably be cost-effective in countries like the US which get most of their electricity from coal (CL 4), because people will figure out how to bring the cost of solar electric systems down and their efficiency up (CL 4).

Five years from now, other renewable sources of electricity will be cost-effective, too (CL 4). Maybe wind, geothermal, old-oil-well geothermal, tidal, wave power, or even biomass.

Five years from now, digital cameras will surveil all urban public places in the OECD countries (CL 4), probably because random individual people will carry digital cameras around with them and always turned on (CL 3), not just because of cameras placed permanently by the owners of those spaces.

Five years from now, despite the rising prices of fossil fuels (see above, CL 6) long-distance transport will remain affordable for most of the world's food (CL 6), so "locavorism" will not become an economic imperative (CL 6).

Five years from now, China will have experienced at least one recession (two consecutive quarters of negative economic growth) (CL 3), as some aspect or other of its current strategy falters.

Within five years, extrajudicial executions of innocent citizens by the police in both the US and Britain will have provoked considerable furor (CL 3), but the general public in both countries will be in favor of such policies (CL 5).

Within five years, the most-widely-consulted cartographic resource will be a community project like Wikipedia (CL 4), and OpenStreetMap will be the most-widely-consulted community cartographic resource (CL 4). Most important scientific papers in hard-science fields will be published first in open-access media such as arXiv, open-access institutional repositories, or open-access journals (CL 4). The same thing will be true of most important academic papers in less scientific disciplines such as history, psychology, and sociology (CL 2). Significant non-open-access journals will still exist, even in hard-science fields, and will still publish some important new results (CL 5).

Five years from now, most people who enter text into computers will still use QWERTY keyboards or phone keypads (CL 4), not Morse code, Dvorak, speech recognition, or something else exotic.

DHTML will still be a widely-used format for new applications five years from now (CL 4) and will not have been eclipsed in popularity by one of the various new alternatives that are being marketed now (CL 3) such as Silverlight and Flash.

Over the next five years, computer and network security will be a bigger and bigger problem (CL 5) as organized crime makes more and more use of computer security vulnerabilities to get money and power (CL 4).

Five years from now, illegally copied software will still be ubiquitous, even in OECD countries, on computers that are controlled by their users (CL 5).

I'm not sure what's going to happen with agriculture. There are clearly better ways to go about it than the Green Revolution approaches, especially in marginal areas like New Mexico, Iraq, much of Jordan and the West Bank, the llanos of eastern Colombia, and so on. Whether those approaches will catch on, who can say?

Five years from now, there will be at least one change that is as

important as anything I've mentioned here, but that didn't occur to me (CL 3).

Five years from now, Communist China still will not have produced any proprietary software that is widely used in the rest of the world, except software bundled with hardware, (CL 4) and probably no large piece of widely-used free software either (CL 3).

Five years from now, quantum computers will not be a practical alternative to conventional computers for general tasks (CL 6), nor will they be in ten years (CL 5), but in five years we'll see them do some impressive things (CL 3). (It's possible that they will instead fail to work; if that happens, it will reveal a fundamental weakness in our understanding of quantum physics (CL 4).) Ten years from now, they will do some practically useful things (CL 2).

Topics

Why you can't run a diesel engine on water and diesel fuel with electrolysis

Kragen Javier Sitaker, 2019-11-24 (2 minutes)

Is it energetically feasible to run a diesel engine on diesel fuel combusted with oxygen derived from electrolysis of water, driven from the engine itself? Although there's no fundamental thermodynamic reason such a thing is impossible, presumably it isn't, or military diesel submarines would do it.

NEL Hydrogen claims their commercially available electrolysis apparatus electrolyzes hydrogen gas from seawater at 49 kWh/m³ at STP; this presumably means it also produces oxygen from water at about 98 kWh/m³, which at 1.429 g/ℓ is about 68.6 kWh/kg. This is 247 MJ/kg, which is about 6× the specific energy from burning common fuels such as diesel fuel (43 MJ/kg) with oxygen.

It gets worse, though, because burning those fuels requires a much larger amount of oxygen than the fuel: a CH_2 unit, weighing 14 daltons, becomes a CO_2 molecule and an H_2O molecule, using 48 daltons of oxygen. So actually burning diesel fuel gives you 43 MJ per kg of diesel, but only $43 \times 14 \div 48 = 12.5$ MJ per kg of oxygen. So you only get back about 5% of the electrolysis energy when you use the oxygen. That's really, really far from being viable.

So you can't run a diesel engine on water and diesel fuel.

There *are* things that have such a strong affinity for oxygen that you can burn them with oxygen from water for a net energy gain; sodium is probably the best-known example. As far as I know, all of them pose serious practical problems for use in a heat engine.

Topics

World War III is starting (?)

Kragen Javier Sitaker, 2016-10-17 (2 minutes)

World War III is starting.

• in mid-2014 old Vlad said a third world war was imminent because of the collapse of the institutions that had prevented it since 1945, and that precision-guided munitions were as dangerous as nuclear weapons;
• in late 2015 old Frank said that Christmas was a charade because the world was at war and on a path to more war;
• last week Daesh, who Russia is currently bombing in Syria and feuding with the US about, sent a mass-market precision-guided munition against the peshmerga and, although it didn't work properly, it killed two of them anyway;
• Turkey, a NATO member since 1952 and the one with the second largest military (and 90 nuclear weapons which theoretically belong to the US), is refusing to withdraw its troops from Iraqi territory; today old Recep is insulted Iraq's prime minister, and he's demanding that the US deliver up old Fethullah because they say he was plotting the coup that was recently attempted.

Also, not shown in these links, largely as a result of the ongoing war in Syria and Iraq and its refugees, Turkey is suffering an unprecedented crackdown on human rights, and Europe and the US are experiencing levels of xenophobia unprecedented since the lead up to WWII, resulting in a resurgent fascist movement, which incidentally Russia blames for the most recent coup in Ukraine and used as justification for reannexing Crimea, and which got their candidate nominated as the US Republican Party's presidential candidate. And the European Union is splitting apart. Also, two nuclear powers are fighting in Jammu and Kashmir.

On top of all this, related to old Vlad's point about PGMs but not fully explained in it, war tactics have changed so much since the last war between great powers that it is inevitable that some great-power militaries have doctrines that are likely to just get their people slaughtered, as happened to the Iraqi army in the US invasions, or more evocatively the futile bayonet charges in the Russo-Japanese war and World War I. That means that nobody can predict what the outcome of a possible conflict will be, which increases the chances that both sides think they can win it.

Topics

• History (p. 3500) (71 notes)
• Politics (p. 3639) (39 notes)
• Human rights (p. 3510) (6 notes)
• Turkey
• Syria
• Russia
• Iraq

• Daesh

Accelerating Euler's Method on linear time-invariant systems by exponentiating matrices

Kragen Javier Sitaker, 2019-03-24 (updated 2019-04-02) (7 minutes)

These were some notes on a numerical method that at first I was excited about, but which didn't pan out; it was a way to compute trajectories of systems of ordinary differential equations with high accuracy, but it turns out not to apply to any interesting systems (at least, none that I can find so far). It can't possibly work for anything that isn't linear and time-invariant (when augmented with some set of auxiliary state variables), which I think means it is limited to systems whose solutions are sums of complex exponentials.

Basics

The Euler method for approximating solutions of differential equations is just a matter of

```
x[0] = a
for i in range(1, n):
    dxdt = f(x[i-1], t)        # calculate derivative at previous point
    x[i] = x[i-1] + h * dxdt   # extrapolate according to derivative
    t += h
```

which of course has some error if the derivative isn't constant, an error proportional to h^2 and d^2x/dt^2 — according to the professor lecturing yesterday, precisely $\frac{1}{2}h^2d^2x/dt^2$ at some point within the (open) extrapolated-over interval, although this obviously depends on at least the second derivative in question being continuous.

The second-order Taylor method attempts to correct for this by adding in $\frac{1}{2}h^2d^2x/dt^2$ computed at the beginning of the interval, on the theory that the second derivative changes slowly enough through the interval that calculating it at the beginning is probably good enough. And of course you can use third-order or higher-order Taylor methods, although you start to risk oscillations.

The Euler method, which is the first-order Taylor method, has an error proportional to the interval size h^2; the nth-order Taylor method has error proportional to h^{n+1}. Since this error accumulates over many intervals, at a given distance from the initial conditions (as opposed to a given number of timesteps) it's actually proportional to h^n, at least. This still means that if you want to simulate accurately, you need some absurdly tiny interval size.

A variant of Euler's method that I had some success with for the special case of computing sin and cos was to represent the transition function for a timestep as a matrix. It happens that d/dt[sin t] = cos t and d/dt[cos t] = -sin t, so that if we have a vector-valued $\vec{x}$ containing a purported (cos, sin) pair, Euler's method would amount to multiplying it by a matrix M(h):

$$\begin{bmatrix} 1 & -h \end{bmatrix}$$

A problem is already evident in that this matrix's determinant is $1+h^2$, but if h is smallish, that can be a very small error. But if h is smallish, you have to do the multiplication a largish number of times.

However, since it's a matrix, you can compute a power of it efficiently. For example, by squaring it 16 times, you can compute $M(h)^{65536}$, or by squaring it 32 times, you can compute $M(h)^{4294967296}$. This approach allows you to start with an exponentially small incremental angle and thus compute with an exponentially small error proportional to that angle. This is sort of similar to using an order-4294967296 Taylor method, but I'm not sure if it's actually equivalent.

It occurred to me that you could take this approach with the Euler method in general: augment your system's state vectors with extra state variables that are the Nth derivatives of the original state variables (with respect to time), and then use the Euler method in this matrix form in order to get absurdly tiny timesteps.

In some cases, you'll run out of (nonzero) derivatives pretty quickly, or you'll be able to write the new derivatives in terms of the known ones, and you'll get the same exponential-order characteristic as with sin and cos. In other cases, you'll need to stop taking derivatives at some point in order for the transition matrix you're exponentiating to fit into your L1D cache. (Or, since the transition matrix is $O(N^2)$ in the number of elements and will be dense after a few iterations, you might run into a problem of diminishing returns.)

It also isn't going to be efficient for systems that intrinsically have a lot of degrees of freedom, even if their derivatives are simple. For example, if you have a 100×100 grid of values, your state vector has 10,000 variables in it before you start augmenting it with derivatives, which means that your transition matrix is 10,000×10,000, containing 100 million elements; though maybe its initial state is pretty sparse, it gets dense pretty quickly.

In some cases, you'll have derivatives that are polynomials made of multiple terms. It may be advantageous to add each term, without its coefficient, as a separate variable, as an auxiliary variable, rather than adding the derivative as a whole. For example, if $y'(t) = 2y(t) - 5\sin(t)$, then rather than adding a $2y(t) - 5\sin(t)$ row (and column) to the matrix, you can just add a sin(t) row, sticking +2 and -5 in the appropriate matrix cells. (Then you'll need to add a cos(t) row in the next step, and then you're done, because cos(t)'s derivative is -sin(t).) For this purpose you might want to multiply out more complicated derivatives into polynomials or ratios of terms.

In the sin/cos case and in the $2y(t) - 5\sin(t)$ example, the transition matrix was constant, so neither it nor its exponent needed to be recomputed. For more general systems, that may not be the case.

Somewhat annoying example: $y' = \cos t^2$

Suppose $dy(t)/dt = \cos(t^2)$, i.e., $y = \int \cos(t^2)\,dt$. After five more differentiation operations, we have seven functions from which we can construct one another's derivatives, and this allows us to construct a 7×7 matrix for taking an arbitrarily small step of Euler's method:

$y' = \cos t^2$

```
y''   = 2t cos t²
y'''  = 2 cos t² - 4t² sin t²
f     = t² sin t²
f'    = 2t sin t² + WTF
```

Further investigation shows that my 7×7 matrix was bogus and this method does not actually work for the cases I was interested in. It does work for some cases that already have closed-form solutions.

Substantially more annoying example: $y' = \cos y^2$

Very annoying example: $y' = \cos (1/y^2)$

Relation to other methods of numerical integration of ODEs

I think this is not the same as the extension of the Euler method to higher-order differential equations, although it has in common with it the idea of augmenting the original variables with some derivatives; it is not the same as Taylor methods (though it shares with them the idea of correcting the first derivative using higher derivatives) and it has nothing in common with the Runge-Kutta method. It is not a multistep linear method; the transformation matrix (including any power of it) implements a transition from a *single* previous state to a *single* new state.

This is very similar to exponential integrators, but I think it is not the same thing.

Topics

FM chirp sonar

Kragen Javier Sitaker, 2017-07-04 (1 minute)

Linear FM chirp sonar is cool.

If you emit a sound chirping linearly from 6 kHz to 10 kHz over the course of two seconds, and it bounces off some object 1.72 meters away through air, the 10-ms-delayed echo signal will be 20 Hz lower than the outgoing signal; if the amplitudes are within an order of magnitude or so, this will result in audible 20 Hz beating between the outgoing signal and the echo. If you jump back down to 6 kHz at the end of the two seconds, you'll have a 10-ms length of time where the beating sound disappears, but that's still a 99.5% duty cycle.

(This is a chirp rate of 2 kHz/s; 343 m/s is 172 m/s in round-trip meters, so this works out to about 11.66 Hz/m.)

You should probably be able to distinguish by ear a difference of ½ Hz as long as it's more than about 10%; this would limit you to 10% distance resolution at far distances and 43 mm distance resolution at near distances. A higher chirp rate would give you tighter distance resolution, but you would also run out of audible frequencies sooner.

Writing a C program to generate such a chirp took a few minutes, and in practice I don't seem to be able to do this by ear.

Topics

Constant space flexible data

Kragen Javier Sitaker, 2018-04-27 (5 minutes)

One of the difficulties with the Lisp memory model used by most mainstream programming languages today is that memory usage is fairly unpredictable, and allocation is fairly ubiquitous. This means that writing code that cannot fail is pretty difficult.

The nested-objects memory model used by COBOL and used with some modification by C avoids this problem, but it's much less flexible. Every object has a fixed size, so you're always running off the ends of buffers, which is another way for your code to fail.

The Z-machine memory model used by Zork is an interesting alternative. Z-machine objects are normally all allocated at compile time, but linked together into a linked-list nested container hierarchy reminiscent of the DOM at runtime, and each object has a sort of property list which you can mutate but not add to or remove from. In games like Zork the container hierarchy is used for possession and location: your book might be contained in your bag, which is contained in you, who is contained in the entry hall, which is contained by the universe and which also contains a handkerchief and two exits.

In the Z-machine, objects are never created or deleted, but merely move from one container to another. Moreover, inserting the object into its new container is a constant-time operation involving the setting of three pointers. Because none of those three pointers is "prev", removing an object from its current container is a variable-time operation. Here's a graphviz diagram:

```
digraph zorkish {

    node [shape=record, label="{\N|{<parent>parent|<child>child|<sibling>sibling}o
o}"];

    universe:child       -> hall        ; hall:parent                -> univers o
o e;

    hall:child           -> player      ; player:parent              -> hall   o
o ;

    player:child         -> bag         ; bag:parent                 -> player o
o ;

    bag:child            -> book        ; book:parent                -> bag    o
o ;

    player:sibling       -> handkerchief; handkerchief:parent        -> hall   o
o ;

    handkerchief:sibling -> exit1       ; exit1:parent               -> hall   o
o ;

    exit1:sibling        -> exit2       ; exit2:parent               -> hall   o
o ;
```

```
}
```

You could imagine generic functions that work over such a structure. For example, you could write a filter function that cleaned any contents out of a container that didn't pass a given criterion, moving them into a given wastebasket, or a sort function.

We could modify the structure somewhat. We could use these nodes to represent relationships between entities rather than entities in themselves — originally we had only a "contains" relationship, but we could include relationships such as "has as title", "has as acronym", etc. This gets us quite close to the Python or Lua or JS data model, but if we adopt that data model directly, we have the problem that it doesn't allow duplicate properties in a dict.

Consider this example JSON:

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {

            "para": "A meta-markup language, used to create markup languages such
o as DocBook.",
            "GlossSeeAlso": [
              "GML",
              "XML"
            ]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

It contains a list, GlossSeeAlso. Suppose we represent that with multiple values for the property GlossSeeAlso of the GlossDev entity, by which I mean the object that is the GlossDev property of the other entity:

```
{ ...
  "Acronym": "SGML",
  "GlossDev": {

    "para": "A meta-markup language, used to create markup languages such as DocB
ook.",
    "GlossSeeAlso": "GML",
```

```
    "GlossSeeAlso": "XML"
  }
}
```

This change to the data model eliminates the need for separate lists. In effect, all property values are implicitly lists.

That fragment might look like this:

```
digraph arcs {
    node [shape=Mbox; label="\"\N\""];
    GML XML SGML paraval;

    paraval [label="\"A meta-markup language, used to create markup languages suc
h as DocBook.\""];
    {
        node [shape=record, label="{type \N|{<kid>kid|<sib>sib}}"];
        "root":kid       -> Acronym;
        Acronym:kid      -> SGML   ; Acronym:sib      -> GlossDev     ;
        GlossDev:kid     -> para   ;
        para:kid         -> paraval; para:sib         -> GlossSeeAlso ;
        GlossSeeAlso:kid -> GML    ; GlossSeeAlso:sib -> GlossSeeAlso2;
        GlossSeeAlso2 [label="{type GlossSeeAlso|{<kid>kid|<sib>sib}}"];
        GlossSeeAlso2:kid -> XML;
    }
}
```

This change to the data model means that removing an arc from an entity (and not deleting it) or adding an existing arc to an entity can no longer fail. Furthermore, fetching a property from an entity cannot fail either; it can only return an empty list.

In a 16-bit world, these nodes probably take up 6 bytes each. If we expand the "kid" pointer to be a hash table of 4 or 8 entries, we lose the sequencing among different properties, but we can still retain sequencing within a single property. We use more memory, though still much less than Python, but property lookup becomes instantaneous on small objects.

Topics

- Programming (p. 3658) (286 notes)
- Memory models (p. 3572) (13 notes)
- Z machine (p. 3781) (3 notes)
- Zork

Matrix exponentiation linear circuits

Kragen Javier Sitaker, 2018-12-18 (4 minutes)

Reading Physical Audio Signal Processing, I was struck by [the]matrix equation for the impulse response]o of the linear state-space model:

$$h(n) = D \text{ if } n = 0 \text{ else } CA^{n-1}B$$

Here D is the direct coefficient matrix from inputs to outputs (not mediated by the system state), A is the state space transition matrix, B is the matrix of input gains (inputs to their effects on state variables), and C is the matrix of output gains.

The equation looks obvious, but suddenly I understand what a vibrational mode is — it's an eigenvector of A, so it survives A^{n-1} unchanged except in phase and magnitude — and I see how to use matrix exponentiation to speed up the simulation of linear circuits.

Maybe I've actually done this before, actually. Specifically, to cut down on errors due to time discretization, you can start with the equations for some time step size, such as a millisecond — maybe $dV(C_1)/dt = 5\ V(L_3) + 2\ V(C_2)$ — so in a millisecond you have .005 and .002 entries in your matrix. But of course $V(L_3)$ and $V(C_2)$ are changing during that millisecond, which generates errors proportional to their second derivatives and the square of the time interval. So, a thing you can do is to divide these deltas by, say, a million, getting the state-transition matrix for a nanosecond — the matrix becomes six orders of magnitude closer to the identity matrix. But simulating the circuit nanosecond by nanosecond instead of millisecond by millisecond would run a million times slower.

So — and this is the clever part — you *square the matrix* to get a matrix for the 2-ns change. (Disregarding the influence of B for the time being, that is.) This matrix is not quite the same as the one you would have gotten by dividing by 500 000 instead of a million, because it includes second-order effects; it precisely captures the effect of doing the 1-ns change twice. Then you square it again, 19 more times, and you have a new matrix for 1.048 576 milliseconds, which allows you to simulate just as fast as before, but with discretization errors that are 12 orders of magnitude smaller.

Taking a really simple example, consider an object whose coordinates are (cos t, sin t). dx/dt = -y and dy/dt = x, so we could try using the matrix

```
[[1 -1]
 [1  1]]
```

which will of course produce immense errors immediately, generating an exponential spiral through the powers of 2. But suppose we divide the derivatives by a factor of 1048576, to get the transition matrix for about a microsecond:

```
array([[ 1.00000000e+00, -9.53674316e-07],
```

```
       [ 9.53674316e-07,  1.00000000e+00]])
```

And then we execute the above procedure. Now we have this:

```
array([[ 0.54030256, -0.84147139],
       [ 0.84147139,  0.54030256]])
```

The correct values of cos(1 rad) and sin(1 rad) are closer to 0.5403023 and 0.8414710, but being correct to six decimal places is nothing to sneeze at. You can make a point orbit with this rotation matrix for many, many generations before it has spiraled outwards much.

(Is this just CORDIC?)

Of course, in this case, we could have corrected the spiraling behavior by noting that the matrix determinant was 2 and thus dividing by $\sqrt{2}$, giving us:

```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

And that is definitely a rotation matrix, and it doesn't lose or gain "energy" (which is squared radius in this case), but it's also definitely not a rotation of one radian.

Topics

Whistle detection

Kragen Javier Sitaker, 2018-06-06 (updated 2018-12-02) (18 minutes)

I can whistle from about 600Hz up to about 1600Hz. There is often substantial unintentional amplitude modulation on my whistle around 25–100 Hz. The vibrations can't ramp up or down by 10dB in less than about 8 cycles, even when I'm releasing a tongue stop, and more typically take 20 or 30 cycles, suggesting a Q factor of somewhere around 10–20. Second, third, and fourth harmonics are detectable, but weak; even the second harmonic is 52dB down from the fundamental. I can easily ramp up from 800Hz to 1600Hz in 1.3 seconds. Doing the same ramp in 100 milliseconds is feasible with more effort.

(Semitone resolution is a Q factor of about 17, so using a linear filter with a lower Q factor would fuzz out frequency selectivity too much to be useful.)

A particular 70-millisecond segment of low whistle had a frequency peak at 615 Hz at -29.3 dB, falling off to -35.6 dB at 610 Hz and -38 dB at 628 Hz. This suggests a Q factor of around 60 during the whistle itself. (Maybe even a bit higher.)

This suggests that the "window size" I should be looking at to detect a whistle could be up to about 20 or 30 cycles or milliseconds in length, and I could band-pass filter the signal to a bandwidth of about 1000 Hz as a first step. Then I could measure the distance between zero crossings or calculate some autocorrelations or something, or run a PLL.

Using high frequencies like this reduces the signal detection latency.

Efficiency questions

I'd like to run this all on an Arduino. So, to reduce the computational cost of this, probably I should start by downsampling to about 4–8ksps. (The Arduino could perhaps capture natively at this rate.) Then, maybe I could resample to 2ksps with downconversion from 600Hz to baseband, but maybe that's not a good idea — among other things, it might increase latency.

There are a variety of possible things you can do from there. You can run a software PLL. These can be quite simple and efficient; this one does 7 simple operations per sample and sort of works to track my voice frequency; you need a few more operations to detect presence or absence:

```
/* A PLL in one line of C. arecord | ./tinypll | aplay */
main(a,b){for(;;)putchar(b+=16+(a+=(b&256?1:-1)*getchar()-a/512)/1024);}
```

You can do STFTs, but that's never going to be fast. (At least within barely more than a single octave, you don't have a terrible tradeoff of window width; you can just use a window of about 30 ms). You can count zero-crossings, but that throws away most of the information in the signal. You can calculate the autocorrelation function for particular frequencies of interest, or when multiplication is slow, the sum of absolute differences. Or you can use particle

filters.

Particle filters

A simple particle-filter-based scheme could measure prediction error of each sample based on various candidate lags, taking the last 20 or 30 cycles (or whatever) as the predictor. If you used 32 cycles, you could calculate the prediction like this:

```
s16 total = 0
  , lag = p.lag  // 12.4 fixed-point
  , lagi = 0
  ;
for (u8 i = 0; i != 32; i++) {
    total += latest_x[(lagi += lag) >> 4];
}
return total >> 5;
```

But that is kind of shitty because you have to do shit 32 times: 64 16-bit additions, 33 4-of-16 bit shifts, 32 indexed byte fetches. It'll take hundreds of cycles to calculate the prediction for a single particle. (Also, you get some aliasing, because you're effectively resampling your signal with nearest-neighbor resampling.)

Incremental prediction

A better approach might be to incrementally update the totals. I will explain.

Suppose we're at 8ksps and our range of frequencies of interest goes from 700 to 1400 Hz. Then each cycle is from 5.7 to 11.4 samples; 32 cycles are from 183 cycles to 370 samples. A frequency in the middle of the range like 1000 Hz will have a cycle every 8 samples. So its current prediction is almost the same as the prediction from 8 cycles ago — it will be adding up almost the same samples. If it cycles through an array of 8 totals, it can update the current total by subtracting the sample 32 cycles ago that fell off the end of the window, then adding the sample from 1 cycle ago. Then it can make the prediction based on that.

The same kind of logic applies for cycles of exactly 7 samples (1143 Hz), exactly 9 samples (889 Hz), exactly 6 samples (1333 Hz), exactly 10 samples (800 Hz), and exactly 11 samples (727 Hz), although some of these will need an array of more totals, up to 11, while others will cycle through less.

Non-integer numbers of samples are trickier, since we don't want to count back by strides of 8.2 samples from the present — by the time we get even 3 strides back, we're no longer looking at the same sample that we looked at the last time we were at this phase, so subtracting the sample 32 strides back from the total makes no sense — it wasn't part of the total to begin with. The solution that occurs to me is to maintain in each phase, associated with the running total, two accumulators like the variable `lag` in the above code, one for the beginning of the window and one for its end. So then the update code for a single phase accumulator becomes something like this:

```
return (p->total += x[(p->head += p->lag) >> 4 & xmask]
                 - x[(p->tail += p->lag) >> 4 & xmask]) >> 5;
```

This is 9 operations per particle update instead of hundreds. However, it omits the logic to select the proper phase, which I think sometimes needs to update two phases.

4 bits of fractional sample allows resolving between lags of 8 samples, 8.0625 samples, and 7.9375 samples — 1000 Hz, 992.2 Hz, and 1007.9 Hz, respectively. This is excessive precision for 32 cycles. We probably can't do better than about half a cycle during our window of frequency precision, which is to say 4 samples out of 256, which means that we really only need 3 bits.

An alternative that avoids the need to track the tail is just to exponentially decay the totals, maybe in a cascade of two or three; that's what the PLL given above does.

Sinusoidal phase detection for PLLs

If you change the particle-filter approach to try to follow the signal by modifying the period of a particle, rather than by spawning new particles with slightly different periods, you end up with a PLL. A simple thing to do would be to look at the prediction error and compare it to the derivative. If the prediction error is of the same sign as the derivative (of either the signal or the prediction), you're falling behind the signal and need to speed up. If it's of the opposite sign, you're getting ahead of the signal and need to slow down. And the prediction error from the phase error will be proportional to the derivative, which has two contradictory implications:

- When the derivative is large, you should expect a large prediction error from a small phase shift, so you should consider prediction errors less important.
- When the derivative is large, the prediction error is less likely to be due to noise and more likely to be due to an actual phase shift, so you should consider prediction errors more important.

I think #2 wins out.

I think all of this will only really work right if your waveform is pretty simple, though, like somewhere in between a triangle, square, sawtooth, and sine wave. Waveforms with weird shapes with negative local maxima and positive local minima will cause trouble. Fortunately, we're kind of filtering those out anyway in this case, and my whistles are 99.75% sinusoidal, as I said at the beginning, even without bandpass filtering.

So it might make sense to just run a sinusoidal oscillator instead of using an average of previous samples as the prediction. (But maybe using previous samples is cheaper.)

A simpler PLL phase detector supporting weird waveforms

Suppose we have an estimated waveform which is the average of the last few cycles (say, a simple moving average of 16 cycles), and we want to know if we're early or late on the phase. Well, we can subtract the sample x_i from the estimate at this point in the waveform e_i and get an estimation error: $e_i - x_i$. Suppose it's positive. Does that mean we're early or late? It depends on the average derivative over the relevant timspan. But what is the relevant timespan? It depends on how big the error is; if the error is small, it's short, and if the error is large, it's long.

It would be relatively straightforward to average derivatives over different spans: for example, (e[i + 4] - e[i - 4]) >> 3, and (e[i+8] - e[i-8]) >> 4, and so on. But perhaps a simpler solution, given that we have a whole estimated waveform, is to look forward and backward in time for the next and previous place the curve crosses x_i. Once we find it, it directly gives us a phase error estimate for that sample. We could get more elaborate and guess how reliable that estimate is, based on how steep the slope is at that point and how many other nearby crossings there are, and indeed we could even compute a whole probability distribution for the phase error and use it to update our prior probability distribution, but it's probably adequate to just use some kind of moving average or median of the phase-error estimates.

Transfer oscillator

An approach sometimes used in analog electronics to precisely measure unreasonably high frequencies is the "transfer oscillator technique", in which "you phase-lock the nth harmonic of a VCO to the input signal, then measure the VCO frequency and multiply the result by n," according to Horowitz & Hill. It seems like you could also do something like this in the digital realm. Suppose you're trying to detect a whistle in the 600–1600Hz range, and you have a candidate frequency, say 1400Hz. You can do a phase detection every *four cycles*, which is to say at 350Hz (every 2.86 milliseconds) to see which way you're slipping out of phase. This could reasonably be done with a modern low-power fast-wakeup microcontroller that goes to sleep in the middle.

It seems like this approach may have some big drawbacks, though. One is that, with the Q factor of 10–20 of my mouth's whistle, you are going to have a really hard time detecting short whistles this way.

Flying saucer sounds

Amplitude-modulated whistles (by vocalizing with the larynx while whistling) are something else entirely; they have the strongest peak at the whistle frequency, but in one signal I looked at, the sum and difference components were only about 10dB weaker, while being 20dB stronger than the valley in between. And the sum and difference components for the second harmonic of the vocal sound, while another 20–30 dB weaker than the upconverted first harmonic, were still 10–20 dB stronger than the valleys in between. The frequencies in this case were 1349, 1476, 1604, 1732, and 1866 Hz, with spacings of 127, 128, 128, and 134 Hz, respectively. The most striking feature, though, is that these inharmonic overtones moved together with the whistle when it changed frequency.

I did try changing my larynx frequency without changing the whistle frequency, but I wasn't successful.

An amusing note: if I could whistle more consistently, with a Q of, say, 60, I could undersample the signal in such a way as to alias the notes down into a much smaller bandwidth than the 1000Hz they occupy now. Each note would only need 40Hz or so, so all 17 would probably fit in a bit under 700 Hz, requiring just 1500 samples per second. (Because they're not linearly spaced, there's some wasted spectral space.) There are a couple of downsides of this: first, a whistle that goes slightly astray from its pitch would be detected as a different note, more or less at random; second, the required Q of 60

also applies to the filtering, and I think it might impose a really large latency; third, you need to bandlimit the signal to just the whistle band, and broadband noise within the whistle band will be impossible to remove.

Musical notes

In A440 12-tone equal temperament, the notes in the 600–1600Hz range are:

- 622.25 Hz
- 659.26 Hz
- 698.46 Hz
- 739.99 Hz
- 783.99 Hz
- 830.61 Hz
- 880.00 Hz
- 932.33 Hz
- 987.77 Hz
- 1046.50 Hz
- 1108.73 Hz
- 1174.66 Hz
- 1244.51 Hz
- 1318.51 Hz
- 1396.91 Hz
- 1479.98 Hz
- 1567.98 Hz

A quick whistle test in front of my laptop with Audacity had me whistling a short melody (from the Myrath song "Endure the Silence") at 1130Hz, 1137Hz, 1235Hz, 1127Hz, 1103/1072/1075 Hz, 1072Hz, 973Hz, 898Hz, 849Hz, 752Hz, 728Hz, and 781Hz, which is about as far from A440 as it's possible to get. As the 1103/1072/1075 indicates, the spectral peak frequencies reported by Audacity are perhaps somewhat noisy. In 12-TET semitones relative to the frequency of the first note, this is 0.00, 0.11, 1.54, -0.05, -0.86, -0.91, -2.59, -3.98, -4.95, -7.05, -7.61, -6.40. While it's possible I'm 46 cents out of tune on some of the notes, I suspect that another reasonable hypothesis is that Myrath is using 24-TET.

Are these frequencies really that unreliable? Upon examining the (last) 1072Hz note again, it appears as 1077Hz, or -0.83 semitones from the first note, so Audacity's noise in that case amounted to 3 cents. Upon further examination, 33 cycles in the middle of it lasted 1352 samples (at 44.1 ksps), zero-crossing to zero-crossing, or 30.658 ms, giving the frequency there as 1076 Hz. 20 cycles at a later point in the same note lasted 809 samples, giving a frequency of 1090 Hz there. 962 samples earlier on in the note contained 24 cycles, giving a frequency of 1100 Hz. So actually they are pretty reliable.

After applying a gentle 12dB/octave rolloff below 600Hz and 1600Hz, the vast majority of the signal power was in the whistle; it was typically 30dB louder than anything else, despite the major traffic noise. Second and third harmonics were still visible.

How's my rhythm?

I was tonguing the notes to get very definite start times.

The time interval from the start of the first note to the start of the

second was 21167 samples. From the second to the third was 21746; from the third to the fourth was 21312; from the fourth to the fifth was 21505; from the fifth to the sixth was 25652; from the sixth to the seventh was 11139; from the seventh to the eighth was 10752; from the eighth to the ninth was about 10728, though it's fuzzy; from the ninth to the tenth was 10873; from the tenth to the eleventh was 10536; and from the eleventh to the twelfth and last was 11210. If we take our nominal quarter-note time as 21505 samples, which is the median of [21167, 21746, 21505] and 2*[11139, 10752, 10728, 10873, 10536, 11210], then the errors here are -1.6%, +1.1%, 0, +3.6%, -.05%, -0.23%, +1.1%, -2.0%, and +4.3%, except that 25652 is 1.193 beat times, which is a pretty weird number. The other errors might be just as much from the mouse selection skillz I'm using in Audacity to mark the time intervals as from actually being off the beat.

The notes mostly each ended a bit before the succeeding note, but I think there was a significant amount of room echo.

The great thing about this is that it provides *plenty* of margin of error for identifying beat timings. If the standard deviation of being on-beat is ±2.0%, as it is in this case, then we can be pretty confident in identifying which beat a note is starting on.

Topics

- Digital signal processing (DSP) (p. 3419) (60 notes)
- Small is beautiful (p. 3714) (40 notes)
- Audio (p. 3331) (40 notes)
- C (p. 3359) (28 notes)
- Ubicomp (p. 3761) (12 notes)

Tapered thread

Kragen Javier Sitaker, 2015-09-03 (updated 2019-06-10) (4 minutes)

Was just watching Dan Gelbart's video on building large structures with adhesives (http://youtu.be/EeEhS3zmnDg), and he demonstrated a flexural clamp he uses: a waterjet-cut slot with a tapered round hole drilled and tapped in the middle of it, parallel to the edge of a rectangular hole. Screwing a tapered thread into the round hole expands it, curving the edge of the rectangular hole inwards. Incredibly simple and with an unbelievably huge mechanical advantage.

NPT, the US standard for tapered pipe threads, is tapered at 62.5mm/m, or 1:16. A common ½" pipe (inside diameter, since that's what determines the pipe's capacity) is tapped at 14 threads per inch, or 0.55 per millimeter; one full rotation of the pipe will advance it into or out of the hole by 1.81mm. You could very reasonably use a 10cm-radius lever arm or gear to turn the pipe, which would make a full rotation 62.8 cm along the outside, which is a mechanical advantage of 346 to the linear axial movement of the pipe. Divide that by the 1:16 taper, and the total mechanical advantage is 5540.

That means that you have a very simple device with two moving parts and no backlash that can amplify your tactile positioning precision of perhaps 100µm down to 18-nanometer resolution (over a total travel of maybe 2mm), and perhaps just as important, amplify your perhaps 1kN bodily strength into 5.5 MN, a bit above 600 tons — although, again, with a travel of only about 2mm.

The pipe won't stand up to that much, though, even if you stuff it with concrete. Regular Schedule 40 ½" mild steel pipe has a 2.77mm-thick wall, maybe 19mm of length on the thread, but you might be bearing that whole force on 7mm of width across the 21mm total width at some point in the movement, only 147 mm², for about 37 GPa. ASTM A36 mild steel can have as little as 152 MPa (0.152 GPa!) of compressive yield strength. You can improve the situation somewhat by using bigger pipes, harder steel (or 2.5 GPa cast iron? or carbide?), and having less travel so the force is better distributed, but basically I think the limitation is going to be the strength or hardness of the metal.

(In the following video, Gelbart says most steels yield at 10 or 20 tons per square centimeter, which is actually a GPA or two. Alumina can get up to 5.5 GPa. I don't think anything makes it to 37 GPa.)

With two of these positioners arranged at right angles to push on arms, rather than to squeeze a hole, you could position a stylus in two dimensions to 18-nanometer precision, for example for scribing microfilm; with three, you could position a stylus in three dimensions. There are a lot of possible sources of positioning error in this system, such as lubricant entrainment, vibration, and thermal expansion, but I think you can probably make it work.

One of the more interesting things I think you can do is emboss a pattern from a hard die into a soft pattern material, using the flexural positioner to drive a stone or whatever, or to press a sheet between two dies (or a die and a soft material). In particular, this might be a feasible way to rule small optical diffraction gratings without the

difficulties attending the standard ways of doing it.

In a later video http://youtu.be/nCfVupLt-Pk/ Gelbart claims that silicone mold-making material has "molecular" resolution, which would make it an ideal inexpensive material for further reproductions.

Topics

Immediate-mode PEG parsers in assembly language

Kragen Javier Sitaker, 2019-12-10 (updated 2019-12-11) (21 minutes)

I think I've reinvented an interesting design for how to do parsing efficiently and straightforwardly in machine code; it seems eminently workable and quite efficient, at least for grammars without an unusual amount of ambiguity, probably achieving parsing speeds of dozens of megabytes per second with easy-to-write assembly code. It's an approach to parsing analogous to the IMGUI approach to building user interfaces --- or, from another point of view, a slight tweak on traditional recursive-descent parsing that adds some expressivity.

I've been working with Meredith Patterson and Andrea Shepard on the Hammer parsing engine. This isn't intended as a proposal for Hammer, but it is certainly inspired by Hammer. Andrea's been working on an LLVM backend for Hammer, and that, along with the difficulties Jeremiah Orians reports in writing interpreters in assembly, has led me to think about these problems.

In Immediate mode productive grammars (p. 898) I wrote about how to use a similar design for *bidirectional* parsing *and deparsing*, that is, serialization. Deparsing is not considered in this note. I also considered a vaguely similar idea for compiling a syntax tree into machine code that iterates over it in Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation (p. 1508).

The basic correspondence

Consider the design of Hammer. Hammer is a relatively traditional parsing combinator library in C; you call a bunch of Hammer functions to incrementally build up an object graph representing your grammar, in much the same way a user of a traditional widget toolkit calls widget-creation functions to build up an object graph representing their GUI. Is there an alternative that would look more like IMGUI?

The fundamental combining operations of PEGs are greedy choice, concatenation, indirection, and negation. These are pretty close to the usual programming operations of conditionals, sequencing, subroutine call, and iteration; although negation and iteration do not correspond, the others do. It would be nice to directly implement concatenation of languages with sequencing of programs, so that you could compile the tiny language `'h' 'i'` into, for example:

```
mov $'h, %al
call letter
mov $'i, %al
call letter
```

That much is fairly easy to do: the `letter` subroutine checks the next byte of input, explodes if it's not 'h' or 'i', and otherwise advances the input pointer, something like this:

```
letter:
    mov (%esi), %cl
    cmp %al, %cl
    jne explode
    inc %esi
    clc
    ret
```

This takes seven instructions per character on the happy path; you probably want a sentinel EOF character which, if you want to recognize it, gets handled specially. (The `clc` is explained later.)

Choice and backtracking

Backtracking can't be done quite as simply as 'h' 'i' / 'j', because somehow the failure of 'h' needs to know to backtrack to the 'j' instead of somewhere else.

Well, it *could* be done that simply by no-opping the intermediate operations, just as in IMGUI toolkits you can call a bunch of functions that don't draw anything because they are off the screen or something, or just as you can null out the effect of all future writes to a file descriptor by closing it (they just get back EBADF). All your side effects need to be indirected anyhow in case you have to backtrack (see the section "Data store" below). But I think a more efficient and more comprehensible approach is something more like this, if I can figure out how to make it work:

```
    jc 1f
    mov $'h, %al
    call letter
    mov $'i, %al
    call letter
1:  call backtrack
    jc 1f
    mov $'j, %al
    call letter
1:  ret
```

The idea is that, on entry to the parser subroutine, the carry flag is initially cleared; but, if 'h' or 'i' fails, the subroutine is restarted with the carry flag set, which leaps to the call to `backtrack`. That redoubtable subroutine clears the carry flag, updates the backtracking state, and returns. If instead it is reached with the carry flag *clear*, because execution fell through all the `letter` calls successfully, it knows that its immediate caller has succeeded, and so it returns *from* that caller, never returning *to* it.

The disadvantage of this approach is that it requires the parsing subroutine to be called specially, via a magic parsing-subroutine-calling subroutine which saves the initial backtracking state, clears the carry flag, and upon the subroutine returning, discards that backtracking state. (Although this would be needed in any case if you want Packrag memoization.)

A less magical approach would look like this instead:

```
    call choice
    jc 1f
```

```
    mov $'h, %al
    call letter
    mov $'i, %al
    call letter
1:  call nextchoice
    jc 1f
    mov $'j, %al
    call letter
1:  call endchoice
    ret
```

This requires you to bracket your alternatives with `choice`/`endchoice` calls, which will return twice if backtracking is needed but are otherwise ordinary. Once one of the choices succeeds, the following `nextchoice` calls skip over the following bodies (by setting the carry flag, as before) until `endchoice` is reached.

This approach has the disadvantage that you can forget the `endchoice` call, particularly if you have multiple return paths, and it requires writing a couple more instructions per choice. It has the advantages that parsing subroutines can call each other directly, you can nest choices, and failure propagation to parents is faster and more straightforward to implement.

This use of the carry flag is the reason that the `letter` routine given above clears the carry flag; otherwise a carry left over from its `cmp` instruction might result in spurious reports of parse failures.

(I think this is more or less the aproach of the Warren Abstract Machine used for Prolog, but I don't understand the WAM, so I might be wrong about that. I should probably read about it to see if I'm reinventing it in a way that is known to be broken.)

Negation

Negation requires the same kind of backtracking transaction to be set up that choices do; the difference is that negation always aborts the transaction, but it fails if any of the choices succeeded. I think we can make this work with a `negatechoice` subroutine; for example, to parse `!keyword identifier`:

```
    call choice
    jc 1f
    call parse_keyword
1:  call negatechoice
    jc 1f
    call parse_identifier
1:  ret
```

Repetition

I'm not *absolutely* sure of this, but it's a belief I have about PEGs. Because repetition and alternation prune alternatives in the same way, strictly speaking, if you have recursion, repetition is unnecessary; these two grammars parse the same language:

```
aab <- 'a'* 'b'.
aab <- 'a' aab / 'b'.
```

But the first one only saves a single backtracking state, which gets repeatedly updated, while the second one saves an arbitrarily large stack of useless backtracking states. I'm not sure it's possible to implement that behavior in terms of the `choice`/`nextchoice`/`endchoice`/`negatechoice` primitives described earlier, so there might need to be a `repetition` subroutine to in effect commit the in-progress transaction so far and start a new one:

```
    call choice
    jc 1f
2:  mov $'a, %al
    call letter
    call repetition
    jc 1f
    jmp 2b
1:  call endchoice
    mov $'b, %al
    call letter
```

I'm not sure if you can just use an `endchoice`/`choice` pair instead; I think it will do the wrong thing by propagating the failure of the last greedy advance. There might be a tweak that makes it work.

More terminals

Of course in practice you will want things like Hammer's `h_literal` which matches a literal string and `h_ch_range` which matches any byte within a byte range. There are lots of ways these could be handled; for example:

```
    mov $('a | 'z << 8), %ax
    call range

    .data
t:  .ascii "const"
t_end:
    .text
    mov $t, %eax
    mov $(t_end - t), %ecx
    call literal
```

Byte classes, like regexp character classes, are also likely useful; these would have a string either in memory or in a register of the bytes they could accept.

Hammer supports parsing by bitfield rather than by byte, but I think a significant number of things don't require that.

Data store

Generally you want to build up some kind of AST or something as you parse; the failure of a "transaction" ought to efficiently backtrack whatever you did when you were building that AST. One reasonable way to handle this is to build up the AST in a pointer-bumping allocation arena (like Hammer's arena allocators, GCC's obarrays, or the Java GC's nursery) and bump the pointer back when a transaction fails. This is only safe if all the pointers into the backtracked part of the arena also become inaccessible, so it's also necessary to supply some

kind of variable-bindings construct that gets backtracked too.

Maybe the following API would work well for within-transaction code:

• new(*nbytes*) -> fresh *pointer* to a node of *nbytes* bytes;
• dup(*pointer*, *nbytes*) -> fresh *pointer*; creates a copy of the *nbytes* at the old *pointer* so that you can modify it in the new transaction;
• put(*opaque*, *pointer*); updates the "variable" "named" by *opaque* to have the value *pointer*;
• get(*opaque*) -> *pointer* to the current value associated with *opaque* (the last thing you called set() with for that value of *opaque*), or 0 if none exists.

You can have a relatively small number of distinct *opaque* values ("variable" "names"), for which you can use function pointers or something else guaranteed to be unique; this is pretty similar to thread-local storage. They can be stored in an alist in arena nodes. If you have five of them, for example, each associated with a five-word node, you have 25 state variables for your parse. (Maybe you'd want to bloat the alist a bit to bound the depth to which you'd have to search it to the number of variables, or twice the number of variables, or something.)

This approach permits the backtracking of the arena state by resetting two pointers, the head of the alist and the allocation pointer.

With such an implementation, you can also easily supply an additional backtrackable operation that is probably frequently useful in parsers:

• pop(*opaque*); undoes the latest put to *opaque*, restoring it to its previous value. This could be useful, for example, for indentation levels. This requires the bindings for each variable to be linked together in a separate list from the list that unites the current bindings of all the variables.

To some extent you might be able to entirely avoid using the variable store, since you can return pointers to arena nodes in registers.

In addition to the calls above for code *within* transactions, you need the usual begin/abort/commit calls for transaction management.

This allocation approach is, however, entirely incompatible with Packrat memoization. The transactional variable-store thing could maybe survive, but the deallocation thing can't, because the memoized return values of nonterminals invoked from within a transaction that later failed would need to survive. Memoizing a nonterminal that might depend on the state of the variable store would be sort of dubious, too, though.

Overall parsing context

I think we can do most of this entirely in registers, which should speed it up considerably. You need:

• PC, which tells you where you are in the parsing of the current nonterminal;
• the stack pointer, which tells you what other nonterminals you're in the middle of parsing, when you get done with this one, with their PC values;

• the input pointer, assuming the data to be parsed is in RAM;
• the backtracking stack pointer, which might be a pointer into the PC stack;
• the allocation arena pointer;
• the current variable bindings pointer;
• the "skip" flag (the carry flag above), which indicates that we want to skip a choice, either because it has failed or because an earlier choice has succeeded.

That's six or seven registers, only three or four of which are general-purpose, plus the skip flag. This should fit even the impoverished i386 register set.

Of these, all but the skip flag would be saved and restored for backtracking. This suggests that the context-save code, part of `choice` and `nextchoice`, might look something like this on i386:

```
pop %eax              # pc
push %ebp             # backtracking stack pointer
push %esi             # input pointer
push %edi             # allocation arena pointer
push %ebx             # variable bindings pointer
push %eax             # pc
mov %esp, %ebp        # point backtrack stack to newly allocated state
push %eax
clc
ret
```

A similar sequence of about ten or so instructions would be needed to backtrack. This suggests possible parsing performance in the neighborhood of 20 clocks per byte on modern CPUs, which could conceivably reach speeds of hundreds of megabytes per second, but probably won't.

(I think there might need to be a bit in that backtracking state record where we can note that it corresponds to a choice that has *succeeded* and we are just skipping over the remaining branches. Also, there isn't enough information there to tell when our arena is full; presumably arena allocations need a check for that unless it's okay for them to just crash or overwrite other data when it gets full, which is how execution stacks are often handled, I guess. So we might need an additional register for that.)

Arbitrary computation

Although we have to be careful not to have any effects we might wish we hadn't had if we backtrack, we can freely intersperse pure computation with the parsing, and even use it to decide whether or not to continue with a parse or fail. This computation can freely read and write the transactional data store described earlier, which might include information like type information for identifiers or indentation levels.

In particular, we can consult some other data structure to decide what to try to parse; for example, an in-memory grammar. We can even translate the choice backtracking to traversals of that in-memory grammar, as long as we keep track of our traversal state in something that backtracking restores correctly, such as the data store. If the backtracking stack is stored on the execution stack, as in the example

code above, then your traversal will need to recurse in order to backtrack properly.

Another less pure thing we can freely do is to skip the read pointer to arbitrary places in the input text; the read pointer will be restored automatically if we backtrack, just as it would for normal sequential reading. This is interesting for things like parsing PDF files, which store byte offsets in their structure for random access to the object tree. (This is, of course, inspired by some work with Hammer to parse PDF files.)

Suspend and resume

If we want to suspend parsing to do something else for a while --- for example, read more input from somewhere else, or parse something else for a while --- nearly all the state we need is either in the context data structure saved for backtracking, in the arena, or on the stack. Doing some other thing that can be done by calling a subroutine that doesn't interact with the arena or unwind the stack is perfectly safe --- that's pretty much just an instance of "arbitrary computation" in the previous section --- but if you need to switch to a different arena or stack, all you need to resume the parse (restoring the stack) is the pointer to the backtracking context. And maybe the arena pointer, if that's not saved in the context.

Immediate-mode PEG parsing in other programming languages

As described above, this sounds like an approach that's pretty strongly tied to assembly language (although not any particular assembly language) because it relies on being able to manipulate stacks and control flow in a counterintuitive way. But it turns out to map in a reasonable way onto *some* other programming languages.

Doing it in C

You can do most of these things in C; `choice`/`nextchoice` require using `setjmp`/`longjmp`, and you'd probably want to use an integer return value from them rather than the carry flag. And of course you can't do alloca()-like things or keep your allocation pointer in a register, and C function calls are usually a lot more expensive, though, e.g., things like `__attribute__((fastcall))`, `__attribute__((regparm(3)))`, and the amd64 ABI can help. The rest is pretty much the same:

```
if (choice()) letter('h'), letter('i');
if (nextchoice()) letter('j');
endchoice();

void *rv = 0;
if (choice()) parse_keyword();
if (negatechoice()) return parse_identifier();
```

In C++ or Rust or D, you might be able to use RAII to automatically do the `endchoice()` calls for you (though I'm not sure Rust or D have the requisite `longjmp` equivalent, although presumably you can invoke `setjmp` and `longjmp`; and using them in C++ is pretty risky because of the profusion of invisible destructors you might be jumping over); in C the only way to do that is to abuse the

preprocessor. Which brings us to macro assembly.

Or macro assembly

A macro assembler with enough of a context-stack facility to implement if-then-else and do-while should enable you to write a PEG grammar in a fairly literal fashion, maybe something like this:

```
expr: either(go(term); either(eat('+'); or eat('-')); go(expr)
             or go(term)); ret
term: either(go(atom); either(eat('*'); or eat('/')); go(term)
             or go(atom)); ret
atom: either(number; or eat('('); go(expr); eat(')')); ret
number: some(range('0', '9')); ret
```

This would keep you from accidentally failing to balance your `choice`/`endchoice` pairs, leaving out a `jc`, or jumping to the wrong label.

Doing it in Scheme

Earlier I said that `longjmp` was in short supply in modern languages; but Scheme of course has `call-with-current-continuation`, which is a generalization of `setjmp` that could easily be used to implement the above. Scheme also has a solid macro system. So Scheme is in some sense the best-suited language to this, except that all Scheme implementations are slow.

Or Ruby

Yeah, Ruby also has call/cc, and it's designed for embedded DSLs like this, although it uses reflection and a lightweight closure syntax rather than a macro system.

Or Forth

Brad Rodriguez published an article in 1990 about "BNF parsing" in Forth in something like this way, in three or four screens of code, including language concatenation by execution sequencing, and I read it sometime in the 1990s but didn't understand it. However, a lot of the details are different. (He uses the "no-opping the intermediate operations" approach I rejected above, as well as the "accept an alternative by returning from the caller's caller" technique.) I think the implementation technique described here might work better (more efficiently, more flexibly) in Forth than Rodriguez's method does. Like C, Forth implementations typically have the equivalent of `setjmp` and `longjmp`, and I suspect they're less dangerous due to the rarity of stack-allocated variables in Forth; and, of course, Forth is ideally suited to embedded DSLs.

How about Python, JS, or Lua?

Python of course doesn't have call/cc, setjmp, macros, or a reasonable lambda syntax. What it does have is generators, which have been adopted by JS now as well, and which are pretty straightforward to use to lazily generate candidate parses of strings, handling backtracking by resuming a generator rather than trashing it. This is a pretty different paradigm and I'm not sure how to map the immediate-mode stuff above onto it; it almost seems easier to do general context-free parsing by backtracking that way, although by default that will take exponential time.

Lua has "coroutines" which are really full-fledged cooperative threads, which provide similar functionality to Python generators but with per-thread stacks, but it doesn't have call/cc or anything similar.

Topics

Dercuano stylesheet notes

Kragen Javier Sitaker, 2019-04-28 (updated 2019-05-09)
(72 minutes)

I wanted to spiff up the Dercuano display a little so that it would be pleasant to read, but more with the objective of being like a book than being like an advertising brochure. [When I started writing this, the only CSS thing I'd done was give it a max-width to keep the lines from being 2000 pixels long.]

But what does a book look like? What does a good web page look like? Are they different?

Medium

Marcin Wichary of Medium wrote an article entitled, "Crafting link underlines on Medium," in which he talks about how to get link underlines to work. He describes a CSS hack using background images "synthesized *via*" gradients, and also describes a further hack using a white CSS text shadow to make it clear descenders, which they apparently don't actually use — but Gwern does something like this, see below.

I looked at Medium to see how they're doing things. They set the article title and article text in a serif font (falling back to Georgia if their proprietary font fails to load) and article section titles in a sans-serif font (falling back, similarly, to Helvetica); everything uses rgba(0, 0, 0, 0.84) except for the line underneath the title in gray and the pullquotes. (0.84 is the fourth root of ½.) The article had a font-size of 21 pixels and a line-height of 1.4; the main article title had font-size of 46 pixels; and the subtitles were somewhere in between in size. The subtitles were somewhat bold and left-aligned.

The article section titles have some vertical whitespace around them, but I don't remember how much.

Article category links are at the end of the article in light gray boxes with rounded corners, generally all on the same line.

They're using a different serif font with Clarendon slab serifs for drop-caps, and to get hanging punctuation on paragraphs that open with open-quotes, they use a paragraph-opening-with-open-quotes class to set text-indent to -0.4em.

The body text is set in a single column with a max-width of 700 pixels, which works out to 33 ems. It's centered with `margin: foopx auto` at the appropriate level of nesting; inline images are also centered but in a wider column, so they can reach the margins on a small screen (Wichary pompously dubs this "our signature full-bleed, blurred images" in his post). Despite using a serif font like a real book, they use a ragged right margin, at least in Chrome — possibly a concession to the lack of hyphenation.

At times they use thin spaces around their em dashes, as I do.

Medium goes to some trouble to use hanging punctuation, like Gutenberg, but it only works sometimes.

Font files: ET Book!

I looked at downloading Noto to bundle, but Noto is humongous; the whole bundle is 1.1 GB, and even just a CJK ideograph bundle is 100+ megs. So I pretty much have to rely on system fonts for

Dercuano, since keeping the download bundle self-sufficient and small is pretty much the whole point. (Also, I can't find a Noto subset that includes the Latin letters this text is written in.)

Later, though, on Fernando Irarrázaval's blog, I found the ET Book Bembo-like font family, which has been released under the MIT license by Edward Tufte and its authors, Dmitry Krasny and Bonnie Scranton. The four fonts I'm including, three of which I'm using, gzip to 180K, even though they cover all of Latin-1, plus a bit more! And it's beautiful, especially the italic, and it has old-style figures. It even has ligature glyphs, although mysteriously the browser doesn't use them, apparently because the font isn't specifying the necessary context rules.

So I've switched to ET Book. But the 22px size I was using before (DejaVu Serif, the system default) is a lot bigger than the 22px size of ET Book. The equivalent ET Book size is nearly 28px. Now DejaVu Sans Mono looks dramatically oversized, its x-height being nearly as tall as the Antiqua capitals. So I twiddled several things to compensate:

- I used ET Book 26px as the default, reducing the line-height from 1.5 to 1.4 to compensate;
- I reduced <pre> and also <code> to font-size: 80%;
- I shortened max-width from 45em to 35em.

These are somewhat suboptimal if ET Book doesn't load for some reason (maybe your browser doesn't support .ttf webfonts, although Firefox and Chromium do, or if you're experiencing a Firefox bug, as I think I am at the moment) but not catastrophic. In particular the line length in ET Book is a little longer than I would like so that it's not uncomfortably short if ET Book fails to load.

Characters not covered by ET Book are typically problematic, e.g., the lowercase Greek in the title of Plato was not particularly democratic; ἄρχειν is not "participating in politics" (p. 1707), because they are taller, wider, bolder, and have less stress than ET Book, at least on my machine. I'm not sure what to do about that; shipping Cardo is not an option, as it would add a megabyte or more to the package size. I might be able to scale the letterforms and bearings in ET Book by 125% to fix it (and undo the stylesheet changes).

The biggest coverage problem is the numeric superscripts: $^{0}{}_{123}{}^{456789}$. ET Book covers ¹²³, while the others come from a fallback font, with fairly disastrous effects on the readability of things like "x^{62}", which looks more like "$x^{6}{}_{2}$" than the intended "x^{62}".

Mark Shoulson points out that in addition to lacking ligature tables, leading to the same missing ligatures I complain about in *The Grammar of Graphics*, the font also lacks kerning tables, leading to kerning that looks precisely like the kerning I complain of below in *Tick, Tock*.

Books

There's an interesting question as to whether type should be larger or smaller on a computer screen than in a book. On a computer screen, of course, you never run out of paper; the "paper" is free; so maybe you should use larger type. But often the screen is smaller than a two-page spread in a book, and if you have two books open at

once they have to share the same screen; so maybe you should use smaller type. But often the screen is of much worse resolution, so maybe you should use larger type. And maybe the screen is further away, so maybe you should use larger type.

But probably the differences from standard book practices for books should be justifiable. For example, books don't have to be readable on a cellphone screen, and when they can take advantage of color or even grayscale, it adds substantial extra cost, and it suffers a substantial loss of resolution due to halftoning. Furthermore, printing with movable type *on top of* a color or even grayscale background, as opposed to inserting figures, is basically impossible. So there might be reasons other than imitating sales brochures for using color.

In the other direction, common browsers (notoriously Chromium) still don't do hyphenation, and their paragraph-filling algorithms are minimialistic at best, and justification in browsers never adds letter-spacing, so enabling justification in CSS gives you wildly varying word spacing, which looks shitty. So if you're targeting web browsers you need to suck it up and accept ragged right margins. Or, if you're sufficiently eccentric, ragged left or ventilated prose.

The rigors of five and a half centuries of metal type printing, concurrent with the birth of capitalism, salutary as they have been for human attempts at civilization, have not been kind to typography. The aesthetic experience of reading has been compromised by first one and then another vulgar commercial innovation — lining numerals, sans-serif fonts, copyright, typewriter fonts, banner ads, paywalls, product placement — while the limitations of metal type have been embraced as virtues by generations of typographers. Rubrication is no longer red, but black, and color is indeed omitted almost entirely; swashes are rare flourishes; illustrations are separated from text by rigid rectangular borders and even omitted altogether; drop caps and other initials are abjured or relegated to extreme rarity, perhaps one per thousand pages; border ornamentation is minimal when it is present at all; marginalia are omitted; justification of lines can only be done by altering spacing rather than stretching glyphs; letters are present only in one to four sizes rather than a full variety; etc.

The major overall limitation, really, is that in movable-type printing, things cannot overlap. Swashes cannot overlap other letters or margins; words cannot protrude into margins; text blocks cannot overlap illustrations; descenders of one line cannot overlap ascenders of another, creating a pressure toward greater and greater x-heights, destroying much of the beauty of minuscule; diacritics cannot overlap multiple letters; and only through the hacks of kerning and ligatures could we even get reasonable spacing for most letter combinations involving capitals and "f".

To be sure, these last centuries have brought beneficial innovations as well. The ample use of whitespace to improve readability, beginning with division into paragraphs; boldface for scannability; the elimination of scribal abbreviations, and especially the abominable Tironian notes, which saved precious parchment at the price of promoting ignorance among the people; the improved calligraphic quality of commonplace fonts over all but the best scribal calligraphy; photographic printing; four-color halftoning; data plots, and especially the work done by Tufte to improve them; algorithmic

rendering; the modern language of mathematical formulas; hyperlinks; simulation; animation; interactivity; but most of all, mechanical reproducibility.

TEX: the Program

This book is arguably some kind of referent for readable typesetting, but maybe not the copy of it I'm looking at. On my monitor at a comfortable zoom, the line spacing in its paragraphs is about 32 pixels, the x-height is about 11 pixels, and the spacing of lowercase "m"s is about 25 pixels; and a line is about 1350 pixels wide, which is a bit wide for my taste (about 19 words per line!) but certainly helps with the justification. And the paragraphs in the book are mostly very short, so long lines don't impede readability as much as they might.

If we take the font size (1 em) to be nominally 24 pixels, that works out to a CSS line-height of 1.33 (ems), a width of 56.25 ems, an "m" of 1.04 ems, and an x-height of 0.46 ems, which is pretty readable.

At this zoom, the page is 285 mm wide, which is 1.36× the 210-mm width of A4 paper and 1.32× the 216-mm width of US letter-size paper it was most probably formatted for, and those 1350-pixel lines are 217 mm wide, which is 161 μm per pixel (158 dpi). We can conclude that the book was set in type of around 24 pix · 1.32 · 161 μm/pix = 5.1 mm = 14.5 PostScript points.

Each section (of which most consist of a single paragraph) begins with a boldfaced paragraph number for hyperlinking, and some of them also include a boldfaced topic for that "part", which is typically several pages' worth, and begins at the top of a new page. This inline "part header" is the same size as the rest of the text. Between sections, there is a blank space (a vertical skip) of about a line in height, and there is a smaller vertical skip between text and code. Code is annotated below with a backlink in smaller type: "This code is used in section 726," and sometimes also, "See also sections 20, 26, …". The smaller type is reduced in size by about a factor of 1.4.

There is no further rubrication other than the index and table of contents at the end of the book.

TickTock

This is a Dean R. Koontz thriller, or possibly a comedic satire of thrillers. The copy I have here is a Ballantine paperback, justified with narrow margins (9 mm) in a traditional curved-serif font, with paragraph indents and no extra vertical space between paragraphs. The apostrophes are poorly kerned with too much space to their right; combinations like "ov", "Vi", and "To" are also kerned, or rather not kerned, with too much space — perhaps if Koontz had known his book would be printed like this, he wouldn't have named his main character "Tommy". There are 32 lines per page; the last 31 of them measure 147 mm in total height, giving 4.7 mm or 13.4 PostScript points per line; the spacing between letters "m" is about 4.0 mm or about 11 points. This suggests that the book is set in 12-point type with about an extra 0.12 ems of leading between the lines.

This book was printed in 1996, so it doesn't have any vertical typewriter quotes; all of its quotes and apostrophes are of the proper 9, double 6, and double 9 types.

Each page holds about 290 words, which works out to 9.1 words per line; within paragraph bodies (excluding indents and last lines) the average is closer to 10. The lines measure 85 mm wide, which would be 20 ems if my 12-point-type guess above is correct. Typically there are zero to two hyphenations per page, so at times the word spacing is a bit uneven, but the average word length is pretty short, so it's not as bad as you'd think; to illustrate the word length, here's a randomly chosen paragraph on p. 210:

"But I *don't* believe they come all the way across the galaxy to kidnap people and take them up in flying saucers and examine their genitals."

There are abundant italics, but outside of titles, there is no boldface.

The main book title on the title page (in bold capitals) has a line-height of about 15 mm (43 points), and the author's name is set below it (in bold capitals) with a line-height of about 11 mm (31 points). Each chapter title ("ONE", "TWO", "A NOTE TO THE READER", etc.) is set (in bold capitals) with a line-height of about 7 mm, underlined with the same small illustration. Chapter titles start on new pages, but many start on even pages, leaving no pages blank. Other than page headers, there is no other rubrication or division in the book, such as horizontal rules or lines of asterisks.

The book feels a bit crowded with its 2-em margins, short lines, narrow leading, and no vertical skips. But I suppose lulling you into too much tranquility would defeat the purpose of the book, as well as cutting into Ballantine's profits with extra paper costs.

The Grammar of Graphics

This is a recent (2005) semi-academic book published in hardcover by Springer, with, unusually, bright white glossy paper and color — apparently finely halftoned CMYK, on every page. That's because it's a treatise on the visual display of quantitative information, and the authors have dedicated a great deal of care to its visual appearance; they set it in Times Roman and Times Italic with FrameMaker. There is a strong hierarchical structure to the book, with numbered section headings that frequently run four levels deep ("*6.2.5.3 Fisher's z Scale*"), which are all in Times Italic. There are occasional subheadings before level-4 headings, which are not numbered. Paragraphs are indented (except when following a heading) and justified, but with no extra vertical skip between them.

There are about 46 lines per page, although there is extra leading around headings, figures, equations, and blockquotes, and the text never runs for an entire page without being interrupted with headings, figures, equations, or blockquotes. Lines are 115 mm long, typically containing about 12–14 words, and the lines within a single paragraph are rather tight; 10 lines are 40 mm high, giving 4.0 mm or 11.3 points as the line height, and the escapement of an "m" also measures 4.0 mm. (XXX apparently it's more like 3.0 mm?) Hyphenation is quite frequent, so whitespace is very consistent despite justification. Margins are a generous 20mm to the sides and 22mm at the bottom of the page; the top margin below the page heading is smaller, more like 9mm, but there's another 15mm above the page heading.

This suggests that the text is set in 11-point Times Roman with no

leading and 30-em lines.

Level-1 headings ("14. Time") are chapter headings. The chapter number is set in bold italic with a line height of about 16 mm (44 pt), while the chapter name is set below it below an 8-mm skip in medium italic at about 8 mm (22 pt). Both of these parts of the heading are right-aligned. The page heading above the chapter title is omitted, and blank pages are used to ensure that chapters always begin on odd pages. Below the chapter heading, there is a skip of some 24 mm above the body text.

Level-2 headings ("14.1 Mathematics of Time") are set in italic, with the heading number in bold, and are set somewhat larger than the body text, perhaps twice the point size (8 mm, 22 pt), though it seems to be about 25% smaller than the chapter title somehow. There is only a small extra skip below them, but about a one-line skip above them. There is an extra large space, about a quad, between the number and the title; they also do this after bullets in bulleted lists, and Knuth also did it after section numbers.

Level-3 headings ("14.1.1 Deterministic Models of Time") are set in the same bold/bold-italic combination of styles, but in a slightly smaller font-size, again by about 25%, but still about a 50% larger point size than the body text. These *do* have extra leading below them, though less than above.

Level-4 headings ("14.1.1 Orbits and Vibrations") are set entirely in bold italic, and the body text is smaller than they are by about 25%. Perhaps the bold is intended to add contrast with the body text to compensate for the smaller difference in size, or perhaps bold was not used in the larger titles simply to avoid overwhelming the reader. Again, they have extra leading above and below.

Level-5 subheadings (e.g., "Updates" on p. 438, within §14.3.3.1 "Data") are the same size and with apparently the same leading as level-4 headings, but without numbers.

FrameMaker's typesetting of the equations is very much inferior to T_EX's, with poor kerning, parentheses that fail to enclose and whose line thickness never varies (though that of the letters and digits does), and so on. Aside from that, though, its layout is much the same.

The leading above headings, of course, goes away when they are at the top of a page.

They use a monoline sans-serif font for the names of their data series.

Figures are captioned below in italic; text never flows around figures.

I think a good approximation of the layout can be expressed as follows:

• Body text is 11-point Times Roman with 30-em justified paragraphs. Paragraphs that don't follow headings are indented by two ems. Body text has no extra leading, even between paragraphs.
• The text sizes of headings of levels 4, 3, 2, and 1 are a geometric progression ending at 22 points; this would make them 13.1-point, 15.6-point, 18.5-point, and 22-point, or, expressed in ems of the body text, 1.19 ems, 1.41 ems, 1.68 ems, and 2.0 ems.
• Except that chapter numbers are on a separate line in 44-point text.

• Chapter headings are right-aligned with page breaks to an odd page

before them. All other headings are left-aligned with leading above them equal to their line-height and leading below equal to the line-height of the body text.

Occasionally two headings (of different levels) occur in succession.

Although the fonts used are pretty nice, and the layout and design are reasonable (aside from the truly lamentable formulas), the text is mysteriously entirely missing ligatures; "fi" and "ffi" do not appear as joined "fi" and "ffi" as they should.

How to Make a Telescope

This is a 1957 English translation, published in the US by Interscience Publishers, of Jean Texereau's 1951 French text *La construction du télescope d'amateur*. This copy was apparently acquired by the Northtown–Shiloh Library in Dayton in 1958 (and rebound by The Kalmbacher Bookbinding Co., in Toledo), so it was probably printed in 1957. Like *Grammar of Graphics*, it's printed entirely on glossy paper, although it's yellowed a bit in the ensuing 60 years; the purpose is again fine halftoning, but this time only in black and white.

Its body text is set in a serif font in justified paragraphs with about twice as much space after periods and colons as between words. The paragraphs occasionally have figures centered in the middle of them; they never flow around figures, and they are all indented, by about three or four ems. More surprisingly, section headings within chapters ("II-18. The Polishing Operation" — bold, in the same size as the body text, with leading below and even more extra leading above) are also indented by the same amount as the paragraphs. 10 lines occupy 42 mm, so the line spacing is 11.9 points — might as well call it 12 points.

The chapter titles ("II. Making the Main Mirror") are preceded by page breaks to put them at the top of an odd-numbered page, and with a page-width vertical rule underneath the (right-justified) Roman numeral, below which is a vertical skip of some 17 mm, and right-justified, given extra letter-spacing, in a sans-serif font, in all capitals, in a somewhat larger text size (perhaps 16 points rather than 12), is the title. Beneath the title is another 20mm of leading and then the section heading of the first section of the chapter.

The Roman numerals at the openings of chapters are still larger than the chapter titles themselves; perhaps they reach 24 points.

Some of the numbered sections are contain unnumbered subsections, indicated in the table of contents; the subsection titles occur at the beginning of a paragraph in italic.

References are given in footnotes in a somewhat smaller font, perhaps 9 pt.

Data tables are captioned above in all capitals (in the same font as the body text) and divided into sections with page-width horizontal rules. Columns are separated by whitespace with no rules. Figures are captioned below in the same smaller font used for footnotes.

The Palmer Method of Business Writing

This book was published in 1915; the copy I'm looking at was digitized from the collection of the University of California, and I downloaded the DjVu from the Internet Archive. A Microsoft

watermark contaminates every page. It's a penmanship textbook. Both its layout and its textual tone are considerably more bombastic than the other specimens I looked at above.

I don't have a reliable way to determine its absolute size, since I have only a DjVu scan.

Sans-serif fonts make no appearance whatsoever, except possibly in the legend on the original book cover, "THE A. N. PALMER COMPANY, New York, …", which I can't be sure of because DjVu has corrupted it too badly. The book is full of black-and-white photos (presumably halftoned) and illustrations (which are not); at times the text flows around them, but usually they are placed above and below the text.

The pages are, unusually, in landscape orientation, and seem to have been bound on a short edge; the ratio is about 5 parts length to 3 parts height. Although the typography is somewhat inconsistent, often varying in the body text size from page to page, it is mostly set in two columns, with extra spacing separating sentences (plus spaces inside quote marks), but headings and most of the abundant illustrations span both columns. Occasionally body text also spans both columns, typically for very short sections. Boldface is used liberally, for all the headings and occasionally for emphasis in the body text, while italic is entirely absent.

The body text is set fairly tight with no apparent leading. The columns are about 35 ems wide, and separated by about two ems. The paragraphs, even the first, are indented by about three ems, and are of course justified, with only very occasional hyphenation, resulting in substantial variation in word spacing. Occasionally there is a vertical skip of a line or so between paragraphs, but not usually. Evidently no concern has been given to preventing widows and orphans, as they occur frequently, while unequal column lengths never occur at all.

The headings found throughout the book ("LESSON 80", "Drill 95") are centered and boldfaced. Typically each section has a title ("LESSON 76") which may be broken onto two lines ("LESSON 70", "Drill 85") or written on a single line with an em dash, and may sections also have a subtitle ("TO RELIEVE MUSCULAR TENSION"); the subtitle is about 1.5 times the point size of the body text, and the title is about 1.5 times larger than that. These headings have equal and miserly leading above and below.

Semantically the "Drills" are subordinate to the "lessons" in the sense that some lessons contain more than one drill, and may contain other sections, whether or not headed by what I've called a "subtitle", that aren't part of a "drill".

Many pages have a centered boldface injunction at the bottom in the same point size as the body text, saying things like, "It is not Palmer Method if the lines are tremulous. Study the instructions for speed requirements." On p. 19 instead this same injunction is repeated in large boldface type at the top of the page, with the period removed so it will fit. Truly, the typographic crimes of this manual are manifold.

Some paragraphs, though indented as usual, begin with an inline heading in boldface followed by an em dash or colon:

Height—Reference has already been made to one-sixteenth of an

To the teacher: If you have studied the lessons in advance, have

As an interesting side note, where arrows are present in the

diagrams, they are fletched.

Standard Handbook for Electrical Engineers

This is the 1922 Fifth Edition, scanned from a copy at the University of Toronto library. It's printed by The Maple Press of York, PA, but copyright by the McGraw–Hill Book Company.

It does use italics, as well as small caps, but still has no sans-serif fonts.

The book is divided into "sections" rather than "chapters". Section titles are centered, bold, in all caps, with extra letter-spacing and a largish vertical skip above them, and a short centered horizontal rule below them above the subtitles, if present, and body text. The title is about 1.5 times the point size of the body text. Paragraphs are indented (by about one em), even the first, and justified with generous hyphenation, and line spacing is somewhat loose. Margins are very narrow (on p. 377 the operator scanned their finger, and much of the material near the spine is lost to the Scribe's glass join). There is extra spacing between sentences, and quotes seem to be concatenated apostrophes, perhaps with extra whitespace inside of them. There are places where letter-spacing is compromised for the sake of justification, as on p. 62.

(Because p. 19 has a ruler printed on it, I can report an absolute text size for this book: at "300%" zoom, 30mm of the original book is 58mm on my screen, so the actual zoom is 193%. 9 lines of body text being 37 mm on my screen means that they were 19.1 mm in the original book, so the body text was printed at the minuscule size of 6 PostScript points!)

Section subtitles are centered, bold, in all caps, at a size intermediate between that of the body text and the section title, so about 1.2 times the point size of the body text. At times there is an additional sub-subtitle in centered italics at the normal body text style. There is some inconsistency. We have:

SECTION 2
ELECTRIC AND MAGNETIC CIRCUITS
BY VLADIMIR KARAPETOFF,
Professor of Electrical Engineering, Cornell University, Fellow, American Institute of Electrical Engineers
CONTENTS

Here "SECTION 2" is what I've been calling a "section title", "ELECTRIC AND MAGNETIC CIRCUITS" is formatted at the same point size but bolder and without the extra letter-spacing; the author's byline is formatted at the intermediate font size I called a "subtitle" before; his credentials are centered, wrapped, in italics; and "CONTENTS" is formatted as a subsection heading as described later. But then, on the next page, we have:

SECTION 2
ELECTRIC AND MAGNETIC CIRCUITS
ELECTRIC POTENTIAL

And here we have returned to the formatting I described above: section title, rule, subtitle, subsection heading.

Sections are mostly divided into subsections, which have their own (centered, bold, all-caps) headings, usually at the same point size as the body text, with a little extra leading above and below — especially above, in cases where the page is a bit loose. In some cases they use a larger font, as on p. 576, and may have a subsection subtitle (centered, all-caps, bold, at the size of the body text), as on p. 897, where it gives

the author's name.

Within the body of the book, paragraphs are numbered and begin with a boldface title, which sometimes forms part of a sentence, e.g., "**5. The English weights and measures** are based upon old Roman weights and measures." Paragraph numbering does not restart after subsection headings. At times the numbering becomes hierarchical, perhaps to maintain numbering compatibility across editions, with numbers such as **2.**, **3(a$_1$)**, **3(b$_2$)**, and **3(B)**, which inexplicably comes between paragraphs **3(b$_2$)** and **3(c$_1$)**. Hyperlinks between paragraphs also use bold for the paragraph number: "(Par.**102**)." In the introductory part, a larger text size is used, making the paragraphs about 25 ems wide, but in the body of the book, the paragraphs are about 32 ems wide due to their smaller font size.

Occasionally an numbered "paragraph" is divided into multiple typographical paragraphs, each indented and commonly with a boldface title.

Text flows around figures, which are captioned below and numbered using small caps for the abbreviation "Fig.". The caption is typically formatted as a wrapped centered paragraph, but sometimes justified.

Footnotes are used frequently to cite sources, separated from the body text on the page with a page-width horizontal rule; they are not numbered but use the symbols *, †, ‡, §, ‖, ¶, and then **, ††, etc. In other cases sources are simply cited in the body text, or in a bibliography at the end of a section. (In one case Simpson's Rule is given in a footnote stretched across two pages.)

Equations are centered, inserted into the middle of sentences with no extra leading, and some of them are numbered on the right. The line before the equation is not justified, as if it ended a paragraph (difficult to avoid in cases where it consists simply of, say, "or", as on p. 59), but the line after it is not indented, and typically continues the sentence containing the equation. In some cases, the line with the equation ends with a comma or semicolon that forms part of the sentence.

Amusingly, integrals are sometimes written using "∂" for the differential, sometimes written with an italic *d*, and sometimes omit the differential entirely.

Boldface is used within paragraphs for emphasis, perhaps to aid skimming:

11. **The meter** was selected as a length equal to the **ten millionth part** of the **northern quadrant of the earth**, or distance from pole to equator…

Page headers have the section title, centered in italic caps, and on the side away from the fold, give the section and paragraph numbers in bold, e.g., "Sec. 1–5 *UNITS, FACTORS, AND TABLES*" or "*UNITS, FACTORS, AND TABLES* Sec. 1–11".

Extra leading is often but not always used between paragraphs, perhaps to stretch the page to the right height; in these cases extra leading is also used above subsection headings, as mentioned above.

Ordered lists are usually indicated with parenthesized italic letters (*a*) (*b*) (*c*) and occasionally with numbers and periods 1. 2. 3. or numbers with parentheses (1) (2) (3). Occasionally they are split into a paragraph (not part of the normal paragraph numbering) per item, and those paragraphs are indented as usual. Occasionally, as on p. 1505, the list items are given a hanging indent of another couple of

ems (so the first line is indented by one em and subsequent lines by three), and even nested lists are found, with the nested list items indented to an intermediate level. No extra leading is used in any case. Bulleted lists do not occur.

Tables generally use both horizontal and vertical rules to separate rows and columns, broken at intersections, and are usually printed in landscape mode. They are numbered and captioned in bold above. Table column and row headings are centered but otherwise use the same typography as table contents. Table section headings (spanning all columns) are additionally boldfaced. In some cases, instead of horizontal rules, they use dot leaders, or are split into five-row groupings using whitespace. Table borders are heavier rules. Numerical table columns are usually decimal-point-aligned, using lining ("capital") figures. Sometimes table contents are flowed across multiple "supercolumns" separated by double vertical rules.

The mini-table of contents at the start of section 2 is split into two columns and uses hanging indents to continue subsection names across lines.

On p. 881 we have a tree structure depicted with braces "{" for branching and blocks of text for the nodes.

Unidirectional arrows in diagrams are usually fletched, but not always. Dimension arrows are not fletched and use the long-short-long dash pattern which to me indicates a center line. Open-triangle arrowheads (with a line through the middle), single-barbed arrowheads, and open-line arrowheads both appear, but solid-triangle arrowheads do not.

Plots of quantitative data are uniformly grid-ruled, with the rules often broken to preserve readability of legend text that runs along a plotted line. In places, as on p. 584, an area of the grid is removed to make space for a plot caption, but plots are usually just captioned externally as figures. Qualitative plots usually have only the axes ruled.

The Book of Useful Knowledge: A Cyclopædia of Six Thousand Practical Receipts

This book, by an Arnold James Cooley ("Practical Chemist"), was published in 1850 in New York, Philadelphia, and Cincinnati, a reprint of the original London edition (and thus presumably done without the consent of the author). The full title and subtitle is "The Book of Useful Knowledge: A Cyclopædia of Six Thousand Practical Receipts, and Collateral Information in the Arts, Manufactures, and Trades, including Medicine, Pharmacy, and Domestic Economy. Designed as a Compendious Book of Reference for the Manufacturer, Tradesman, Amateur, and Heads of Families." This is centered on the title page in somewhere between eight and twelve different fonts, one of them a blackletter, but of course no sans-serif fonts.

The scan is, unfortunately, by Google, so the scan quality is very poor, and I have no idea of the original physical size.

The main body of the text is a series of alphabetized entries, each beginning with an all-caps headword, set in two columns of 68 lines with a vertical rule between them; most paragraphs within each entry begin with an italicized header. Paragraphs, even the initial paragraph of each entry, are all indented by about one em. No leading seems to

be used. The columns are only about 20 ems wide, and of course are justified with heavy hyphenation.

Occasionally an entry is subdivided into sequences of paragraphs with a centered italic header with a blank line above it.

The end of the body text is followed by the notation, centered, in boldface, in all capitals, with extra letter-spacing, near the bottom of the page: "THE END."

Frequent use of italics for emphasis gives an impression of great excitability:

> then spread it *thinly* on a dish, and expose it before the fire, or to a current of dry air, until nearly dry. It will then keep for years in wide-mouthed bottles or pots, covered over with bladder. For *use*, a little is dissolved in water.

There is extra spacing betwee sentences, and it also uses thin spaces before the colons and abundant semicolons and following open double- and single-quotes, as in "do the ‘ important’ at".

A typical entry begins:

> ABRASION. A superficial injury of the skin, resulting from the partial removal of the cuticle by friction.
> *Treat.* When the injured surface is small, and unexposed, no application is generally required,…

Trailing punctuation goes inside close parentheses:

> not only suffers refraction at the spherical surface, (called spherical aberration,) but the different colored rays, forming the beam of light,

Each page has a horizontal rule above its body text, above which are the guidewords for that page (or their first three letters, at any rate) and its page number.

Page margins are fairly generous, but column margins are quite cramped — perhaps a single em of column separation, with that vertical rule running down the middle of it.

Tables have horizontal rules delimiting them and separating their header, but not separating their rows; by contrast, they separate their columns with vertical rules, but usually have no vertical borders. Often table row boundaries are indicated only by braces "}", and intermittent dot leaders facilitate column-to-column traversal. They are neither numbered, but sometimes captioned above. As in the *Standard Handbook*, tables are sometimes flowed into "supercolumns" separated by double vertical rules. Lining figures are used. Decimal points are vertically centered: "its sp. gr. about 1·069 or 1·070;", and numeric columns are aligned by them.

Some simpler tables (e.g., the alkaloids on p. 47) are simply two columns of text joined by dot leaders with italic headers at the top.

In a few cases, the columns end to make room for a large table, like the table captioned "I. *French Decimal Measures of Length*", on p. 419.

Text flows around small figures; large figures interrupt paragraphs. As with tables, figures are neither numbered nor captioned, but sometimes they include a legend set in smaller type below them.

Footnotes are set in smaller type below a horizontal rule at the bottom of a column; they are linked with * and †.

Hyperlinks to other Cyclopædia entries are at times merely indicated with notations such as ", (which see.)" or even "; to each of which the reader is referred, in their alphabetical places.". Small caps are used frequently for synonyms and surnames and I think they may

also indicate hyperlinks, as they traditionally do in dictionaries: "(See BREWING.)" Sometimes they seem to merely indicate emphasis, as in the bipartite name given for the almond tree.

Ordered lists are sequences of paragraphs, indented as usual by about an em, but beginning with the ordinals "1st.", "2d.", "3d.", "4th.", etc., or the Hindu-Arabic numerals "1.", "2.", "3.", etc., or italicized letters "*a*.", "*b*.", etc., or the Roman numerals "I.", "II.", "III.", etc. Commonly these varying forms of numeration indicate nested list structures, but no typographical cues as to the hierarchy are visible.

Bulleted lists do not occur, though simple tables fulfill their function. On p. 108, for example, we have this table of plants for attracting bees:

Shrubs, &c.
Rosemary,
Broom,
Heath,
Furze,
Fruit-blossoms. *Flowers.*
Mignonette,
Lemon thyme,
Borage,
White clover,
Bean-flowers.

The book uses a number of ideograms that are no longer in use, although perhaps surprisingly, Unicode has them. Its rendering of "%" uses a horizontal bar, and many recipes are given in apothecaries' weight, using lower-case j-terminated Roman numerals, in scruples (written as "℈", as in "℈iv"), drachms (written "ʒ"; now often called "drams", though using that spelling now implies you're talking about avoirdupois drams), and ounces (written "℥"). This is explained on p. 561. Thus "ʒij" means "two [apothecaries'] drams", which is to say, a quarter of an apothecaries' ounce (which was the same as the troy ounce) or 7.78 g in modern units. The Roman numerals here also represented ½ with "ss".

Greek words appear commonly in the etymologies of terms, but the Greek font used is to my eye quite poor; it has serifs, but its capital height is just a little taller than the x-height of the Roman font used, and its own x-height is smaller still. Typically it is oblique. The forms also seem somewhat crude and irregular to me, but that may be merely an artifact of the very poor scan quality common to Google Books scans and the small font size.

The Mechanism of Weaving

This book was unfortunately scanned by Google, so the scan is of extremely poor quality.

Published in London in 1894 and in New York by MacMillan. Everything is Roman or italic except the word "London" on the title page, which is blackletter and includes many fonts, and the lettering in the illustrations. Headings are all-caps, bold, centered, and at times with extra letter-spacing. The table of contents uses indentation to show structure, with sparse dot leaders, as well as small caps for top-level headings; and the index, set in two columns separated by a vertical rule, also uses indentation to show structure. The index has a

blank line before each new letter.

Line spacing is somewhat loose. There are 33 lines per page. Paragraphs are justified and indented by about two ems, except for (at times) the first paragraph after a heading, whose first word is instead rendered with small caps. In one unusual case on p. 250, the section heading begins a sentence completed in the body text; the same thing happens with a subsection heading on p. 311. Paragraphs are not separated by extra leading. There is extra spacing between sentences, and thin spaces before semicolons and colons and inside double quotes. Hyphenation is sparse, and word spacing is consequently somewhat uneven.

The text is divided into parts, which are divided into sections, which contain subsections. Any of these may begin in the middle of a page. Subsection headers are in small caps of the size of the body text; section headers are slightly larger (say 1.2× the point size), in caps; part headers ("PART I") are slightly larger still (say 1.5× the point size of the body text), also in caps, with a bit of extra letter-spacing. All three types of headings are centered and given extra leading, about a line's worth; part headers have even more extra leading above them, maybe two lines' worth. Subsection headers may have a little extra leading above them, but perhaps only when preceded by a paragraph; it varies from page to page.

Widows occur frequently, and on p. 56, there's a totally avoidable orphan due to placing a figure between the first line of text and the other four lines on the page.

Illustrations are numbered below in small caps ("FIG. 1"). Unfortunately they are rarely on the same page as the text about them, even though text sometimes flows around them. Italics are used in the text for letters referencing figures, and mysteriously at the end of the preface for the word "September", but barely used at all for emphasis, though occasionally for Latin phrases like *vice versa*.

Illustrations contain hand-inked letters and digits which do not have serifs, though they are rarely monoline, so you could reasonably argue that this book *does* include sans-serif text. But the first occurrence of a sans-serif letter in text as such is an "X" in the middle of p. 138 ("fulcrumed at X"), and it happens again on p. 391, though, as I said before, typically italic is used for this. (Of course, the multiplication sign × is always monoline and never has serifs.) You could argue that sans-serif is playing a role here similar to the role played by typewriter text in modern texts about programming.

(This book was printed in 1894. Sometime between 1890 and 1898, the Berthold Type Foundry released Akzidenz-Grotesk, one of the "German sans-serifs" that started appearing toward the end of the 19th century, and the first one to become popular. But it was intended for use in advertising rather than running text. Wikipedia tells me that "Egyptian"-style letters — meaning primitive monoline sans-serif letterforms modeled after surviving inscriptions from classical Egypt and Italy — became popular for advertising signs in England in the late 1700s. Caslon released his "Two Lines English Egyptian" font in 1816, but it is only known from specimen books.)

Tables have ruled borders, ruled column separators, and a rule between the headers and the table body, with rules interrupted where they would cross, but no ruled row separators. They are not numbered or captioned. Lining figures are used in tables and text,

though page numbers use old-style figures. Decimal points are vertically centered ("1·125") but vulgar fractions ("1⅛") are used more often; inversely, periods are used at times in formulas to indicate multiplication, though "×" is used more often.

Tables with no rules occur occasionally as well, in the form of a non-indented paragraph set in smaller type with extra leading above and below, consisting of many nearly identical lines with the identical parts replaced with ditto marks (double low 9 quotation marks, here.) An example is on p. 121. In a more unusual case on p. 264, the table has a horizontal rule and a total placed beneath it, and the paragraph continues from there.

In one case, a numbered list ("1.", "2.", "3.") is indented by the same amount as a paragraph, but then a hanging indent is used so that later lines in the list are indented more deeply. They have short vertical skips above and below them, but as with paragraphs, none within. In another case (p. 275–8) the list items are paragraphs leading with a parenthesized number ("(1) Power consumed.") which function as headings for the following paragraphs, but are not otherwise typographically distinguished.

Formulas, when interrupting a paragraph, are set in slightly smaller type, but are not numbered. Sometimes tall formulas in running text are accommodated with more leading.

Inches are indicated by oblique double prime signs, which differ from the "" signs used for quotes; similarly, the prime signs used to indicate parts of figures are distinct from apostrophes. Em dashes are used (often immediately after colons) and almost never have spaces around them (except once on p. 326), but en dashes are not used; on p. 119 and p. 167 a hyphen is instead used to indicate a range of years. (Conversely, on p. 247 there are spaces around two otherwise ordinary hyphens, and again on p. 249 and p. 260.) The ideogram "∴" is used to mean "therefore", and "a:b::c:d" is used to mean "a÷b = c÷d". Arrows are extensively fletched, but otherwise the visual language of the illustrations is quite modern — dashed lines indicate hidden lines; gears are indicated by a circle at their pitch radius plus dash-dot lines at their root and outermost radii; hatched potato-chip cutaways are used to suggest round shafts; in one place a line alternating short and long dashes indicates the line above a center of rotation; hatches up to a line suggest a solid surface.

First Folio *The Tragedie of Othello*

Looking at this scan from Shakespeare's First folio from Wikipedia , which is set in a Humanist roman and italic, I see vertical and horizontal rules around the whole page. The play's title is split across two page-width lines: "THE TRAGEDIE OF", in large roman type with large but somewhat uneven letter-spacing, and below "Othello, the Moore of Venice", in smaller type. The body text is set in two columns with an additional rule between them, mostly with a ragged right margin due to the iambic pentameter of the text, but justified where the text runs to longer lines. There is a act/scene title running across both columns, "Actus Primus. Scœna Prima." between horizontal rules, and on the facing page, "Scene Secunda" is in the same italic font, with column-wide horizontal rules above and below. Italics are used to distinguish characters' names and, strangely enough, nationalities. Spaces are somewhat uneven, but generally there is

extra space after sentence-ending periods, and spaces appear before colons, semicolons, and question marks, though sometimes they are omitted entirely around colons and commas. Occasionally spaces occur inside parentheses and occasionally not outside. There is no boldface and of course no sans serif.

Hyphenation is occasionally used. The spelling and lettering is as you'd expect for the 17th century: i/j, u/v, ſ/s are alternate forms of the same letters governed by position, and ligatures include ff, ſt, ct, ſh, ffi. (I don't actually see any "j"s, but I assume they would occur if a word ended in an "i".) On one occasion, perhaps to save a bit of space, "my" is spelled "ỹ". Most nouns and occasionally adjectives are capitalized.

"THE TRAGEDIE OF" is about four times the point size of the body text, and "Othello, the Moore of Venice" about twice the point size of the body text. "Actus Primus.", etc., is about 1.4× the point size of the body text. The first paragraph is begins with an ornamental capital "N" from Rodorigo's opening line, "NEuer tell me,I take it much vnkindly", which is about three and a half lines tall.

These titles have a bit of extra leading above and below: the play title has about half a line above, half a line below its second line, and almost a whole line below its lower line. At times stage directions are centered in a column, in italics, with a line of leading above and below; other times they are simply italics on a normally spaced line. No extra leading is used between lines of body text; each time a new character speaks, it begins a new indented paragraph, again set tight with no extra leading. There are 66 lines per column (not counting catchwords) at the normal line height.

Wikipedia tells me that the First Folio has a page height of 320 mm. About an eighth of that (≈40mm) is a margin at the bottom of the page, and about another sixteenth (≈20mm) at the top, so only about 260 mm holds those 66 lines, so the line height is about 3.9 mm, or 11 PostScript points. Just as in *Tick, Tock*, about 9–12 words fit into a column, when the lines aren't being broken to fit into iambic pentameter.

Wikipedia

Wikipedia regrettably uses almost exclusively sans-serif type. Figures are very frequent, and almost always floated right, with captions below in the regular body text style. My tables of contents, like Gwern's, are directly modeled on MediaWiki's. (I don't remember if UseModWiki had them.)

Wikipedia has `border-bottom: 1px solid #a2a9b1` to underline its `<h1>`s and `<h2>`s. (The rule applies to other headings, but for them it's overridden with a `border-bottom: 0`.) This is not entirely unobtrusive, but it's less obtrusive than horizontal rules in black ink typically are in books.

They also have navigational links in the left, top, and bottom margins of the article; these margins are a somewhat darker background color. Category links are at the bottom of the article. Since these margins take up a lot of screen real estate, Wikipedia uses a totally different style on hand computers.

Wikipedia tables have a variety of formats, but commonly (class="wikitable") they are captioned above in bold (and not

numbered), have uniform-width rules around them, separate both rows and columns with rules, and use a darker background (#f8f9fa) than the surrounding text's #fff — darker still for the headers (#eaecf0), which are bold and centered; they have substantial padding (`padding: 0.2em 0.4em`) in the table cells.

Gwern

Gwern's website is set in a serif font (body text and headers too) with justified paragraphs, indented where they follow other paragraphs (`p+p { text-indent: 2.5em }`); headings with some extra leading; drop caps (in some cases, like magnesium, quite elaborate ones); no leading between paragraphs; and browser-provided hyphenation. Some of the headings use small caps (with `font-feature-settings: 'smcp'` rather than `font-variant: small-caps`), though this varies by epoch; others use caps. He mostly uses old-style numerals (`font-variant-numeric: oldstyle-nums`). These would be unremarkable choices for a book, but on a web page they seem radical. There is more leading above headings than below. He's also using quite a number of less book-like choices: right-aligned second-level headings (underlined in black) (though several of the books I've looked at have some right-aligned headings), darker backgrounds with borders for blockquotes, underlined links, bulleted lists with four-pointed star bullets, varying gray levels in text, letter-sized logotype icons inserted into text, looser line spacing (line-height: 1.5), relatively long line length (50 ems or so).

His `<h3>`s are a bit outdented (as well as bold and all-caps), and his `<h4>`s are also outdented, which could in theory make them stand out when scanning visually.

Generally there is an abstract of a few hundred words beneath the metadata, next to the table of contents (see below), in a double-bordered gray box, before the article proper begins.

When he has nested blockquotes, the inner one has an even darker background than the outer one, which is achieved with a `blockquote blockquote` rule rather than alpha-blending. (It would appear that his blockquotes actually alternate in color.)

He's also using the hack mentioned earlier for underlining links: `text-decoration: none` to disable the normal underline, then a background image synthesized from gradients, with 12 overlaid white text-shadows to make holes in the underline around descenders. (This hack is unnecessary in recent versions of Chromium, which perforate the default link underline for descenders.)

Unlike Wikipedia and Medium, he puts his category links at the top of the article, after a brief summary paragraph in italics (a summary rather like the extended titles in Victorian-era books, more than 100-word abstracts of modern research papers).

Often Gwern captions his figures below in bold, with syntax-highlighted source code above (generally R with ggplot, sometimes Haskell, sometimes CSV); the source code plays the role of equations, but it is not centered; instead it is formatted as is conventional for source code, except on a light grey (#fafafa) background with a box around it (and limited in height, with scrollbars for overflow). On the occasions when he does use equations, they are typically done with MathJax (including its fonts: MJXc-TeX-math-Iw sure gives nice italics, and it seems to be 19K),

and when they are not inline, they are centered with 1.25 ems of leading (`margin: 1.25em auto`), which I think may be MathJax's default. (See Dercuano formula display (p. 495) for my thoughts on what to do for this.)

In the rare case where Gwern has tables, they are very fancy indeed, with an italic centered caption at the top with heavy rules above and below (matching a heavy rule at the bottom of the table) with half a line or so of leading, bold column headers with sorting buttons in them (switching from black-on-white to text-shadowed white-on-blue when activated), alternating light-gray and white row backgrounds with a smaller font size, and about half a line of leading above and below data rows (`padding: 7px 10px`) except in very long tables, and a short gap (≈2px) in the gray background between columns. Additional mouseover handlers add dotted outlines to table rows and light-blue background to table headers. The sorting (and probably the mouseovers) is done with something called tablesorter.js via jQuery. The only real imperfections are that the sorting defaults to ascending, the table headers can scroll out of view, and numeric columns are left-aligned (but correctly sorted numerically!) He's using `font-variant-numeric: tabular-nums` but I think I'd prefer the old-style numbers.

For headers and body text, he uses Source Serif Pro (by Frank Grießhammer, licensed under the SIL OFL by Adobe in 2014; it's 72 kilobytes, though that seems to be just ASCII and an accent or two, and it's just the Roman; the italic is another 38K, and there are seven more variants), falling back to Baskerville or Libre Baskerville; for source code, Liberation Mono, falling back to Consolas or even Courier. The roman Source Serif Pro is very nice, but its italic is insufficiently italic for my tastes, more like a oblique/italic hybrid. Actually, I'm not totally sure his italic font doesn't vary from article to article.

His tables of contents are similar to mine, but he flows the text around them, and uses varying text weights, sizes, and colors to distinguish importance levels — outline numbers are in light grey, text in a lighter grey. He does use sans-serif fonts for the table of contents.

Eccentrically, he adds a light gray background to `<q>`s.

Fernando Irarrázaval's blog

Fernando Irarrázaval has a very pretty blog. He's using ET Book (it's where I first saw it), falling back to Palatino for characters like ∞, and he has a .newthought class to make the first word of a blog post small-caps and half a magstep larger (1.4 rem, while the body text is 1.2 rem). He takes advantage of the ample margins on modern screens to replace "footnotes" with "sidenotes" in the margin, in smaller text (1 rem), with the numbers in red ink. (The ample margins are due to the fact that they basically don't support multi-column text. I don't know what his hand computer strategy is; hand computers barely have enough room for a single column.) Source code blocks have a light grey background, and syntax highlighting done server-side with Pygments. As you'd expect from a Tufte fan, tables have a single thin rule separating headers from body. Figures are centered but sometimes are "captioned" in the margin to the right, as well as having source code above. His `<ul>`s are without bullets, but with

plenty of whitespace, to the point of having uncomfortably short line lengths.

Firefox default styles

Firefox's default for body text on my laptop is 16 pixels with font-weight 400, for <h1> it's 32 pixels with font-weight 700, and for <h2> it's 24 pixels with font-weight 700.

As I said before, my pixels are about 161 μm, so 16 pixels is 7.3 points, which seems pretty small to me. The 14.5 points at which T_EX is set seems more reasonable, and the 19 points to which I had it zoomed is eminently readable.

Firefox's default margins for <h2> etc. are equal on top and bottom, which is semantically wrong, though it duplicates formatting that is common with T_EX; headings belong to the text below them, not the text above them. Right now I have an <h2> with 29.6px margin-bottom and a font-size of 35.7px, and an <h3> with 29.4px margin-bottom and a font-size of 29.4px. And the <h1> on top of everything has a 28.1px margin-bottom and a font-size of 42px.

Typewriter type

One complicating factor is that I use Markdown's typewriter type <pre> fairly often to "quote" things from character-cell terminals, which normally have at least 80 columns. It would be nice for those quotes to fit into the column. Here's an 80-column box:

```
0123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789
|                                                                              |
0123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789
```

If that doesn't fit nicely into the column of text, the aesthetic experience is suboptimal. With a max-width of 40em and the same font size among the paragraph text and the typewriter text it only fits up to about column 66, so I need about 21% more width, or 17.5% less, from the other point of view. Right now, I'm using 21-pixel body text, which is pretty ideal from a readability point of view. I should be able to make a reasonable compromise and get things to fit: 6.6% larger body text, 6.6% longer lines, and 6.2% smaller typewriter text; that works out to about 43em width, 22px body text, and 19px typewriter text, giving a total compression of 24.5%.

That worked nicely! And 22 pixels (which Firefox is indeed interpreting as 22 of the literal screen pixels) is still only 10 points on my screen, and I still get 25 or 30 lines in the browser window, so the larger size doesn't compromise readability.

Later I switched this to "80%" and applied it to <code> too, because once I switched to the old-style ET Book font with its small x-height, the much larger x-height of DejaVu Sans Mono made it jump out of lines unpleasantly. 80% works out to 18px rather than 19px at the default font size of 22px; at the new larger 26px size, it's 21px.

A possible tweak might be to add 10% (I don't know, 0.03em or so) of letter-spacing to body text to reduce the number of words per line, since at 43em it's a bit longer than I think is ideal.

Looser letter spacing

As Aristotle Pagaltzis taught me, to some extent, we can trade off text size against letter spacing; one of the bottlenecks in reading is apparently the crowding of information per unit area in the fovea. So text can be hard to read because, although each letterform is individually fine, there are too many of them in too little space. This means that if you shrink the letterforms but leave them at the same number of letters per square inch, you can maintain the same readability, up to a point.

I may try this with the body text: add an extra 10% letter-spacing and line-height and reduce the font-size by a corresponding 9.1%.

As it says in *The Palmer Method* I examined:

> [S]tudents sometimes think that large figures are necessarily plainer, but such is not the case. Examine carefully the diagram. At the left are figures that are absolutely plain; one could not be mistaken for another, and yet their extreme size in the small spaces makes them difficult to read. At the right are the same figures, no more perfect, but not so large. Please note carefully that these, surrounded by white paper, and much smaller, are more legible, even at a distance, than the large figures at the left.

I tried that, and also a more extreme version where I changed max-width from 45em to 54em, the font-size from 22px to 18.3px, the line-height from 1.4 to 1.68, and added .2em of letter-spacing. The result still took up about the same space and was still readable from the same distance as the normal text, but it feels noticeably less comfortable. Instead of spacious, it feels fussy. The looser line-spacing does seem like an improvement, so I increased it to 1.5.

Smaller headings

Since yesterday I've been using these sizes, which closely coincide with the size hierarchy from *The Grammar of Graphics*:

```
h1 { font-size: 2em }
h2 { font-size: 1.7em }
h3 { font-size: 1.4em }
h4 { font-size: 1.2em }
h5 { font-size: 1.1em }
h6 { font-size: 1em }
```

However, because most articles only use a single <h1> and some <h2>s and occasionally an <h3>, most headings are 1.7em and 1.4em, which is a bit large. Since I'm using a sans-serif font for all the headings (except <h1>), there's little danger they'll blend into the body text in any case, so perhaps I should make the last three levels the same size as the body text, distinguished only with different typography (say, bold, roman, and italic), permitting the use of sizes like 1.4em and 1.2em for <h2> and <h3>.

(At the moment, actually, only Low-power microcontrollers for a low-power computer (p. 2602) even used <h5>, and it only does so once; nothing used <h6>.)

So that's how I ended up with these somewhat less extreme font styles for headings:

```
h1 { font-size: 2em }
h2 { font-size: 1.59em }
```

```
h3 { font-size: 1.26em }
h4, h5, h6 { font-size: 1em }
h4 { font-weight: bold; font-style: italic }
h5 { font-weight: normal; font-style: italic }
h6 { font-weight: normal; font-style: normal }
h2, h3, h4, h5, h6 { font-family: Helvetica,Arial,sans-serif }
```

Later, after spending probably an unhealthy amount of time looking at medieval manuscripts, incunabulae, and other printed books (from many centuries), I switched to using serif fonts for headers, just with small caps, extra letter-spacing, and red ink to make them stand out. (I'm not daring enough to use the #c54 kinds of bright reds used for headings and other rubrication in incunabulae and medieval manuscripts, but #600 seems like it should work.)

Here's the current cascade from <h3> on down, with some sample body text I didn't write:

Waltz, nymph, for quick jigs vex Bud <h3>

He made also ten lavers, and put five on the right hand, and five on the left, to wash in them: such things as they offered for the burnt offering they washed in them; but the sea was for the priests to wash in.

Waltz, nymph, for quick jigs vex Bud <h4>

And there were six steps to the throne, with a footstool of gold, which were fastened to the throne, and stays on each side of the sitting place, and two lions standing by the stays:

Waltz, nymph, for quick jigs vex Bud <h5>

And the Philistines gathered themselves together to fight with Israel, thirty thousand chariots, and six thousand horsemen, and people as the sand which is on the sea shore in multitude: and they came up, and pitched in Michmash, eastward from Bethaven.

Waltz, nymph, for quick jigs vex Bud <h6>

The hand of the LORD was upon me, and carried me out in the spirit of the LORD, and set me down in the midst of the valley which was full of bones,

Topics

Slotted tape with skewed involute roulette bristles as an alternative to hose clamps and possibly screws

Kragen Javier Sitaker, 2014-07-02 (6 minutes)

An invention that could replace hose clamps and maybe screws and nails: roulette-bristle slotted tape.

If you wrap sticky tape around a thing, you can form a sort of hose clamp holding the thing, but the tension in the clamp is limited by the fact that the stickiness of the tape prevents the tape from sliding not only to loosen, but also to tighten. (The constant-tack nature of most sticky-tape adhesives also means that tape joints like this will tend to creep over time if under stress, losing whatever tightness they may have originally.)

Velcro has similar behavior, but it doesn't creep. It does loosen a bit by a fixed backlash distance upon being first attached, though.

In both cases, the clamping force is limited to the force you applied pulling the tape tight when you first made the joint, multiplied by the number of rounds the tape goes around.

But suppose instead that the bottom layer of tape has an array of round rods sticking out of it, like widely-spaced Bristle Blocks, which fit through round holes in the top layer of tape. This eliminates the creep and backlash problems, and also potentially gives you a bit of mechanical advantage: if the rods don't quite line up with the holes so you have to bend them backward to fit through the holes, then the rods form a sort of ramp that tightens the tape as it slides down them. (If it so happens that they don't quite line up in the other direction, the ramp loosens the tape, instead.)

This is a bit awkward, though, because the holes travel in an involute of the shape you're wrapping the tape around, while the rods, so far, are straight. But, with things like a 3-D printer, you could form the rods in an involute shape so that the holes in the tape naturally slides over them with no resistance.

If you skew them ever so slightly from perfect involutes, they naturally become the ramps I earlier suggested you could bend them into; this can serve to tighten the tape as it closes over them. It also means that tension on the tape tends to lift it up along the bristles. If the final upper-tape-thickness or two of the bristle is a backward ramp instead of a forward ramp, then the tape will instead tend to stay there under tension.

If the holes are more like lengthwise slots in the tape

then there is some slop in where the bristles can pass through. If you make the spacing of the bristles along the bottom tape slightly larger than the spacing of the holes along the top tape, then the tape will stretch progressively as you wrap it over further and further bristles; the slot shape then can allow the earlier bristles to slide back

in the slot as the tape stretches, avoiding the problem with sticky tape.

Ultimately you could have the tension of the tape held by the final, say, fifteen rows of bristles; if you are able to wrap the tape separately over each row (perhaps with a curved tape shape and bristles more resembling a cycloid than the involute of a circle) then you could apply the downward force on the tape against only one row of bristles at a time. (Ideally the rows would be diagonal so that this is a smooth experience.) If the ramp of the bristles, relative to the involute or cycloid or whatever, is 20:1, then the row of bristles gives you a mechanical advantage of 20:1, while the 15 rows of bristles give you an additional mechanical advantage of 15:1, for a total of 300:1.

This is better than the mechanical advantage of a screwdriver with a 25mm-diameter handle driving a 1-mm-thread-pitch screw, which is only about 79. It's more like using a small 5-cm-long wrench to drive a 1-mm-thread-pitch screw. That is to say, this is a fastening device that could easily replace the screw in many applications, unless I'm overlooking something.

If you can wrap the tape, say, five times around, before putting the holey part on top of the bristley part, then you have potentially an additional 5:1 mechanical advantage, which is like using a 25-cm-long wrench to drive a 1mm-thread-pitch screw. I'm not quite sure how you could do this, but I suspect it's possible.

If you have lateral freedom in where to connect the holey tape and the bristley tape, the aforementioned diagonality of the bristle rows can give you fine-tuning of the distance, with greater precision but less convenience than a screw. If the bristles are 1mm diameter with 2mm spacing and the rows are 20 bristles wide, then each bristle to the left or right in a row could be 50μm ahead or behind of its neighbor, providing you with that degree of fine-tuning of the distance, at the cost of some loss of strength (or doubling the width of one of the tape components).

Topics

Compact code cpu

Kragen Javier Sitaker, 2017-07-19 (3 minutes)

Suppose we have a non-stack-oriented VM intended for dense code; now maybe we can afford 16-bit instruction words (instead of Smalltalk's 8) because we don't need to spend half our words on stack manipulation and fetching local variables. We can avoid three-address instruction formats in a few different ways; the most appealing is to use something like the Mill's Belt for instruction results.

In particular, I think that the usual instruction format could probably have two operands, and I think that part of the namespace of operands should be devoted to the belt, while another part should be devoted to a traditional set of normal registers, handled perhaps in the usual way; perhaps you'd have 8 belt registers and 8 normal registers. As an alternative to handling them in the usual way, each function could have its own set of registers, or you could use rotating windows like the SPARC.

The Smalltalk VM additionally has a bunch of implicit context that goes with a method execution: you implicitly have the object's instance variables mapped into your bytecode namespace, and the method is associated with a vector of method selectors that it can invoke with its bytecodes. This may save space in the bytecode, although for the indirection to pay for itself, you probably need several methods to share the same vector.

If we take a more traditional approach, we could pack two 12-bit instructions into a 24-bit word in the usual case, or three 11-bit instructions into a 32-bit word (with one bit omitted). This gives us three or four bits of opcode plus 4 bits per operand. A special tag bit could indicate a 23-bit or 31-bit literal, at the cost of making half the opcodes (or operands) illegal in a given position. (Literals go onto the belt.)

If we estimate that each of these VM instructions are roughly equivalent to two stack-bytecode instructions, then we are getting 1⅓ to 1½ times the standard Smalltalk bytecode density, which is already world-beating. Then there's just the issue of what those 8 or 16 opcodes should do, exactly.

(Alternatively, we could pack two 16-bit instructions into a 32-bit word, leaving us a very generous 6 bits for the opcode and 5 for each operand, or 4 bits for the opcode and an even more generous 6 for each operand. This is just the same instruction density as Smalltalk.)

We at least need to be able to do arithmetic, load and store values into registers, and do conditional jumps, or at least conditional returns.

Probably at least one of the opcodes would do well to invoke a method/"send a message". Smalltalk lumps arithmetic and array access into this, too: if the object you're sending the message to happens to be a number, and the message is an arithmetic message, then it does arithmetic; if it's an array, and the message is an array element access message, then it accesses array elements.

Topics

Archival with a universal virtual computer (UVC)

Kragen Javier Sitaker, 2014-06-29 (17 minutes)

How can we keep Nintendo games playable and WordPerfect 5.1 files readable in the 23rd century?

There's a page on Wikipedia entitled "UVC-based preservation". "UVC" stands for "universal virtual computer", and the idea is that you keep old file formats from becoming unreadable by writing readers for them that run on a virtual machine that never changes. That way, if someone in 2251 discovers a long-lost WordPerfect 5.1 document containing crucial historical information, they can run it through a WP5.1 reader written in the 2010s to run on the UVC. The UVC implementation they use in 2251 will be different from the UVC implementation we use in the 2010s, but they will be compatible.

The idea here is to emulate the hardware platform WordPerfect ran on with a UVC program, which seems sensible, and the Koninklijke Bibliothee has been working on an x86 emulator using this approach, and they have JPEG and GIF87 decoders working. They've also, bizarrely, been trying to develop file-format exporters that convert things from old file formats to XML. (Raymond Lorie's reason for wanting this is that, you know, it's better to have the parse tree and text of the WordPerfect file than just the pixels that emulated WordPerfect puts on your emulated screen.)

What's necessary for a UVC from the 23rd Century to be compatible with a UVC we write today? It seems like the UVC itself needs to be well documented, contain a bare minimum of functionality, be easily testable for compliance, and have very, very few special cases in the specification, since special cases are opportunities for incompatibility; but despite that, it needs to be a reasonable target to write a compiler for. Finally, I argue that a UVC ought to have predictable performance.

The impracticality of Raymond Lorie's UVC

Lorie's UVC is specified to some extent in his paper, "A Methodology and System for Preserving Digital Data"; he explains in his rationale:

> In order to run the UVC on a future machine, an emulator of the UVC on that machine will be required; but writing such an emulator is much simpler than writing an emulator for a real machine....
>
> What is important however is that it does not need to be implemented physically. Therefore there is no actual physical cost. For example, the UVC can have a large number of registers; each register can have a variable number of bits plus a sign bit, the sequential memory, also, can be as large as desired. Speed is not a real concern since machines will be much faster in a distant future, and an emulation of the UVC on a future machine will run faster, much faster, than a machine language program running on today's machines.

And then:

> Since a UVC interpreter will need to be written in the future, the definition of the UVC must be precisely specified and preserved. ... Only the UVC machine

language is part of the Convention (although we implemented a high-level assembler, and could develop, at any time, a compiler supporting a high level language in vogue.)

But then he sort of loses the plot:

> The design goal for the UVC was not to define a minimal general-purpose computer. Instead, the idea was to develop an intuitive computer with rather powerful and flexible instructions for handling bit streams, and to take advantage of the fact that it is virtual and that performance is of secondary importance. ..., without secondary features often introduced for improving the execution performance and the memory usage. It also tries to be intuitive. ... The memory is bit addressable; there is no notion of byte, word or alignment. This is extremely convenient for manipulating bit streams. ... The interface allows for variable length registers, allowing for manipulation of large addresses, and of course, large integer data items. A register expands to the left when needed. There is no overflow condition. ... The instruction length is also variable; it may have a variable number of operands.

This seems rather strongly opposed to the primary goal of "Since a UVC interpreter will need to be written in the future, the definition of the UVC must be precisely specified and preserved." Instead he has created a virtual machine specification that is unnecessarily difficult to implement, unnecessarily difficult to specify precisely, and impossible to test — while you can verify that an implementation of the UVC correctly handles, say, 128-bit integers or 256-bit integers, you can't test that it correctly handles integers of all possible lengths. And in fact it is nearly guaranteed that it will not. (The machine I'm typing this on has RAM and disk sufficient for a single 1.2-petabit integer, and is pretty much guaranteed to not be able to execute an operation involving two petabit integers.)

Plauger explained in one of his books why the C standard only required compilers to handle things up to certain limits: block nesting up to a certain depth, variable names of a certain length, and so on. Some earlier standard (I am guessing ALGOL-60 or perhaps Pascal) had required compilers to handle arbitrarily long names successfully. Somebody published standards-compliance test suites that included a test for long names, but since they couldn't test arbitrarily long names, they tested names of some large but finite length: 16 letters, maybe. And all the compiler vendors made their compilers pass the tests.

So, in effect, there was still a limited portable name length, even though the standard specified that there shouldn't be such a limit. This requirement in the standard simply resulted in the limit being undocumented.

Lorie ends up with 21 machine instructions and a segmented memory model. The instruction encodings are not specified in the paper, making it impossible to implement a UVC interpreter from the paper. He admits that his UVC emulator limits registers to 32 bits, memory segments to 8 megabytes and 100 registers each, and so on. Also, he neglects to mention crucial things like what happens when the "divide" instruction gets a division by zero.

It seems to me that if you take seriously the ideas that "speed is not a real concern,", "the definition of the UVC must be precisely specified and preserved,", and "develop a compiler," you will end up with something quite different — something much closer to a "minimal general-purpose computer", with no thought whatsoever given to "powerful and flexible instructions" and "develop an intuitive computer". Instead you'd want a virtual machine that was very easy to write an emulator for, very easy to test an emulator for,

and unlikely to have subtle bugs or implementation-dependent behavor.

In short, you'd want something much more like Brainfuck, Wireworld, or Urbit Nock, than like the UVC in Lorie's paper.

Brainfuck

Brainfuck (inspired by Wouter van Oortmerssen's FALSE) is specifically designed to be as easy as possible to write an emulator for. The original implementation, written in 1993 by Urban Müller for AmigaOS, was 240 bytes, and was a compiler. Several Brainfuck compilers under 200 bytes have been written; Brian Raiter's 166-byte Linux compiler is notable. It has 8 instructions with no operands, two registers, simple linear memory, and I/O. The memory cells are guaranteed to be 8 bits, and you're guaranteed at least 30,000 of them.

Brainfuck is very hard to program in (thus its name) but people have written real programs in it; Linus Åkesson wrote the Game of Life, for example, and Daniel B. Cristofani wrote a Brainfuck interpreter in Brainfuck, and an anonymous author wrote a implementation of DVD CSS decryption in Brainfuck.

If you somehow managed to write an Nintendo NES emulator in Brainfuck, you could be sure that any future computer could play NES games as soon as it had a sufficiently fast Brainfuck interpreter. Implementing a Brainfuck interpreter is easy (I wrote my first one in 22 minutes, in C, at 4 AM, and now I am running the Game of Life in it, veerrrry slooooowly) and so we can be sure that programmers in the future will be able to do it too.

But Brainfuck, Turing-complete and I/O-enabled though it may be, is probably not a reasonable compilation target. It lacks not only function pointers (as I think Lorie's machine does) but also functions and, in some sense, pointers, and it isn't clear to me how to, say, implement a linked list.

And even Brainfuck has incompatibilities between interpreters. Some interpreters, for example, provide bignum cells, which means you can't zero an arbitrary cell just by repeatedly incrementing or decrementing it until it reaches zero (e.g. with `[-]`); and there are different, incompatible, and ambiguous ways of handling end-of-file on input (0, -1, or unchanged, any of which could be a valid input byte).

Brainfuck is also a challenge to implement efficiently; copying a value from one cell to another, for example, involves alternating between decrementing the value in one cell and incrementing the value in the other. Recognizing this idiom can speed up your interpreter by a couple of orders of magnitude.

Wireworld

The Wireworld cellular automaton is, in some sense, even more extreme than Brainfuck: when you start, it's a challenge to figure out how to AND two bits together, adding two numbers is at least a one-day project, and even a single bit operation on a naïve implementation takes many thousands of instructions. It has the unfortunate problem that your circuit has no external memory; it cannot expand.

Wireworld also has even less predictable performance than Brainfuck, in the sense that a "smart" implementation can recognize

progressively bigger patterns.

Nock

The Urbit virtual machine named Nock is the first serious attempt I've seen to define a minimal computational machine for purposes of interoperability and preservation. Nock, unlike Brainfuck, is pure-functional; its data model can be expressed in OCaml as:

```
type noun = Atom of int | Cell of noun * noun
```

except that Nock, like Lorie's machine and unlike OCaml, uses arbitrary-precision integers. (Yarvin writes that it's "common" to represent whole text files as atoms, so we should expect integers of hundreds of thousands of bits at least.)

Nock has five instructions: `*` function invocation, `?` type dispatch, `+` increment, `=` equality testing, `/` array indexing; but function invocation has 11 opcodes defined (as the numbers 0 through 10, as atoms) which do things like function composition, the S combinator, and the ternary operator.

Open Firmware/OpenBoot

Open Firmware defined a Forth bytecode "FCode" for non-x86 PCI card BIOS drivers. However, its core functionality is ANS Forth, and as such, it's relatively large and complicated. There's no chance people will stop introducing subtle changes in the semantics of Forth operations over the next few centuries.

The λ-calculus and the ς-calculus

In light of the above horrors, perhaps it might make sense to use something like the λ-calculus or core Lisp (cons, car, cdr, atom, null, cond, eq, nil, lambda, label, quote) as our UVC? Or, if we were interested in human-editability, Abadí and Cardelli's ς-calculus or object calculus might be an almost equally simple choice?

The problem with these as they stand is twofold:

- Implementing them on real computers, while achievable and in fact achieved many times over, is substantially less trivial than implementing Brainfuck. Nobody is going to write a 166-byte λ-calculus compiler.
- You need to augment them with some kind of numerical operations. Brainfuck has increment, decrement, and while-nonzero. Nock has increment and equality-test.

Performance

Most of the things I've mentioned so far require loop analysis to get decent performance — adding X to Y by repeated incrementation will take X steps with a simple interpreter, but only 1 step with an interpreter that is able to analyze the loop's performance.

While it's unavoidable that different implementations of a virtual machine will differ in performance by potentially large factors, I don't think it's unreasonable to ask them to be in the same big-O complexity class! So I think you probably want at least addition in the basic operations.

(In some sense, if your fundamental data items are limited in size, then this O(N) slowdown becomes an O(1) slowdown; 255 increments is still only a constant factor larger than a single addition.)

So I think it might be worthwhile to consider RAM machines with some kind of built-in arithmetic.

This also argues against including garbage collection in the basic model, which is necessary for computational models like the λ-calculus, pure Lisp, and Nock. Maybe it's not a strong argument, though.

SUBLEQ, an OISC

SUBLEQ is of the family of one-instruction-set computers. It's an assembly instruction that is sufficient to construct arbitrary computations without needing any other instructions.

The SUBLEQ or SBN instruction is "subtract and branch if less than or equal to zero": `SUBLEQ a, b, c`, where all three operands are memory addresses, subtracts the value at a from the value at b, branching to c if the result is negative.

This instruction is sufficient for arithmetic, conditional and unconditional jumps, and memory transfer; but it does not, in itself, support indirection of either memory reads, memory writes, or program transfer. So it doesn't give you pointers, arrays, structs, or functions.

There are several ways to get them, though. You can use self-modifying code to modify any of the addresses in an instruction, although not to pass a return address to a subroutine. You can redefine the instruction's addresses to be indirect, or indeed support multiple addressing modes. You can memory-map the program counter.

MOVE machines

Once you start memory-mapping core parts of the processor, though, you're getting into MOVE machine territory (aka "transport-triggered architecture"). MOVE machines are another kind of OISC. If you map the following things into fixed places in memory:

- the program counter;
- an index register;
- the memory cell pointed to by the index register;
- a "subtractor" register which subtracts upon write rather than overwriting;
- a "signum" register which reads as -1, 0, or 1 after having a negative, zero, or positive value written to it, or alternatively after the subtractor produces such a result;

then you have a Turing-complete machine that supports pointers, array indexing, function calls, function pointers, arbitrary arithmetic, loops, if-statements, structs, exception handling, multithreading, dynamic dispatch, and so on. You handle a pointer by writing the pointer to the index register and then reading or writing the indirect cell. You zero the subtractor by subtracting it from itself. You get array indexing by subtracting an offset from the "base address" of the end of the array. You get conditional values by array indexing with the signum register, and if-statements by moving such a conditional

value to the program counter. A function call copies the stack pointer to the index register, copies the program counter to the cleared subtractor register, adds a fixed offset to it, saves the subtractor output on the stack, and then moves a constant to the program counter. And so on.

You can try to cast that into a set of instructions and/or addressing modes, but you're going to end up with more than the five things listed above.

Historically, MOVE machines have been parallel with tricky timing constraints that vary from one version to the next, and so they might not seem like great candidates for a UVC. But this one is defined in a strictly serial fashion.

Interrupts and I/O

Interrupts are arguably just an efficiency hack: instead of scanning the keyboard every so often (as the actual NES did) the keyboard invokes an interrupt handler, which context-switches away from the user code. They're frequently a source of nondeterminism and behavior that varies between processor versions.

I think that probably the right solution is just to have memory-mapped input and output ports.

Topics

- Performance (p. 3621) (149 notes)
- History (p. 3500) (71 notes)
- Small is beautiful (p. 3714) (40 notes)
- Instruction sets (p. 3526) (40 notes)
- Archival (p. 3322) (34 notes)
- Forth (p. 3461) (19 notes)
- The Brainfuck esolang (p. 3350) (5 notes)
- Cellular automata (p. 3367) (2 notes)
- Universal Virtual Computer
- Urbit
- Conway's Game of Life

Tagged dataflow

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

The whole tagged-token dataflow thing explains how to do something I'd been wondering about for a long time. It's too bad I didn't find out about it earlier!

One of the big problems in these tagged-token machines seems to be how to avoid filling up your memory. It occurs to me that some node types never increase the number of outstanding messages, others never decrease them, and a few may do one or the other according to some conditional.

If there is a scheduling executive in charge of deciding which pair to next consume from memory, it can use this node type characteristic to manage the size of memory at run-time, and thus perhaps the available parallelism. When it has available memory, it can run node types that won't decrease the outstanding messages, as long as there are any pairs of those types; and when it doesn't have available memory, it can run node types that won't increase the outstanding messages, as long as there are any pairs of those types.

The tricky node types can be handled by splitting them into three node types. Given an original node type T, we create TD, which does the same thing as T in the cases where it decreases the number of outstanding messages; TI, does the same thing as T in all other cases; and TC, which evaluates the conditional to decide which path would be taken, and then creates a pair invoking either TD or TI with the same arguments.

I think we can still end up with cases where one execution order results in filling up memory with unmatched messages, while a different execution order would chug along indefinitely inside some constant space bound. I don't know how to avoid that inside the dataflow paradigm.

XXX update with eeprom/tinycpu/fpga ideas (you can reduce code-switching overhead in a machine with small code memory by processing a lot of nodes of the same type all at once)

Topics

Exponential technology and capital

Kragen Javier Sitaker, 2016-02-18 (updated 2017-07-19) (8 minutes)

Capital goods are a feature of human life that goes back millions of years; their existence doesn't require exchange between people, or even multiple people. Capital accumulation is the fundamental mechanism that drove the Industrial Revolutions, which dramatically improved material abundance, quality of life, and life expectancy over the last several centuries. However, in the next decades, capital will be eclipsed by another factor of production, one which is often misclassified as merely another form of capital: knowledge, which behaves more like plant cultivars than like capital. This is the most significant change in the material conditions of human existence since the beginning of the Stone Age, a hundred and fifty thousand generations ago, and it is likely that people after this eclipse will not be what we recognize as "human" today, differing from us as much as chimpanzees do.

Capital goods are a feature of human life that goes back millions of years

"Capital goods" are durable goods that are only used as a factor of production, rather than being consumed and thus having a use value in themselves. At least 2.6 million years ago — a hundred and thirty thousand generations — we began splitting river-smoothed cobbles in half to get sharp-edged stones to butcher animals with, among other things. Those tools were capital goods: what we consumed was the meat from the animals, which required labor to produce, but less labor with the tools, which also required labor to produce. By taking some time off from butchering animals to make tools, paradoxically, we could butcher more animals.

This is a recursive process. You can cut green wood with a hand axe held in your hand, but if you make an adze blade and mount it on a wooden haft, you can cut green wood much more easily — including making more hafts for more adzes. So accumulating some capital increases the productivity with which you can accumulate more capital. The "Primitive Technology" video blog demonstrates this process.

Knowledge is central to this process, as indeed it is central to hunting or gardening even without any capital goods.

During the Stone Age, it seems that capital accumulation was largely limited by available knowledge. One stone adze can make you much more productive at cutting wood; two adzes can perhaps make you a bit more productive still if they are suited for different purposes; but you run into diminishing returns fairly quickly. Adzes don't appear in the fossil record until the Mesolithic. So from 2.6 million years ago until only 20 000 years ago — over 99% of the history of capital — people were apparently using choppers and hand axes without hafts, so accumulated capital perhaps did not yet recursively facilitate the accumulation of further capital; our

productivity was determined more by our knowledge than by how much capital we had accumulated.

Capital goods don't require exchange between people, or even multiple people

It should be apparent from the above that even if you are Robinson Crusoe alone on a desert island, accumulating some capital is likely to be useful. Indeed, the "Primitive Technology" video series shows a guy doing more or less just that, albeit with survival support from modern society. Beginning with only land, its anonymous author successively constructs hand axes, a rainproof shelter, a hafted stone adze and a hafted celt axe, a pottery kiln, and a cob house with underfloor fire heating and a ceramic-tile roof. He singlehandedly recapitulated the entire technological development of the Stone Age, reaching the level of our ancestors about 12000 years ago — six hundred generations — in a bit over a year. This shows convincingly that the primary limiting factor of production throughout the Stone Age was knowledge rather than capital or materials.

The principle of investing some percentage of your production in capital goods rather than consumption goods in order to achieve an exponential improvement in your material standard of living is thus not limited to scenarios where you can trade with other people; it is a principle that has worked to some extent for millions of years, recursively since the Neolithic, and spectacularly since the 18th Century.

And the reign of this principle, this fundamental aspect of our human nature, is coming to an end — not in millennia or centuries but in decades.

Capital accumulation drove the Industrial Revolutions

From the time agriculture triumphed in the Neolithic until the Industrial Revolutions began in the 1700s, the primary constraint on economic productivity was land. Land was wealth; landowners called themselves the "nobility". The agricultural productivity of the land, which changed only slowly, determined its population, which would shrink when it periodically descended into wars of Malthusian desperation when food was scarce.

[XXX: okay, if that's so, then why did the First Industrial Revolution even matter at all? Did people start living longer? They weren't fertilizing their fields with industrial products, were they?]

In the 1700s, steam engines became capable of doing useful work, first sucking water out of coal mines and then driving boats and wagons around. This distinguished a new factor of economic production, one which had previously been confused with labor — what we now call "energy". Although we had used draft animals, water wheels, and windmills to do heavy work before, steam engines immensely increased both the total wattage available and the total wattage manageable per worker. The marketing of motive energy as a commodity in the form of coal enabled economic production to be limited only by the necessary human labor to operate the machinery — and so machinery became immensely more complex

and expensive in order to be able to turn energy into consumable goods with use value.

From the 1770s until, let's say, the 1970s, the primary constraint on economic productivity was no longer land, labor, energy, or knowledge, but rather capital goods. The First Industrial Revolution pioneered mass production, as in Adam Smith's pin factory, and mechanized the production of many goods; and the recursivity of capital goods increased dramatically, as the tools, materials, and processes used by machinists to make machines advanced in leaps and bounds; but the actual machines in question were relatively specialized, and the stock of them grew only slowly. The Second Industrial Revolution reduced the need for both labor and capital to reach a given level of productivity by way of "mass production", which to a large extent was a matter of things like digging with a bigger shovel and never leaving it idle. This was the age when it was reasonable to value a publicly traded industrial or transportation company in large part by its book value, the accounting value of its assets, largely capital goods.

The Industrial Revolutions dramatically increased material abundance

The Industrial Revolutions improved material quality of life and life expectancy

Capital will be eclipsed by knowledge in the next decades

Knowledge behaves more like plant cultivars than like capital

The eclipse of capital by knowledge is as big as the Stone Age shift to capital

The end of the human race as we know it

Topics

I think I understand how to use libart's antialiased rendering API now

Kragen Javier Sitaker, 2007 to 2009 (10 minutes)

I've been trying to puzzle out the libart2 API for antialiased rendering in `art_svp_render_aa.h`. I think I understand it now.

Basically, the problem it's trying to solve is that rendering a bunch of antialiased shapes that share borders is kind of a pain.

The Alpha-Blending Approach

Rendering one antialiased shape is pretty straightforward: you just alpha-blend the edges with whatever the background happens to be at the moment. Rendering multiple antialiased shapes that are supposed to share borders, well, you end up with these "cracks" where the background shows through, as follows.

Suppose you have a black background and two light gray polygons, say 80% white and 90% white, and we're looking at three adjacent pixels, A, B, and C. If the border between the two polygons falls neatly between, say, B and C, then A is 80% white, B is 80% white, and C is 90% white. No problem.

But suppose the border runs down the middle of B. Now A is still 80% white and C is still 90% white, and ideally you'd like B to be somewhere in between, say 85% white. But with the alpha-blending approach, here's what you get. First, you render the 80% white polygon, which takes B (currently at 0%) and alpha-blends 80% white into it with a 50% alpha corresponding to its 50% coverage of B, leaving B as (B * (1 - 50%) + 80% * 50%) = 40% white. Then, you render the 90% white polygon, which also covers B to a 50% extent, and so B gets ((B = 40%) * (1 - 50%) + 90% * 50%), or 65%. 65% is not between 80% and 90%. B's color ought to be composed half and half of the two polygons' colors, but instead it's composed of three-eighths of each of their colors and one-quarter the background color.

This is, for example, the approach the `<canvas>` tag in Firefox and Safari take, and it results in transparency artifacts whenever you try to do 3-D rendering on top of it.

Other Approaches

There are other simple approaches you can take. You can decline to do antialiasing, or you can do antialiasing by rendering non-antialiased to a much larger pixel buffer and then downsampling. You can randomly sample one or several points inside each pixel rather than just using the center point. And `art_svp_render_aa` has a different approach.

art_svp_render_aa

The most transparent function in `art_svp_render_aa.h` is the following:

```
void
art_svp_render_aa (const ArtSVP *svp,
                   int x0, int y0, int x1, int y1,
                   void (*callback) (void *callback_data,
                                     int y,
                                     int start,
                                     ArtSVPRenderAAStep *steps, int n_steps),
                   void *callback_data);
```

An ArtSVP is a "sorted vector path", which is the kind of thing you reduce everything else to in libart as the last step before you try to render it into pixels.

An ArtSVPRenderAAStep is this:

```
struct _ArtSVPRenderAAStep {
  int x;
  int delta; /* stored with 16 fractional bits */
};
```

So I made an ArtSVP from an ArtVpath representing a diamond between (160, 0), (240, 120), (160, 240), and (80, 120), and I called art_svp_render_aa with a callback that just dumped out the arguments it got, scaling the delta argument down as suggested by the comment (and scaling down the 'start' argument in the same way). The results looked like this:

```
y=0 start=0.5 n_steps=3
  x=159 delta=-85
  x=160 delta=-1.52588e-05
  x=161 delta=85
y=1 start=0.5 n_steps=5
  x=158 delta=-21.25
  x=159 delta=-212.5
  x=160 delta=-1.52588e-05
  x=161 delta=212.5
  x=162 delta=21.25
y=2 start=0.5 n_steps=4
  x=158 delta=-170
  x=159 delta=-85
  x=161 delta=85
  x=162 delta=170

   ...

y=25 start=0.5 n_steps=6
  x=142 delta=-21.25
  x=143 delta=-212.5
  x=144 delta=-21.25
  x=176 delta=21.25
  x=177 delta=212.5
  x=178 delta=21.25

   ...
```

And so on until the last scan line in the region. It happened that the x-coordinates were roughly where the boundaries of my shape were

on those scan lines.

It turns out that libart wants your Vpaths to be counterclockwise, and that's why the values are negative; and the start value is 0.5 to make rounding work properly.

So this gives you a sort of map of how inside the shape each pixel is. The start value tells you how inside the shape you are at the start of the scan line, with 0.5 being "completely outside" and 255.5 being "completely inside", and the ArtSVPRenderAAStep items tell you where the degree of insideness changes along that scan line. You'll notice that, except on the first line where the scan line kind of gently touches my shape, the negative numbers always total to 255, as do the positive numbers, because each scan line goes all the way into the inside-out shape (once) and all the way back out of it.

So this is sufficient to do antialiased rendering of a single shape in the dumb alpha-blending approach explained at the top: the start and delta values give you your alpha, although on kind of a funny scale. Maybe this is how `art_rgb_svp_alpha` works; I don't have access to the libart source at the moment. But you can do it like this:

```
struct alpha_blending_shape_info {
  art_u8 *buffer;
  art_u8 r, g, b;  /* foreground! */
  int x0, x1;
  int rowstride;
  enum { even_odd_rule, art_rgb_svp_alpha_rule, interesting_rule } rule;
};

// callback for art_svp_render_aa for the antialiased polygon rendering
void alpha_blending_shape_callback(void *callback_data, int y, int start,
                                   ArtSVPRenderAAStep *steps, int n_steps) {
  struct alpha_blending_shape_info *info = callback_data;
  int value = start;
  int x = info->x0;
  int ii = 0;
  int new_x, alpha;
  for (;;) { // N + 1 fills for N steps
    new_x = (ii < n_steps) ? steps[ii].x : info->x1;
    switch (info->rule) {
    case even_odd_rule:
      alpha = 256 - abs((((unsigned)value >> 16) % 512) - 256);
      break;
    case art_rgb_svp_alpha_rule:
      // same winding rule as art_rgb_svp_alpha.  if I were literate I
      // would probably know what it's called.
      alpha = abs(value >> 16);
      if (alpha > 255) alpha = 255;
      break;
    case interesting_rule:
      // This shows more clearly what's going on under the covers with value.
      alpha = value >> 18;
      break;
    }
    if (alpha) {
      art_rgb_run_alpha(info->buffer + y * info->rowstride + x * 3,
                        info->r, info->g, info->b, alpha, new_x - x);
```

```
    }
  if (ii >= n_steps) break;
    x = new_x;
    value += steps[ii].delta;
    ii++;
  }
}
```

The Iterator Interface

But there are other functions in `art_svp_render_aa.h` as well:

```
ArtSVPRenderAAIter *
art_svp_render_aa_iter (const ArtSVP *svp,
                        int x0, int y0, int x1, int y1);

void
art_svp_render_aa_iter_step (ArtSVPRenderAAIter *iter, int *p_start,
                             ArtSVPRenderAAStep **p_steps, int *p_n_steps);

void
art_svp_render_aa_iter_done (ArtSVPRenderAAIter *iter);
```

Those look like a classic iterator pattern in C. First you have a function to initialize the iterator; then you have a function to step the iterator and get some output values from it (although no way to tell when it's exhausted); and a function to discard the iterator when you're done.

And, as you'd expect, it turns out you can reimplement `art_svp_render_aa` on top of the iterator interface as follows:

```
void svp_render_aa(ArtSVP *svp,
                   int x0, int y0, int x1, int y1,
                   void (*callback) (void *callback_data,
                                     int y,
                                     int start,
                                     ArtSVPRenderAAStep *steps, int n_steps),
                   void *callback_data) {
  ArtSVPRenderAAIter *iter = art_svp_render_aa_iter(svp, x0, y0, x1, y1);
  int y;
  int start;
  ArtSVPRenderAAStep *steps;
  int n_steps;
  for (y = y0; y < y1; y++) {
    art_svp_render_aa_iter_step(iter, &start, &steps, &n_steps);
    callback(callback_data, y, start, steps, n_steps);
  }
  art_svp_render_aa_iter_done(iter);
}
```

And it seems to work the same as the original. (I wish I had the source of libart handy; I imagine it's not very different.)

But this iterator interface is useful because it allows you to iterate through the scan lines of several different shapes, possibly in different colors, in parallel. Which means that you can calculate, for each

boundary pixel, what percentage of it is occupied by each shape. So you can get results without cracks between them in the case where the shapes share some border, unlike the alpha-blending approach, but without using a humongous amount of memory or processor time, as in the supersampled-rendering approach.

However, the interface still doesn't tell you whether a pixel is 50% occupied by shape A and 50% occupied by shape B because the two are on opposite sides of a shared border that runs through the middle of the pixel, or because shape A has a diagonal border running from upper left to lower right, while shape B has a diagonal border running from lower left to upper right, both through the center of the pixel. So at least one of those two cases will be rendered incorrectly. Perhaps more seriously, if you have two shapes in different colors but with the same border, the straightforward way of using this information will give you a jaggy border where the color of the bottom shape leaks through. I think you can avoid these cases by cleverly manipulating the geometry of the shapes so that they do not overlap before you try to render them.

Additionally, I'd think that if you were using it this way, you would want some kind of iterator in the x direction over the various ArtSVPRenderAAStep[]s that describe the scanline you're currently on. But there doesn't seem to be such an iterator facility defined in libart.

Topics

Solar computer 2

Kragen Javier Sitaker, 2017-07-19 (3 minutes)

Solar panels are now down below US$1 per watt; some of them are 22% efficient, which works out to about 220W/m² in full sunlight. The Dell Inspiron Mini 10 netbook I'm typing this on is 26.8cm × 19.7cm, or about 0.05 m²; a solar panel occupying that space would yield almost 12 W in full sunlight. The keyboard is big enough to type on comfortably.

Suppose you wanted to design a computer in this form factor that could run off such a solar panel, without a battery. If you covered one side of it with a US$10 solar cell, then at times it might have up to 10 watts available, but often it would need to run on 1W or less, and it would have to handle the loss of power well: ideally without the screen going blank or losing data or anything. Even 1W is a lot better than you can reasonably provide with a handcrank, and it's solid state.

Probably you need an E-Ink screen (so you can keep reading without using energy), a couple of different processors, a physical keyboard, some capacitors, and Flash storage would be in order (although FRAM, MRAM, or PCM might also work; I suspect MRAM has longer retention; but NAND Flash is huge compared to the others. Any kind of HTML5 site is going to suck shit at best.

E-Ink screens

https://www.digikey.com/product-detail/en/EA%20EPA20-A/1481-1130-ND/4896769 is a 172×72 pixel SPI e-paper display http://www.lcd-module.com/fileadmin/eng/pdf/grafik/epa20-ae.pdf of 59.2mm × 29.2mm for US$46.28; it runs on 3.3 volts. This is unacceptably lame compared to a Swindle screen.

http://www.aliexpress.com/item/Best-quality-6-0-eink-screen-ED060SC3-for-ebook-reader-eink-display/544963786.html is a US$50 E-ink display which is supposedly 6", 1280×1024, 5ms response time, and 16.7 million colors, which I don't trust one word of except that they'll take your US$50.

http://www.aliexpress.com/item/Original-ED060SC4-ED060SC4-LF-6-e-ink-ebook-LCD-screen-for-Amazon-kindle-2-for-PocketBook/32242651648.html is a "100% Original ED060SC4 ED060SC4(LF) 6" E-link EBook LCD screen for Amazon kindle 2 for PocketBook 301 plus for Sony PRS500 600". For US$14. I think I believe this one. The PocketBook 301+, from 2009, has an 800×600 16-level grayscale display, and this does seem to be it.

A datasheet at http://essentialscrap.com/eink/ED060SC4V2.pdf reveals details. It's 122.4mm × 90.6mm (or 137.9mm × 104.1mm around the outside), about 21% of the size of this netbook, so using two might be good: 1200×800 total pixels for US$28. (Or maybe three.) 800 pixels over 122.4mm is 153µm/pixel, so a 12-point letter is about 28px tall by 14px wide. It has a 39-pin interface, needs negative and positive 20 volts or so to run it, updates in 1000ms, and uses

600–1250mW while it's doing so, so about 1 J per screen update, or about 2µJ per pixel update, 48 µJ per tiny 4×6 letter displayed, 80 µJ per small 5×8 letter displayed, or 784 µJ per 12-point letter.

That datasheet does not reveal the protocols to use to control it, but those should be available from somewhere, or for some similar display. They're clearly parallel, so you need about 25 GPIOs to control it.

784µJ per letter is 4.7 mJ per 6-character word, so reading at 350wpm would consume 1.6 joules per minute, or an average of 27mW.

(Hmm, somewhere else I was thinking

Topics

Quasicard: a hypothetical reimagining of HyperCard and TiddlyWiki

Kragen Javier Sitaker, 2017-04-18 (updated 2017-06-09) (18 minutes)

Quasicard is a hypothetical reimagining of HyperCard and TiddlyWiki, born out of my combined frustration and contentment with reading textbooks and taking lab notes on an iPhone and doing exploratory data analysis and algorithm design in IPython/Jupyter.

Reading textbooks on an iPhone

I'm currently reading Horowitz & Hill's "Art of Electronics" on this iPhone. Using whatever downtime I have on public transport to read a page or two is a fantastic way to gradually come to understand things that have mystified me for years, sometimes decades. And, unlike most things on the iPhone, the PDF reader app doesn't break all the time when internet access is disabled. But it is frustrating.

The column width of the PDF is such that about a column and a half of text fits on the screen in landscape mode at a comfortable reading size; in portrait mode the text is slightly too small if a whole column width is on the screen, but about 36 lines of about 10 words fit (360 words). At the larger but more comfortable size where a column width fills the landscape screen width, 11 lines fit (110 words). This leads to a lot of uncontrollable fits of interaction, sliding the words on the screen around every five or ten seconds, and the sense of reading the whole book through a peephole.

While the text is still not unreadable when shrunk so that both columns are visible side-by-side on the landscape-mode screen, it is uncomfortably small, and the other column is usually not useful — the two chunks of text are not linearly sequential, unless most of the top or bottom of the page is taken up by a figure. Indeed, the text is still readable with a magnifying glass when the entire page text is shrunk onto the screen in portrait mode, but this is not comfortable.

The iPhone's Notes app fits 15 comfortably readable landscape-mode lines on the screen at once with about 12 words per line, for a total of about 180 words, with space spent on screen furniture on all four sides.

So the optimal reading experience is having somewhere around 200 words visible. However, this is still kind of shitty in the textbook case, because the book is full of schematics and graphs, and the text refers frequently to them; following these references is so much trouble that I merely accept that I will have to read through the text again more than once to understand it.

For example, p.669 contains the end of §9.8.3C, all of §9.8.3D, the beginning of §9.8.3E, two footnotes, and Figure 9.86. The §C text at the top of the page continues from a sentence that began on the previous page, refers to a footnote at the bottom of the page and the TOP201 controller chip simplified block diagram in Figure 9.83 on p.666, neither of which is visible at the same time as the §C text. The footnote refers to the second edition of the book for material that was

omitted from this edition. §D also refers to several details of Figure 9.83. The §E text continues to describe Figure 9.83, and also refers to Chapter 1x, which is published in a separate book forthcoming later this year, another footnote (which for a change is simultaneously visible), Figure 9.85 on p. 667, and Figure 9.86, which is adjacent but which requires zooming out to fit on the screen; then it continues to the next page in the middle of the sentence.

This page 669, which contains about 80 or 90 lines of text (about 800 or 900 words) and one figure, thus contains eight references to things that the reader cannot see in the PDF viewer but must remember, refer to, or hope to buy later; one of these is on the same page, and another one (the rest of §C) is on the facing page 668. This level of hypertextuality or allusivity is, I think, typical of this book, although the book is probably several levels more hypertextual than an average textbook, which doesn't have "Chapter 1x" or refer you to previous editions or put the student exercises in a separate book by a different author.

Reading this book, I was struck by the realization that printed books require such compromises due to their imposition of a fixed linear order on the underlying mangle of interconnected ideas. Such a linearization will unavoidably sometimes put significant distances between a drawing like Figure 9.83, which calls for many pages of explanation, and most of that explanation.

But the benefit of computerized hypertext systems is ostensibly precisely that they liberate us from such enforced linearity. Yet, looking at Wikipedia's page on the same subject, I find that such references to invisible things are present within the first few lines — it begins with an anatomy of an ATX power supply, and by the time I have scrolled down enough to see "D: output filter coil;", I can no longer see D, the output filter coil, in the photograph. Most of the many links in the first paragraph similarly refer to things I must remember or refer to to understand it, and unlike in the Horowitz & Hill PDF, I can tap my finger on them to see the definition; but this takes me away from the paragraph I am reading. Scrolling down to the first citation, I can tap on it to display the citation at the same time as the article; the citation window then remains unless I dismiss it, although it can change to display a different citation.

This kind of multiwindow mode could potentially be a useful mode for, for example, viewing the relevant parts of a figure as you scroll through the relevant parts of its explanation. It was attempted on the WWW with "framesets" and later iframes, but these did not work very well except for niche uses like advertising; most of the problems were not fundamental to the concept.

Beyond the issues of mere concurrent visibility of referents, there are missed opportunities. Textbooks like this one are full of formalized models of real-world phenomena in the form of equations, tables, and graphs. To learn how these models behave, it would be useful to be able to experiment with them (for example, to simulate the behavior of a switching power supply, perhaps to design one to fulfill an exercise) and use them to calculate solutions to real-world problems, without having to convert them into machine-readable form by hand, with the possible errors that entails.

Taking lab notes on the iPhone

I'm using the same iPhone to take notes on the things I try in the pottery studio, with words, interspersed photos, and time-lapse videos. Typically I take photos and videos during my studio time, then crop them and compose the text afterwards on the way home. As Bush predicted in 1945, "A scientist of the future records experiments with a tiny camera fitted with universal-focus lens." Or, anyway, that's what I'm aspiring to be doing; I can't really call myself a "scientist".

The photos and videos are extremely valuable for watching the progress of a piece through the traditionally slow process of shaping clay, and for recording the setup for many parallel experiments that can take weeks.

Unfortunately, there are a lot of things that lab notebooks really need that are clumsy or impossible on the iPhone. The iPhone can't be backed up (except to Apple's proprietary system), can't export to HTML, can't produce an audit log so that particular material can be confidently dated, can't interface with data acquisition equipment to record measurements, and can't do automatic calculations in the way that VisiCalc did on an Apple][in 1979 in 32K of RAM. It can, at least, switch back and forth between taking notes and doing calculations with its pictures-under-glass facsimile of an infix scientific calculator.

The calculation facilities needed for a lab notebook are similar to those needed for a textbook, except that you need some kind of assurance that the results of the calculations haven't changed since you took the notes.

On top of these problems, this app silently reduces the resolution of the photos you store in it, which is a potentially fatal problem if you're, for example, photographing something with text on it.

Exploratory data analysis and algorithm design in IPython/Jupyter

In IPython with Numpy and Scipy, I can load up measured data consisting of tens of thousands of data points, plot them, and immediately apply a whole panoply of signal-processing, statistical, and linear-algebra algorithms to them, plotting the results. It's far better than anything else I've seen for this, except sometimes Matlab. And it also lets me include notes in pidgin Markdown with pseudo-LaTeX for equations, and Sympy can output formulas and equations in pseudo-LaTeX or LaTeX as well.

Powers like these would be extremely valuable for a lab-notebook application.

Unfortunately, code in an IPython notebook isn't reusable for other IPython notebooks, because the atomic unit is the notebook. They can be sort of source-controlled in Git, but the file format mixes source code and output, the diffs are not readable, the diffs include ubiquitous spurious changes, and there was a backward-incompatible notebook file format change within the last few years, so the promise of auditability or even file compatibility across machines is false. Also, the results are not reproducible, because they depend on the entire Python installation, as well as the sequence of evaluations of "cells" within the notebook.

Also, of course, IPython/Jupyter is not equipped for photography,

audio recording, or other data acquisition.

Existing or Previous Hypertext Systems

HyperCard was, in Nielsen's 1995 nomenclature, a "frame-based" hypermedia system, unlike the WWW, which is a "window-based" system. The unit of hypertext was a fixed-size "card", which did not scroll (it was the size of your screen) rather than a scrollable arbitrary-size document that your screen was a window onto. To write things that were longer than a screenful, you would normally organize them into multiple different cards with links between them, although, in versions of HyperCard 2.1 and later, you had the option of putting your text (if it was just plain text) into a scrollable text field.

TiddlyWiki is a "personal nonlinear web notebook" based on WikiWikiWeb. Unlike other Wikis, the unit of editing in TiddlyWiki is smaller than an entire web page; it's a "tiddler", and you can have many "tiddlers" visible at a time, just as Twitter displays many "tweets", Facebook displays many "narcissists", or Slack displays many "messages". Rohit Khare's understanding of the 2008 web as fragmenting its "atomic" pages into "subatomic" units gave him the name "Ångström" for his startup, acquired in 2010 by Google, which I worked on briefly.

It is not a coincidence that Twitter refers to the link previews it displays with some "tweets" as "cards".

In Nielsen's terminology, TiddlyWiki is a "frame-based" system that simply displays multiple frames simultaneously:

> Nodes are the fundamental unit of hypertext, but there is no agreement as to what really constitutes a "node." The main distinction is between frame-based systems and window-based systems.
>
> Frames take up a specific amount of space on the computer screen no matter how much information they contain. Typical examples are the KMS frames and the HyperCard cards. Often the size of the frame is defined as the size of the computer screen, but that determination may not hold in all systems. Since the frame has a fixed size, the user may have to split a given amount of information over several frames if it cannot fit into one. The advantage of frames is that all user navigation takes place using whatever hypertext mechanisms are provided by the system.
>
> In contrast, window-based systems require the user to use a scrolling mechanism in addition to the hypertext mechanisms to get the desired part of the node to show in the window. Because the system can display only a (potentially small) part of the node through the window at any given time, the node may be as large as needed, and the need for potential unnatural distribution of text over several nodes is eliminated. Guide and Intermedia are typical window-based systems.
>
> A great disadvantage of window-based hypertexts is that the hypertext designer has no control over how the node will appear when the user reads it since it can be scrolled in many ways. The advantage is that windows may be of different size depending on the importance and nature of information they hold. One can imagine a window-based system that did away with scrolling and thus kept most of the advantages of both display formats.
>
> The real world is not quite as simple as the clear distinction between frames and windows. HyperCard is mostly frame-based but includes the possibility for having scrolling text fields as part of a card. Hyperties uses a full-screen display without scrolling but instead requires the user to page back and forth through a sequence of screens in case the node is too big to fit on a single screen.

Bush's original Memex proposal was a frame-based hypertext system that could display two frames simultaneously on two separate screens (or more: "he has several projection positions") in order to enable the user to add links between them.

Quasicard

Quasicard is a card-based hypertext system — currently in initial design stages — similar to TiddlyWiki, but with some important differences.

First, Quasicard card titles are optional. This sounds dumb, but it turns out to have a profound effect on the way the system can be used. (Imagine trying to use a version of Twitter that asked you to invent a globally unique title for every Tweet!) Although there isn't a hard character limit like Twatter, Quasicard cards are intended to be small enough that several of them can fit comfortably on a cellphone screen at once — about a sentence or two in length, like a line of chat, rather than a paragraph or a document. Something like 128 to 256 characters, or half that in CJKV languages. To accommodate this, Quasicard makes it easy to split an existing card into new cards with Previous and Next links, automatically generating new IDs for those cards.

Second, Quasicard normally doesn't scroll text. Instead, it displays up to however many cards will fit on your screen at once, closing whichever ones were least recently used. The last several closed cards are displayed in a list; you can look further back. Quasicard tries to find a reasonable layout on the display to accommodate the most recently used cards.

Third, Quasicard cards can include special non-interactive links which open other cards automatically as soon as they are opened, or maintain those cards open.

Fourth, Quasicard cards, like WWW URLs, can take parameters. In the case of image cards, this just allows them to be cropped to a particular area and zoomed to a particular minimum size when they open; but there are also calculation cards, which can take input data and perform some calculation on it, presenting the result in some format.

Fifth, there are calculation cards in Quasicard. You can tap on arbitrary numbers on the screen to add them to a calculation stack, enter new numbers, then apply arithmetic operations to them to create a calculation. Moreover, you can tap on other kinds of datasets to calculate on them, too.

Sixth, you can create cards in Quasicard by taking photos, recording videos, taking screenshots, recording sounds, recording your geographic location, and adding files, as well as typing text.

Command interaction

In the Quasicard user interface, commands (invoked by keyboard commands or mouse clicks or finger taps) take no arguments; the equivalent is to create a new ephemeral card with input fields in it, which can be filled by typing, drawing, or drag-and-drop; and it can have buttons. The card can then react, but it acts only with the authority it brought with it and the authority you granted it by dragging things onto it. Similarly, command output is generally provided by creating a new card. This makes Quasicard dramatically less modal than other interfaces.

The internal data of a card can easily be orders of magnitude larger than the 128–256 bytes displayed, which means that in some sense they can contain many other cards within themselves.

Topics

Smooth hysteresis

Kragen Javier Sitaker, 2019-06-11 (13 minutes)

I was looking at my example code for The Bleep ultrasonic modem for local data communication (p. 966) and came across a mention of hysteresis, which triggered the following thoughts in me.

Hysteresis in the Schmitt-trigger sense is deeply discontinuous, and maybe it would work better if it were more continuous.

And also:

How would you parallelize the computation of hysteresis over a long signal?

Continuification

In digital electronics, hysteresis on digital inputs — Schmitt triggers — is used to prevent noise on a slow transition from converting that transition into multiple transitions. Without hysteresis, you set the 1/0 threshold at some level, like 1300 mV (the TTL level), and consider anything above that level a 0, and anything above it a 1. This means that, if the voltage level being impressed on the wire by the gate you're connected to is 1295 mV during some interval, noise of anything over 5 mV can in theory result in many, many transitions between 0 and 1 and back again. You could imagine that a transition from 0 V to 5 V with a roughly linear ramp over 10 ns would result in being in the 1295–1305 mV range for about 20 picoseconds, and so you might see hundreds or thousands of glitch transitions between 0 and 1 during those 20 ps, which would be fine for combinational logic that ultimately feeds into a synchronously-clocked flip-flop, but potentially disastrous for logic intended to do something like gate a pulse train into a divide-by-10 counter.

Of course, this doesn't normally happen, because if the gate sending the signal takes 10 nanoseconds to transition, the gate receiving it normally isn't going to be able to transition in 20 picoseconds, much less 20 femtoseconds. But it can happen sometimes; for example, if there's a heavy capacitive load on the line, if the input signal doesn't come from a logic gate, if the input signal is from a much slower logic family, or if your noise situation is just totally off the hook.

So, in those situations, we use a Schmitt trigger, which moves the threshold once you cross it. Say the input is initially low and the threshold is initially at 1350 mV; once the rising voltage gets past 1350 mV, the input reads as a 1 and the input threshold snaps down to 1250 mV, and it won't read a transition back to 0 unless the voltage moves back down by those 100 mV. This means that as long as your noise voltage never quite exceeds that 100 mV of hysteresis (peak to peak), your input voltage can transition as slowly as you like without turning the noise into multiple transitions. The only thing noise will do is make the transition happen a little sooner and with a little more jitter.

So how high should you set the hysteresis so that this never happens, so your circuit will work reliably? Suppose your signal suffers from interfering additive white Gaussian noise, and the noise's bandwidth (in theory infinite, but not in practice) that includes some frequencies considerably faster than your signal transitions.

Trick question! Gaussian noise will occasionally exceed any given limit (with probability exponentially small in the square of that limit), and white Gaussian noise will occasionally exceed it twice, in opposite directions, during any given interval size. Given that all circuits are subject to non-band-limited additive white Gaussian noise (from Johnson noise and from antenna pickup of thermal radiation, if nothing else) why does this work so well, even with the 100-mV hysteresis common on the inputs of things like the STM32 microcontroller line (200-mV for 3.3-volt inputs; see Notes on the STM32 microcontroller family (p. 3176))?

Again, we are saved by the limited speed of our circuits. A sufficiently short pulse of 100 millivolts or even 10 volts won't trip the Schmitt trigger because even the Schmitt trigger, though fast, can't respond instantly. Its output is a continuously-changing signal (in the sense that over a sufficiently short time interval, the change to the output is arbitrarily small) whose response to input changes is also continuous (in the sense that an arbitrarily small change to the output can be achieved with a sufficiently-small change to the input). So it isn't enough for the input voltage to jump randomly to 10 volts for a femtosecond; it needs to stay there long enough to overcome the small but finite time delays inside the circuit.

This is aided by the local linearity of the responses: if the input jumps by 1 volt for a picosecond, the output changes about five times as fast as if it jumps by only 200 millivolts for that picosecond, assuming a picosecond is fast enough to mostly avoid nonlinear effects. Since a one-volt jump is much less likely to happen due to noise than a 200-mV jump, it provides much more information if it happens.

Contrast this with what I'm doing in my Bleep prototype code:

```
def _schmitt(diff, size):
    val = 0  # presume diff starts negative
    for item in diff:
        threshold = size/2 if val == 0 else -size/2
        val = 1 if item > threshold else 0
        yield val
```

There's no continuous output transition time here that happens faster if the input is stronger. And, I suspect, this makes the code more susceptible to noise.

But does it really? The signal being processed there is a linear function of several neighboring input samples, which means that in some sense it's an average of the signal over that window. (Hopefully of the signal, that is, rather than of something else.) Although the *transition* isn't continuous, the *probability* that a given noise sample *causes* a transition *is* continuous in its amplitude, because the average of its neighboring samples is a continuously distributed random variable. So, perhaps the smaller response to smaller signals is already sufficiently present.

And, beyond some limiting frequency — Nyquist if nothing else — high-frequency noise is increasingly filtered out before it gets to the Schmitt trigger routine. So a totally lazy way of reducing the probability of noise-induced glitches is to decimate the signal a lot, ideally after low-pass filtering it. If you decimate it to, say, four

samples per expected transition time, then at most you can only get four glitches per expected transition. This introduces more jitter, of course.

Bistable and Schmitt-trigger systems as continuous differential equations

Suppose that, instead of thinking in digital space, we think in continuous space for a while. What's the simplest system that exhibits the kind of bistable behavior we'd like to see from a logic gate, say, a buffer?

Functions

First, let's consider memoryless systems — functions — before moving on to Schmitt triggers, which are by necessity stateful.

Simply $y = x$, like an idealized buffer op-amp, won't quite cut it, because that doesn't have any signal restoration — any noise on the input will be faithfully reproduced on the output. We want something that restores signal levels somewhat, so that at least if we hook up a chain of them in series, the signal will approach the discontinuous threshold behavior we expect from logic buffers — values close to 0 or 1 should converge to 0 or 1, while values close to 0.5 should be repelled from 0.5.

We can do this as a piecewise-linear system: `x < 0.25 ? 0 : x > 0.75 ? 1 : 2*x - 1`. We can do it as a trigonometric system: $\frac{1}{2} - \frac{1}{2}\cos(\pi x)$. Or we can do it as a polynomial: $-2x^2 + 3x^2$. Wait, where did that come from?

Well, our function must have attractive fixed points at $x=0$ and $x=1$, and we'd like one around $x=\frac{1}{2}$, too. That the fixpoints are *attractive* is to say that the absolute value of the derivative there should be less than 1; and we'd like the fixpoint at ½ to be repulsive. The smallest polynomial (other than just $f(x) = x$) that's going to be able to give us three fixpoints is going to be a cubic, $ax^3 + bx^2 + cx + d$. If we use Hermite interpolation, we can arbitrarily choose values and derivatives at two points of a cubic; let's choose $f(0) = 0$, $f'(0) = 0$, $f(1) = 1$, $f'(1) = 0$. This is a simple enough case of Hermite interpolation that we can do it directly: $f(0) = d = 0$, and $f'(0) = c = 0$, so we have just $ax^3 + bx^2$, whose derivative is $3ax^2 + 2bx$; so we have $f(1) = 1 = a + b$, so $b = 1 - a$, and $f'(1) = 0 = 3a + 2b = 3a + 2(1 - a) = a + 2$, so $a = -2$, and $b = 3$.

So we have $-2x^3 + 3x^2$, with the derivative $-6x^2 + 6x$. As it happens, this function *does* have a fixpoint at $x=\frac{1}{2}$, and its derivative there is 3/2, which makes it a repulsive fixpoint. This is fortunate, but if that weren't the case, we could use Hermite interpolation with $f(\frac{1}{2}) = 0$ and some arbitrary value for $f'(\frac{1}{2})$, deriving a quintic.

We can't do two attractive fixpoints at 0 and 1 with just a quadratic $ax^2 + bx + c$, because its derivative (a linear function $2ax + b$) must average 1 over that interval to hit the fixpoints, so if it isn't identically 1, it must be less than 1 at one fixpoint and greater than 1 at the other.

Ordinary differential equations

So suppose that our output, instead of being a pure function of the input, is instead some quantity y that varies over time in a continuous way: $\mathrm{d}y/\mathrm{d}t$ always exists, though it might depend on different things.

In particular, if we want it to be attracted to some target value v,

the simplest thing to do is to set $y' = (v - y)/\tau$, where τ is some positive time constant, which is necessary for the units to be consistent. This causes y's distance from v to exponentially decay, regardless of its initial state, if v is constant. So, for example, $y' = (-2x^3 + 3x^2 - y)/\tau$ would be a reasonable description of this logic-buffer system. (It's a terrible approximation of the behavior of real-world TTL or CMOS logic gate inputs, but it has some key characteristics in common with them.)

Even if v is not constant, y will move toward it, but the rate of convergence may not be exponential. Consider, for example, $y' = (-2y^3 + 3y^2 - y)/\tau$. If you start it somewhere in the interval [0, 1], it will converge to whichever endpoint of the interval was initially closest.

Now suppose we combine this with a tendency for y to move toward x, the input: $y' = (\alpha(-2y^3 + 3y^2) + (1 - \alpha)x - y)/\tau$. When $\alpha = 0$, y exponentially decays toward x, and when $\alpha = 1$, it exponentially decays toward whichever of 1 or 0 is closer to its initial state, as before. For intermediate values of α, this single-equation system displays a kind of Schmitt-trigger-like behavior (XXX verify this) in which y follows x, but tends toward the endpoints; for any given constant value of x, y may have either one or two attractors (XXX verify this). For $\alpha > 0$, y has two attractors at $x = ½$, but (XXX I think) for $\alpha < 1$, sufficiently extreme values of x will force it to have only a single attractor.

If this equation describes a single "logic gate" and its x input is an affine function $x = \Sigma_i a_i y_i + b$ of some other gates' outputs, this can be made to converge to an arbitrary logic function, provided the Schmitt-trigger behavior isn't too overwhelming. If the a_i are nonnegative, it can only compute a noninverting logic function. (This is reminiscent of the universal-approximator theorems for neural networks, and in fact it might be a special case of them.) See Snap logic (p. 2580) for more details on such logic. XXX in this case we might as well use the a_i for the feedback path too and dispense with the separate α and the unnecessarily linear feedthrough.

Topics

- Electronics (p. 3430) (138 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Physical computation (p. 3631) (26 notes)

Human memorable secret sharing

Kragen Javier Sitaker, 2019-08-10 (2 minutes)

Suppose you wanted any three of your six grandchildren to be able to decrypt your will, so you generated a 96-bit random key to encrypt it, then used Shamir secret sharing to split the key into six subkey shares. How big would these subkey shares have to be?

Clearly you can use the Galois field of a 97-bit prime, such as GF(131083052269145407673490965609†), and then all your polynomial coefficients and polynomial points can hold 96 bits. But your grandchildren then have to memorize or otherwise safely store 96 bits, which is a pain. For example, bitwords.py encodes the random 96-bit number 263087975429510937799940094 96 as "lock fall alpha index piano verb cups barns", which is clearly memorizable but also clearly nontrivial to memorize. (And actually they need to memorize a few more bits so that we know which coefficient or point they hold.) Couldn't we do with less, since we're putting in 291 bits and only getting out 96?

Not really. Suppose we come up with some scheme such that each grandchild only has to memorize 48 bits ("hans graph na laugh", "turns high sat lust"). Now, two grandchildren colluding can rent a supercomputer and run the secret-sharing computation with their two keys and each of 2^{48} possibilities from a third grandchild. On average, 20% of the way through the search space, they'll hit one of the other grandchildren's shares, decrypt your will, and possibly hatch a plot to kill you or or one of the more favored grandchildren.

This reasoning is entirely independent of the secret-sharing algorithm — it doesn't depend on any special features of Shamir's protocol — but using a slow KDF (say, 2^{40} to 2^{48} work per decryption attempt, minutes to hours on fast silicon) might slow down the attack by a useful amount, allowing the original secret to be of size 48–56 bits without a loss of security.

† Prime verification was deterministic, but if the factoring code was buggy, I might not know. Rely on this number's primality at your own risk.

Topics

- Math (p. 3564) (78 notes)
- Psychology (p. 3669) (18 notes)
- Cryptography (p. 3397) (9 notes)

Byte prefix tuple space

Kragen Javier Sitaker, 2018-07-14 (updated 2018-07-15) (4 minutes)

Linda was a coordination language for parallel computing where you could "in" some tuples from a global blackboard (or "rd" them, which is the same thing without removing them) and then "out" some tuples; the tuple space served as a sort of hybrid of communication and storage. It's sort of like Prolog, and I think it gave rise to the family of Concurrent Prolog systems. It's really dramatically easier to program than message-passing systems.

Many new software systems are built on ØMQ (ZeroMQ) or LevelDB, which are new minimalistic software designs that combine extreme efficiency with extreme flexibility.

ØMQ is a sort of hybrid of sockets and message-queuing systems like RabbitMQ, one that doesn't necessarily require a message broker as such. Like message-queuing systems, it has message framing, permits publish-subscribe communications, and can queue messages in RAM until they are processed. Like sockets, the messages are mere strings of bytes (rather than serialized data structures with an associated type system), and producers can connect directly to consumers. In order to reconcile publish-subscribe with using mere strings of bytes, the messages can be divided into a key and a value, and the subscriptions are byte prefixes of the key, or if not present, the message.

LevelDB is sort of a modern replacement for ISAM using log-structured merge trees. It stores a set of bytestring keys, each associated with a bytestring value, which may be empty. It provides efficient batch insertion/updates/deletes, which vanilla ISAM can't, and efficient in-order traversal by key.

Both LevelDB and ØMQ are one to two orders of magnitude more efficient than the more elaborate traditional systems they can replace: ØMQ can route two or three million messages per second on my laptop, while implementations of OpenMQ are around a hundred thousand or so, and LevelDB can insert about 14,000 to 300,000 records per second, while Postgres manages about 3000. (This is on an SSD.)

So it occurs to me that it might be interesting to build a "tuple-space" system which is really a "bytestring space". Workers would attempt to "in" or "rd" keys with a given prefix, and if successful might "out" others. The bytestring space might be persisted to disk or purely in RAM, and it might be hosted on a single server, sharded across servers, or even replicated across servers. If the ins and outs are transactional, it might even be possible to make it fault-tolerant.

Zooming down to the other end of the computational scale, there are truly astonishing amounts of computational power available in tiny, cheap microcontrollers at this point (various 48MIPS Cortex-M models from Philips, ST, Atmel, and others cost under US$1 at this point) and they use a tiny amount of power — in theory an STM32L011x3/4, similar in computational power to a Sun-3 workstation from the 1980s, should be able to run at 16 MIPS for a week on a CR2032 coin cell.

But it's difficult to get them to do anything complex because they have a very small amount of memory. You might have 4K to 32K of RAM and a somewhat larger amount of Flash. If you can decompose a system into pieces that fit into the RAM, communicate via message-passing, and can manage with a somewhat sequential access to the messages, you can do decent computations on these things. The problem is somewhat similar to the problem Unix solved with pipes on the PDP-11.

So, in particular, I was thinking that you could hook up an external nonvolatile storage such as a Flash chip storing a "tuple space" and have a set of "actors", each waiting on one or more prefixes, and load a single "actor" into the RAM and feed it items from the space until it's blocked on a prefix that has no existing items. Then you could context-switch to a different "actor" and repeat the process — hopefully keeping the total number of context switches low enough that you spend most of your time running actors instead of context-switching.

Topics

Waterproofing

Kragen Javier Sitaker, 2015-09-03 (4 minutes)

Thinking about underground construction: could you fire individual rooms out of porcelain (or salt-fired terra-cotta, if you can somehow detoxify the fumes?) and join them together with silicone caulk/ gaskets to prevent leakage? The water table is high enough here that any hole in the ground becomes a well. For a sufficiently high budget, of course, you can just carve a chunk of granite or basalt out of a mountain, carve rooms into it, and bury it with a little bit sticking out; it will be waterproof, but that fabrication technique is costly! The question is how to get waterproof underground construction at a reasonable cost.

The traditional approach is to build things out of concrete and then pump the water from the inevitable leaks back upstairs.

You could probably use the traditional approach but seal the entire outside with tar or pitch, like an old wooden sailing ship, but that will probably only last a few decades.

If you have a conductive mesh shell around your construction, you could use it to electrolyze the groundwater to deposit minerals encasing your construction. They will preferentially deposit inside cracks that are filled with water thus sealing the cracks. This will only work if you have enough minerals dissolved in the groundwater. In a sense it's similar to the pumping approach, in that it will fail if you have a sustained power outage, but over a much longer timescale: months or years rather than days.

Another possibility would be to seal the entire construction with some kind of flexible barrier, like latex rubber or silicone. Latex will break down spontaneously over decades and maybe biologically over years; I think silicone should be stable over a century timescale, but it is of course more expensive.

Polyethylene might be the ideal material for such a membrane, though. It's even more chemically stable than silicone, it's biologically stable (though less so than silicone, probably: http://www.cigre.org/var/cigre/storage/original/application/883f280o3abeba34bc3bdeb8caf0050c44.pdf, even though silicones can be biodegraded http://www.green-flow.co.il/Documents/Aquatain%20AMF/%D7%o9E%D7%97%D7%A7%D7%A8%D7%99%D7%9D%20%D7%95%D7o%A0%D7%99%D7%A1%D7%95%D7%99%20%D7%A9%D7%98%D7o%97/Degradation%20of%20Silicone%20Polymers%20in%20Nature.po df http://aem.asm.org/content/65/5/2276.full), it's cheap as shit, it's plastic so that it can handle subsidence without cracking, and it's a lot stronger than silicone. If you have a 1mm LDPE or LLDPE layer in your walls, that seems like it should be plenty to keep your walls from ever leaking, at 0.94 kg/m² and €1300/tonne, that's about €1.26/m².

This is apparently called a "geomembrane", and it's usually used for landfill liners to keep stuff *in* rather than to keep stuff *out*. HDPE geomembranes (actually MDPE with filler (typically carbon black) and antioxidants) ship in 6–9 meter width rolls of 1–2.5mm thickness, which are then thermally welded together onsite to form the landfill liner. Where I live, I can buy 5-meter-wide 1mm HDPE

geomembrane for US$4.21/m² from some guy named Pedro Bernaldez up in Salta: http://articulo.mercadolibre.com.ar/MLA-555707538-geomembrana-polietileno-hdpe-500-de-ancho-x-1-mt-x-1000-mic-_JM

(A minimal living space of 16m³ might be something like a cylinder of 2m height and 3.19m diameter: 20.04m² of walls, 8m² of ceiling, 8m² of floor, figure really more like 10m² each of ceiling and floor what with cutting waste, total of 40m²: US$170 of geomembrane. This would involve 20 meters of seams, plus whatever's needed to get people and air in and out.)

Topics

A review of Wirth's Project Oberon book

Kragen Javier Sitaker, 2019-02-04 (updated 2019-03-19) (63 minutes)

I'm reading Wirth's *Project Oberon*, feeling it might be useful for BubbleOS.

Certainly I am not going to take the severe approach he claims was essential (p. 8):

> Concentrate on essential functions and omit embellishments that merely cater to established conventions and passing tastes.

To a great extent, whimsy is the entire raison d'être of BubbleOS, thus Tetris (216 lines of C), Toki (some 130, not counting the data tables), and Cuerdas Caóticas; does that doom it? Certainly many embellishments will be omitted, but by no means all. My feeling is that the embellishments are as essential as the functions.

As one simple example, Oberon presents the keyboard as an input stream of characters (p. 11). Perhaps it's an "embellishment" to deliver keyup events as well, but without it, Tetris is substantially less enjoyable. (On p. 19 we find that the Lilith keyboard was connected via RS-232, aka V24 as we find on p. 209, so perhaps it sent ASCII characters rather than multibyte key-event sequences, thus rendering keyup events impossible at the hardware level. Indeed, this is confirmed for the Ceres keyboards on p. 205. On p. 208 we find it used an appalling 300 bps baud rate, implying a horrifying minimum of 33 ms of keyboard latency.)

It's interesting to note that Oberon uses the "viewers" terminology from Cedar — as well as, at least initially, the non-overlapping, column-oriented layout of Cedar.

It's amusing that even in 1992 Wirth felt it necessary to explain what the "so-called mouse" was and how it worked (p. 12), though it had been 24 years since Engelbart's Fall Joint Computer Conference demo — and, moreover, 8 years since the mass-market introduction of the Macintosh. This naïveté perhaps explains the catastrophic "interclick" interaction design in Oberon (p. 96, p. 375, especially p. 386), despite the attention given to usability in, for example, the rejection of modes and hidden state (pp. 13–14) and memorization of abbreviations (p. 15).

I had forgotten that command texts could have textual parameters textually following them (p. 13), parsed when you middle-clicked on the command name.

The mention of traps and leaving global variables in an inconsistent state (p. 13) makes me somewhat queasy. Surely rolling back a transaction is a better response?

The note about how much simpler it is to have an event-loop system rather than preemptive multitasking (p. 14), while correct, seems rather overoptimistic to me; the ostensible consequent elimination of background computations seems like a very heavy price to pay — though at least they made the concession to practicality of an "abort character" to halt runaway infinite loops ("cntl-shift-delete" — p. 208), and p. 18 explains that background tasks such as the garbage collector do indeed exist.

The paean to the versatility of text on p. 15 seems a bit dated; it would be nice to have _hyper_text output, and with reasonable layout and styling at that, so that you can not only copy and paste filenames from a directory listing into a command line somewhere, but also click on the filename to mark it for a mass action or to pop up a menu of available operations.

I'm wary of inheritance, which was not present in Cedar and which Wirth endorses fervently at the bottom of p. 15. I've used inheritance extensively in C++, Python, JS, Java, and Ruby, and I've come to the conclusion that inheritance was a mistake.

The explanation on p. 16 of the benefits of dynamic linking — that each module is present only once in the "store", by which he means RAM — makes me think that perhaps dynamic linking's time has come and gone. I'm typing this on a machine with 4 GiB of RAM. If I have both libjpeg 8.0.1 and libjpeg 8.0.2 loaded, that wastes 360 kB of RAM, one eleven-thousandth of the total. Perhaps some benefits accrue to this flexibility that make it worth the cost? For example, perhaps I would like to test a new version of libjpeg and automatically compare its output to the old version, or perhaps I would like to port my JPEG-using programs to the new version of libjpeg one by one instead of all at once?

Perhaps more urgently, what if I want to load a copy of libjpeg linked with special testing functions for accessing the filesystem, rather than the real functions? This is very important functionality that can easily be provided by dynamic linking, but it would be catastrophic for other arbitrary things that wanted to call libjpeg to inadvertently call the copy of libjpeg under test.

The total size of the core system (p. 18) is very impressive indeed: 12,277 lines of Oberon code (186 pp.) for the kernel, filesystem, windowing system, graphics editor, device drivers, text editor, compiler, and networking stack, plus some undetermined amount of assembly code (4664 bytes, so presumably about 1200 instructions for the NS32k), all compiled together to 131,800 bytes of executable code. Of this, 4000 lines is the compiler. The graphics system in particular (Fonts, Viewers, and Display) total 5324 bytes, substantially less than the 9597 bytes of text in yeso-xlib.a at the moment, much of which has to do with interfacing to Xlib and graphics-file libraries.

The description of viewers on p. 22 makes me think that perhaps Unix processes might simplify some things after all — the `ViewerDesc` type requires a `Handler` procedure for each window to be invoked with events, while in Yeso no such thing is needed, because each window normally has an entirely separate Unix process, and in any case its events can be read by the client as it pleases. (But perhaps cisco's travails getting sshd working on IOS are a better argument.)

On p. 23 the note about inheritance of message types is very interesting — the idea of adding new derived message types as subclasses of existing message types is simultaneously fascinating and horrifying.

It's interesting to read about a stackless "task" on p. 24.

The examples of inheritance on pp. 23–24 are basically closures — they exist in order to allow a task or viewer handler proc (like a WndProc) to maintain per-instance data.

The "system management tool" on p. 32 is interesting; it's more or less a form with clickable buttons, with the nice feature that it's just a

piece of text, typable by the user and WYSIWYG anywhere. (The ↑ glyph for what was described as "A reference-character '^'" on the previous page is interesting — it's from ASCII-1963, obsolete in ASCII-1968, but still in use in Smalltalk until the 1990s or even 2000s — Wirth's sabbatical at PARC, Smalltalk's birthplace, may be relevant here.)

A thought-provoking question is how far you could go with building an editable user interface that consisted of such templates and replacements — a popup menu replacing a menubutton and vice versa. Perhaps when you click on one command button (or press a control key bound in the context), it replaces the whole form it's embedded in with a query result, simply by textual replacement. Some of this could even be done with a simple text template language. And maybe a "reveal codes" toggle would be helpful for hiding the necessary markup at times.

(Similar things existed in Gypsy at PARC at the time; the menu at the top of the screen had a section labeled "*Scan* for {}" and another labeled "*Substitute* {} for {}", where the words "Scan" and "Substitute" were italicized to indicate that they were clickable, and the curly braces enclosed their arguments, which were of variable length.)

No introduction to the syntax of the language has been given before p. 34 as far as I can tell, so the reader is left to guess that # means ≠ and the semantics of the procedure-call mechanism (although it seems to be the same as Pascal's.)

The long sequences of statements on a single line, such as "c := b; b := a; a := (c MOD 509 + 1) * 127 + ORD(s[i])" are not in keeping with my aesthetic sensibilities, but those sensibilities may be a product of C and its assembly-language roots; certainly I wrote a lot of code in such a style in the 1980s myself (in BASIC), and Knuth's "TeX: The Program" (1984) is full of lines like `else begin back_input; cur_tok ← par_token; back_input; token_type ← inserted;`, though somewhat better typeset than the Oberon code.

(This may go some distance to explaining how the Oberon compiler is only 4000 lines of code.)

The password-hashing setup on p. 34 is cringe-inducingly naïve. Also, though, I infer from the code that Oberon's INTEGER type was 16 bits, either at times or always, since all of its variables would fit easily in 32 bits (19 bits I think). See p. 258 for how it's set on the server and p. 260 for why.

The setup of Oberon.User (also p. 34) as ARRAY 8 OF CHAR seems strangely primitive, given that Wirth conceived the system after heavy use of Cedar, from which it gets its tracks and viewers. Why didn't he use ropes? Ropes don't appear at all in Oberon. (Perhaps Wirth was skeptical about immutability in general, or at least when it comes to strings; see the bletcherous inline strtok() code in Call on p. 38.)

I'm suspecting that the "flip" in the names of FlipArrow and FlipStar on p. 35 means "XOR", which I suspect is the meaning of the fifth parameter of Display.CopyPattern, given here as "2"; this could be more explicit. It's somewhat horrifying to see the cursor bitmap sizes hardcoded into the bounding code like this; it suggests that the arrow and the star were the only two sprites in Oberon, so Gutknecht (or Wirth) didn't feel that factoring out a generic

sprite-bounding system was worthwhile. (The DrawCursor function immediately afterward may be the one that's actually used, though; perhaps the Flip variants were obsolete?) (Perhaps not; on p. 40 they're assigned to the Fade and Draw methods of the Arrow and Star markets, reinforcing the XOR hypothesis.) (Further explanation of cursor management is given on p. 59, and on p. 60 we have, "markers are usually painted non-destructively in inverse-video mode.", which I think means using the XOR hack; p. 61 defines Display.invert, explained as the "mode" "s XOR d", as "2", and indeed the fifth parameter of CopyPattern is "mode". Whew.)

On p. 36 we see HandleFiller, which handles events in parts of the screen containing no windows ("viewers"). The IF M IS InputMsg THEN WITH M: InputMsg DO stuff is a bit tiresome by contrast with ML. (I'm increasingly of the opinion that garbage-collected languages should be almost purely functional. Imperative code's advantages, such as they are, are vitiated by GC — maybe there's a local optimum near Golang.) The Display.FrameMsg argument is explained on p. 48.

On pp. 36–37 we see some clearly duplicated code, suggesting that the "dead code" hypothesis for FlipArrow and FlipStar is plausible: OpenDisplay should call OpenTrack twice, but instead duplicates its contents. Perhaps at some point in the past OpenTrack did something that was undesirable in OpenDisplay.

On p. 38 we see some examples of code with added reading complexity due to having no early exits; here Install wants to avoid adding a task to the task list if it's already there:

```
t := PrevTask;
WHILE (t.next # PrevTask) & (t.next # T) DO t := t.next END;
IF t.next = PrevTask THEN T.next := PrevTask; t.next := T END
```

This translates to C as:

```
t = PrevTask;
while (t->next != prev_task && t->next != T) t = t->next;
if (t->next == prev_task) {
    T->next = prev_task;
    t->next = T;
}
```

But I think it would be more natural to write it with an early exit, avoiding the redundant test:

```
TaskDesc *t = prev_task;
for (t = prev_task; t->next != prev_task; t = t->next) {
    if (t->next == T) return;
}
T->next = prev_task;
t->next = T;
```

(I wouldn't call my variables `t` and `T`, though.)

We also see what the authors were talking about when they said that GetSelection was implemented by broadcasting a message to all viewers on p. 38. After the Viewers.Broadcast call, GetSelection fishes the information about the selection out of the fields of the

SelectionMsg — presumably one or more of the viewers mutated it, perhaps conditionally based on the timestamp (which would explain the otherwise strange comments on p. 29 about the definition of the current selection, and indeed the existence of the timestamp field, as well as its initialization here to -1.)

On p. 39 we have some hints about the character encoding, as the main loop has some random code wedged into it to compensate for the difference between the keyboard's character encoding and that used elsewhere in the system. Assuming this PDF is faithful, both encodings are unknown to me; one has € at 0x81 and the other has it at 0x80.

On p. 40 we have the initial construction of the circular linked list of tasks, starting with just the garbage collector. A separate list header structure would have almost required a special case in the task list mutations (the Install and Remove procedures on p. 38 and the task-unlinking case in the main loop near the top of p. 40) because (if Oberon is like Pascal) you can't take a pointer to a struct field — although you can pass it as a VAR parameter, and maybe that would have been adequate.

It's somewhat jarring to find Min (for INTEGERs) in module Oberon on p. 34 and Max (for LONGINTs) in module System on p. 40.

On p. 41 we find the System.SetUser command, which it turns out separates the username from the password with a "/": "WHILE (ch # "/")". (This was actually suggested by the instruction "{ type user/password }" on p. 31.) Interestingly, this constant uses the same doublequotes used for strings, suggesting that in Oberon (as in Python, but very much not as in C or Pascal) a character is a string of length 1. Also we find that the username's length limit is actually 7 characters, not 8, because of the terminating 0X.

We also see the implementation of the rules for input parsing explained on p. 31: System.SetFont (also on p. 41) has a special case for "^", as do SetColor, SetOffset, etc., all implemented with separate calls to Oberon.GetSelection. But they're missing the "@" cases (although perhaps they wouldn't apply.) At least they're all using the same Texts.OpenScanner call to tokenize the parameter text.

A thing conspicuously missing in these user commands is error handling. If the specified font name doesn't exist or isn't supplied, there is apparently no feedback to the user at all that a command was invoked but merely failed, much less why it failed and how to proceed. Omitting error handling is indeed a very effective way to reduce the amount of code in your system, but it may not be a good tradeoff.

On p. 42 we see the bitfield datetime system in its full glory. I'm glad I don't have to do date arithmetic on Oberon.

On p. 44 we have the save-unders problem that accounts for so much of the complexity of both X11 and the Blit: "Any efficient management of overlapping viewers must rely on a subordinate management of (arbitrary) sub-rectangles and on sophisticated clipping operations. This is so because partially overlapped viewers must be partially restored under control of the viewer manager. For example, in Figure 4.1(b), rectangles a, b, and c of viewer A ought to be restored individually after closing of viewer B," although A and B are reversed from the diagram, in which A is on top of B. The Blit

did indeed do this; Microsoft Windows opted instead to dispatch paint messages to the exposed viewer.

The scrollbars on p. 44 would seem to owe a lot to MacOS, although perhaps I'm mistaken and PARC had nearly identical-looking scrollbars. The sans-serif code font on p. 44 and vertical italic email font on p. 45 (perhaps they are in fact the same font?) look like they come from Smalltalk.

The custom one-click shortcuts to print on particular printers or set particular fonts in the editor on p. 44 are particularly appealing.

On p. 48, the Oberon variant of MVC is most amusing: any time any document changes, all viewers are notified so that they can update if necessary, thus avoiding the necessity to register and deregister views of a model. This is a step toward the usual ImGui approach of repainting all viewers on every screen frame. (But see p. 79 for where this falls down, p. 98 for the unification, and p. 385 for a redundant re-explanation.)

We also see that Oberon has an X11-like hierarchy of windows, though it calls them "frames". Oberon windows are pretty lightweight, though; they contain seven words (presumably 28 bytes on the NS32032 they were designing for), so you can have tens of thousands of them if you like. However, the main purpose of this hierarchy in practice seems to be the pairing of inverse-video menu texts with contents panes as a single object, since the other use (vertical tracks of viewers) doesn't seem to really need handlers or really anything other than a global list of widths.

On p. 49 we see an email from "Griesemer" on "30.10.91". Could that be the Golang Griesemer? Yes, Golang Griesemer got his Ph.D. at ETHZ under Wirth and Mössenböck in 1993 on a vector version of Oberon for the Cray Y-MP, called Oberon-V.

On p. 50 we are introduced to the "logical display area". I wonder how decoupled from displays it ended up being in practice; the XOR hack mentioned earlier suggests not much.

On p. 51 we are faced with yet another definition of OpenTrack, or the declaration of one at any rate, this time in module Viewers. What happened to the one on p. 37?

On pp. 54–55 we have a small contradiction: "according to the principle of information hiding an internal data structure is fully private to the containing module and accessible through the module's procedural interface only," yet, "Although there is no language facility to enforce it, the variable *state* is to be treated as read-only by every module other than *Viewers*." I guess once you get a pointer into the internal data structure you can navigate it as you please!

On p. 56 we have an interesting viewpoint on object-orientation:

> Message handlers in Oberon are implemented in the form of "procedure variables" that obviously must be initialized properly at object creation time. In other words, some concrete behavior must explicitly be bound to each object, where different instances of the same object type could potentially have a different behavior and/or the same instance could change its behavior during its lifetime. Our object model is therefore *instance-centered*.

The procedure variables themselves, though, at least in the cases we've seen so far, handle all the messages directed to a given viewer or other frame, not just one. So they aren't the equivalent of a C++ virtual method, but rather of a C++ vtable. This is explained in more detail on pp. 379–81.

(This is in addition to the parallel mechanism for identifying record subtypes at runtime.)

Pascal procedure variables could only be passed down the stack, not stored in records, with the consequence that this style of programming was impossible in Pascal. The reason was that Pascal had nested procedures with block scope, but no garbage collection. The same reason explains why Pascal var parameters — which are pointers that may point inside of stack frames or records, unlike Pascal's heap pointers — can only be passed down the stack, rather than stored in data structures. This restriction would seem to be unnecessary in Oberon.

I'm not a big fan of the four-deep nested IFs at the bottom of p. 56.

On p. 57 we see an observation which is in a sense very trivial from the point of view of the Actors model, but quite a mind-bomb for the Pascal crowd:

> The essential point here is the use of new *outgoing* messages in order to process a given *incoming* message. We can regard message processing as a transformation that maps incoming messages into a set of outgoing messages, with possible side-effects.

(That's all there is to the Actors model, really, except to view all of computation in that way.)

On p. 59 we see why XOR-drawn cursors flicker when something else is being drawn without double-buffering.

On p. 61 the drawing API is explained. It lacks the ability to copy a block of pixel data to or from an offscreen buffer; cursors (and fonts, p. 62) are drawn using CopyPattern, which takes a foreground-color parameter and a one-bit-deep (p. 62) Pattern.

On p. 62 we collide with the Oberon language's lack of slices and consequent weakness in dealing with variable-size bitmap data.

On p. 63 we seem to be considering double-buffered Xinerama-style-unified displays of varying color depths; double-buffering is somewhat surprising for the 1984–8 timeframe when they were building the Oberon system — the Macintosh, the CGA, and the EGA didn't have enough RAM to double-buffer in popular modes, and X-Windows (1984, X11 in 1985) isn't double-buffered — it hurt responsiveness. Working multi-depth Xinerama is surprising even today.

For now I'm going to skip over the complete implementation on pp. 67–77.

On p. 78, where they're starting to talk about texts, they explicitly call out Cedar's influence:

> Motivated by our positive experience with integrated text in the *Cedar* system [Teitelman] we decided to provide a central text management in Oberon at a sufficiently low system level.

On p. 79, they say, "It is important not to confuse this type with the far less powerful type *string* as it is often supported by advanced programming languages." Of course we've already seen a fair bit of NUL-terminated string handling in the previous chapters; and Oberon's Text type is mutable, preventing it from being used for many of the things Cedar used ropes for. Oberon texts have fonts, colors, and "vertical offsets" (which seems to mean superscript or subscript; see p. 102 for an illustration), making them essentially quite graphical objects. They have no implicit line breaks, so they are

intrinsically formatted for a particular line width. However, texts to be displayed in standard text viewers cannot have multiple font sizes! See pp. 100–101 for this disheartening development.

On p. 79 we also see the weakness of their minimalist MVC implementation explained on p. 48: there's an additional observer interface to be notified of changes to Texts — though in this case it's so broken it only supports notifying a single observer. (But see p. 98.)

On p. 80, we see that their text handling is plagued by the same kind of unnecessary multiplication of interfaces as their pixel handling — instead of having a single string type, they have three or more — ARRAYs OF CHAR, Texts (which may have properties or "looks"), and now Buffers.

On pp. 80–81 we are introduced to the Reader type, which, like an Emacs marker, points to a place in a Text without being merely a number. However, I suspect that Reader doesn't have the virtue of Emacs markers: that it moves with the buffer contents when something is inserted earlier in the buffer. We shall see. Certainly there seems to be no possibility of passing Readers instead of integers as positions to procedures that take text-position arguments. (On p. 87 we see that Readers contain a reference into the piece-chain data structure of Texts that would enable them to survive mutations in other pieces undisturbed.)

I am distinctly unimpressed with the bug-prone Scanner sum type, with its 5 different value fields and a sixth tag field to tell which is active.

On p. 82 we have this absolutely fascinating note:

> Typically, readers and scanners are… of a *transient* nature, … This fact manifests itself by the absence of any possibility to reference readers and scanners by pointers.

This is a very surprising note! It implies that in Oberon you can't declare a POINTER TO Scanner type whenever you please; perhaps only their own module is permitted to declare such a type. Presumably, though, you can embed them in other records, and reference those other records with pointers, if you want to, so I am at a loss as to what the benefit of this limitation might be.

Lower on p. 82 we have Oberon's text formatting capabilities, such as they are; they're similar to TeX or Smalltalk in that the favored approach to produce an output of ten tokens (strings, integers, and decimal fractions, say) is to invoke a sequence of ten procedures, passing the same argument.

On p. 83 we find that they have chosen to represent Texts with Bravo-style piece chains.

On p. 90 we have the curious feature that does impedance-matching between the piece-chain structure and keyboard input:

> And finally, typed characters that are supposed to be inserted into a text need to be stored on a continuously growing file, the so-called *keyboard file*.

This doesn't really cover text generated as command output, but I suppose we could have output files for all commands, too.

The keyboard file could be useful for other purposes, too; for example, you might want to look at it, or even have the end of it continuously displayed for a screencast.

We also see the usual conceit of totalizing systems like Microsoft Word: plain ASCII text files are accepted, as input files, but only

"for compatibility reasons". Ugh.

The discussion of file page immutability on p. 90 is also interesting — the piece-chain data structure they're using shares with ropes the feature of being able to include arbitrary amounts of file data without reading it from disk until needed, so for it to be completely safe, it is necessary for that file data not to be modified in the meantime. (Perhaps a change notification scheme would work under some circumstances, but Oberon seems to have no such scheme.) This is expressed on p. 91 as follows:

> Once a file page is allocated it must not be reused (until system restart).

But it's not talking just about reuse *for another file*, but also about reuse for a different version of the same file.

This is an interesting desideratum for filesystems: the ability to retain immutable on-disk snapshots of particular files. The "system restart" backdoor might be thought clearly unacceptable nowadays, requiring as it does regular reboots to prevent the filesystem from getting full. (This is elaborated on p. 163, where file pages are called "sectors".)

It's also interesting in that, although Oberon's Text isn't immutable as Cedar's ROPE is, its implementation depends on immutability — but of disk files. (But note that disk files aren't immutable; see p. 164! So this can fail.) In a way, this seems perverse — it's depending on a more expensive kind of immutability, but doesn't fully deliver the decoupling benefits of Cedar's ROPE, in that you can't just hand out Text references willy-nilly — you'll get aliasing bugs where something you handed it to goes and mutates it later. But it seems like making a copy of a Text might often be a sufficiently cheap operation as to make this forgivable, since only a small number of Pieces need to be copied. (The association of "looks" with Piece objects will multiply the number for fancy documents.)

On p. 98 we have the link between the Text change notification mechanism (p. 79) and the Viewer change notification mechanism (p. 48): changing a Text being displayed in a Viewer invokes Viewers.Broadcast with an UpdateMsg (with the typically painful and repetitive Pascal/Oberon record initialization syntax.) I suppose this means that you can't share a Text between a text frame and another kind of non-viewer user that wants to be notified if it changes, but sharing it between multiple text frames is no problem.

On p. 99 we see concern about flickering, but manifested by using bitblt for scrolling, rather than double-buffering a screen update. This is an example of extra complexity introduced by optimizations that were necessary in 1988 on a 1-MIPS machine that are no longer relevant.

On p. 100 we see the rather dismaying pronouncements:

> 1.) For a given text frame the distance between lines is constant.
> 2.) There are no implicit line breaks.

This means that there is no way to get a title in a larger font to take up more vertical space. This is justified by the desire to display text in a single pass, but is that really worth damaging the formatting quality of the text so severely?

My propfont.c spike does paragraph filling (implicit line breaks) with a proportional font, though not variable line spacing, in grayscale at 22 megabytes of text per second on one core of my 1.6GHz laptop. Assuming 1.5 instructions per cycle, this is about 109 instructions per

letter. On a 1-MIPS workstation, in a 15-millisecond screen frame, this could manage to format about 15 kilobytes of text.

On p. 101 we find that we can't even get a title in a larger font at all:

> line spacing is fixed for every text frame. Therefore, different styles of a base font are possible within a given text frame while different sizes are not.

Watching Larry Tesler's 2017 demo of Gypsy from 1981 I see that Gypsy and Bravo didn't have different sizes or different typefaces either, at least a couple of years earlier, except for italic and bold (and underlined and inverse-video) variants of the same base font.

On p. 102 we see, for the first time I've noticed, that Oberon's Y-coordinates increase upward rather than downward. Also, its font system purports to support hinted outline fonts, but on p. 104 we see that this is not the case; it only supports raster fonts.

We also see another of the lamentable COBOL-style arbitrary limitations on string sizes: font names are 32 characters long.

On p. 104 we see that in the "ultimately stable" font file format, we have a single-byte enumerated value for the font family ("*Times Roman*, *Syntax*, etc.") This seems like a choice that is almost guaranteed to produce incompatibility; since more than 256 font families exist in the world, someone must maintain a registry of font family identifying bytes, which will need new families added to it from time to time, both producing the chance of incompatibility (if two different installations of Oberon start using the same identifier for different fonts) and inflexibility (since you cannot simply add a new font file to your Oberon installation, but must also add the family to the family registry.)

(Of course, the continuous and implicit conflation of "ASCII" with "text" seems quaint in 2019.)

The font file format has a couple of other drawbacks:

- It supports raster fonts only, and apparently monochrome ones without antialiasing at that.
- There is no provision made for output device resolution: the units of "height" are unspecified. (On p. 63 we saw that Oberon output devices supposedly do know their physical height.) This presumably means that they are pixels, so you might use a height-10 font on a standard 75-dpi screen and a height-30 font on a 300-dpi printer; this precludes the kind of adaptation normally needed to preserve readability at small sizes.

The printing graphics model on p. 105 seems rather impoverished compared to PostScript, and presumably the PARC language JaM it derived from; like X11, for example, it provides only unrotated text, and seemingly no facilities for clipping or even color — its primitives completely lack composability! It's also rather depressing that (like X11 but unlike GDI and Quartz) they chose a completely different imaging model for printing and onscreen drawing.

I am skipping for the time being past pp. 105–143, which contain several source-code modules.

On p. 144 we have the Oberon philosophy on dynamic linking, namely, that you should make your linker fast enough that you are not tempted to link things statically. This philosophy does more or less prevail in modern Unix and Microsoft Windows, but it does have a certain amount of cost in the Unix context, where the loading of a

compiled program is conflated with the initiation of a process. `ldd /usr/bin/mate-terminal` on my laptop, which merely runs the dynamic linker to link the 58 dynamic libraries into mate-terminal and prints the results, takes about 30 milliseconds, which is a human-detectable period of time; that's on a CPU core that runs about 2.4 billion instructions per second. Valgrind reports that it takes about 7 million instructions in the last-spawned child process. So on a 1-MIPS workstation this would take about 7 seconds, or about 72 seconds if you believe the 2.4-billion number.

Probably, though, you can use jumps through program linkage tables (called "link tables" here on p. 144) to reduce this cost if it becomes significant; the cost of linking then diminishes to merely a merge of the tables of subroutines in all the libraries; as explained on p. 145, this is the approach Oberon uses. PC-relative addressing on modern processors eliminates the need to use these tables (or a separate library-base-pointer register) for intra-module references. (On p. 147 we see that apparently the NS32k's BSR instruction is PC-relative.)

Indirection through such a table does impose an extra cost on each static inter-module subroutine call (though not necessarily dynamic calls through function pointers.)

On p. 145 we are faced with the question of numbering procedures. How does the linker avoid trying to load module A compiled against version 1 of module B, attempting to link it to version 2 of module B with a different procedure numbering? Because the page says, "Procedure names are not needed, as they have been transformed by the compiler into numbers unique for each module."

On p. 146 we see that Oberon uses two separate registers for intra-library relocation, rather than ELF's single register (which is %ebx on i386 IIRC).

We also encounter something claimed to be microcode; I'm not clear on whether this is the actual microcode for the NS32032 CXP and RXP instructions (it seems to be; the instruction opcodes are given on p. 355), or some kind of Oberon-specific portable assembly.

On p. 156 we encounter the notion of a "rider", which is to say, a file cursor; Michael Franz gives this as one of the great advances in Oberon over other systems, but I don't understand what the difference between a Rider and a Unix file descriptor is, except that in Oberon you need to first call Old (or New) to open the file and then Set to place a Rider, rather than just calling open().

On p. 157 we encounter the presumption of filenames:

> A file system must not only provide the concept of a sequence with its accessing mechanism, but also a registry. This implies that files be identified, that they can be given a name by which they are registered and retrieved. The registry or collection of registered names is called the file system's *directory*.

What do we get if we question this? We've already seen that Oberon does question it somewhat — previously written file pages can continue to be referenced by Texts even when they are obsolete — but what if our files were anonymous? Perhaps we could include references to them in other files (in a way that is explicit to the filesystem), then run a garbage collector.

It seems that all Oberon files are open read-write; even all riders are read-write.

On p. 159 we encounter the horrifying suspicion that perhaps

Oberon files require a function call and return for every byte read — and an inter-module call and return, at that. But p. 157 allays our suspicion — there is a ReadBytes call that takes an (unsafe) variable-length array. This is elaborated on p. 163, which gives the measured speedup on the Ceres-3 as 18× (though still only 2.5 megabytes per second — to RAM! — which seems painfully slow. It's about 7000 times slower than my current laptop's in-cache RAM access of 17 gigabytes per second, according to lmbench's bcopy benchmark.)

On p. 163 we encounter the first clue that Oberon in fact used memory protection after all, though perhaps not virtual memory:

> In this table, every sector is represented by a single bit indicating whether or not the sector is allocated. Although conceptually belonging to the file system, this table resides within module Kernel, because for safety reasons it is write-protected in user mode.

(But see on p. 194 that Ceres-3 had no MMU; p. 197 goes into the details of the memory mapping.)

Also we find that the Oberon startup routine involves scanning the entire filesystem for free blocks, like jffs, to build the in-RAM sector reservation table; this seems that it would not scale well to modern disks with billions of sectors, but perhaps it depends on the locality of the file directory. However, as in Unix, the block pointers ("sector table") and indirect pointers ("extension table") in Oberon are stored in the inode ("file header"), so the locality can't be very good. On p. 190 we are told, "[T]he initialization of the sector reservation table clearly dominates the start-up time of the computer. For a file system with 10'000 files it takes [on] the order of 15s to record all files."

(The whole sector GC thing seems rather bletcherous to me, even for 1984.)

On p. 164 there is a clue as to why Franz thought riders were so awesome: in Oberon, disk buffers are associated with a particular file, rather than with the disk as in typical Unix systems. Clearly associating buffers with a particular file descriptor would lead to inconsistencies (failure to Read Your Writes, among others).

The description of the buffer semantics here make it clear that the immutability property required by Texts on p. 90 is not in fact provided; you can change the contents of a Text by rewriting parts of its underlying disk file, though not by writing an entire new version of the file and then invoking Register to update it.

On p. 174 we see that Oberon's Register includes an fsync() (Unbuffer) to avoid losing data. My recent experiences with recalcitrant USB drives make me wonder if some kind of limit to dirty in-memory data is needed to guarantee responsiveness in cases like this — Oberon's four buffers (4KB) per file, for example, ensures that this Unbuffer call won't take that long.

Also on p. 174 we see that they didn't know about weak pointers, so they hacked the garbage collector's root set to compensate:

> there exists a pitfall that is easily overlooked: all opened files would permanently remain accessible via root, and the garbage collector could never remove a file descriptor nor its associated buffers. This would be unacceptable. We have found no better solution to this problem than to design the garbage collector such that it excludes this list from its mark phase.

On pp. 174–175 we see that they use a 24-way B-tree to map file pathnames (capped at 32 characters) to sector addresses, where their inodes are stored, though it's defined naïvely without hysteresis and

can thus thrash splitting and unsplitting the root when oscillating about a crucial number of entries. This choice is justified on simplicity and efficiency grounds on p. 189 and contrasted with Unix's hierarchical filesystem, but the simplicity grounds make me suspect that the authors never read the Unix filesystem code, which used a linear search through an unordered directory at the time, and indeed in many later implementations.

It's unfortunate that the B-tree code wasn't generalized and exported to provide a general-purpose ISAM facility.

On p. 176 we see that FileDir.Enumerate is in the "non-public" API of FilesDir (the reason being given on p. 183); also, I note that it takes a callback but no userdata. Does Oberon support nested procedures with lexical scope, as Pascal did? The notes about BSR on p. 147 didn't suggest that the regular procedure-call sequence involves passing a context pointer, as nested procedures normally require (except when circumvented with GCC's stack-trampoline hack).

On p. 192 we find an RT-11-style/VMS-style command-line switch "/date"! For the Directory command.

On p. 194 we find some explanation for the earlier ambiguity about memory protection: the Ceres-3 has no MMU!

On p. 195 the memory maps confused me by having 0 at the top and addresses increasing downwards; it led me to wonder whether the Ceres-[123] all had stacks that grew *upwards*, which they don't, as we can see from the "microcode" on pp. 146–7.

On p. 198 we find deep skepticism about the usefulness of MMUs: "a dispensible [sic] overkill for single-user workstations." Yeah, maybe if all your software is written in OCaml. Although I suppose the Macintosh was pretty useful without an MMU from 1984 until MacOS X (1999?).

On p. 199 we are introduced to the garbage collector; surprisingly, they mention reference counting (though giving the algorithm incorrectly) and the mark-and-sweep scheme they use, but not copying collectors. Oberon uses a non-incremental stop-the-world mark-and-sweep collector using pointer reversal.

On p. 202 we are introduced to the heap object format, which has a 4-byte header (ick). Maybe it was hard to do better than that in 1984, although BIBOP kind of did. BIBOP may be easier if you have an MMU, but I don't think it really needs one.

On p. 207 we have the entirely unnecessary SetMouseLimits interface, which has no equivalent in PS/2, USB, or evdev mouse protocols, but the Oberon mouse driver reports absolute rather than relative X and Y. Even if you wanted to limit the memory needed to buffer mouse events, you could buffer a ΔX and ΔY! This unconventional choice of border behavior also implies a lack of awareness of the Fitts's-Law-driven design of 1984 Macintosh menus:

The position "wraps around" in both the horizontal and vertical directions.

To be fair, though, nobody had written about this outside of Apple at the time.

On p. 209 we have "the module's initialization sequence", which I suppose is the block of code at the end of the module; I hadn't realized this was run when the module was loaded, but of course such a thing is urgently necessary in a Pascal-family language that doesn't permit you to initialize values in their declaration. In this case, it's

also being used to install interrupt handlers.

We also see the module V24, for accessing the serial port; its interface is entirely incompatible with those of file access, of handling keyboard events, and of access to Text objects.

On p. 210 I finally realized that "0FFFFC000H" is a LONGINT while "30X" is a CHAR. (This is explained in the scanner on p. 307.)

The serial-port driver on p. 210 is quite compact for a device driver. Its use of a CPU delay loop in Break (p. 211) despite being a driver for a chip with a sufficiently accurate real-time timer on it seems suboptimal. This goes unremarked here, but a similar (but more justifiable) delay loop on p. 216 merits the comment, "This is rather unfortunate."

On p. 211 we have a description of an RS-485 network. They claim to have gotten 230 kbps, close to contemporary Arcnet's 300-kbps performance. It's strange that they claim that clock accuracy limits packet length, though, since they're using SDLC bitstuffing (and in fact the whole SDLC protocol, including its 256-station addressing limit) to ensure adequate transition frequencies to maintain CDR. (On p. 222 we find that the length limit is 512 bytes.)

On p. 216 we have the account of SDLC's alternative to CSMA/CD, whose code is near the top of SendPacket on p. 214:

> Before sending a packet, it must be verified that the line is free by testing the so-called hunt bit. If the line is busy, the line is polled again after a delay. The delay is influenced by the station's address, causing all stations to have a slightly different delay. Actual collisions can only be detected by the receiver through the CRC-check at the end of the packet.

What's not mentioned is that this whole procedure, including the retry delay, blocks the entire machine; presumably this will always happen if the line is faulted in a certain way. There is no retry limit.

During the packet transmission the driver disables interrupts in order to be able to meet the SDLC chip's timing constraints, like the old Linux PIO disk driver. This seems like it could detectably interfere with interactive responsiveness: 512 bytes is 4096 bits, which is almost 18 ms at 230 kbps, and it's possible that we might have to wait for several packets from other senders to finish passing before we can send our own.

I haven't read the SCSI driver on pp. 218–220, but I'm pleasantly impressed that it's only two pages of code.

On p. 222 we find the description of the TFTP-like file transfer protocol; its lack of windowing might slow down file transfer (since the ACK packet will be delayed until whatever user interface task has the receiving processor busy is done) but I'm not sure that the SCC driver described on pp. 211–216 can buffer multiple packets anyway.

The lack of cryptography in the user authentication p. 222 is entirely unremarkable in the 1984–92 context; although it is of course unacceptable nowadays, some web hosts still use unencrypted FTP.

It's amusing that the file transfer protocol includes an instant message protocol (though no group chat). This becomes less amusing when we reach the end of Chapter 10 on p. 230 and realize that the file transfer protocol is the only application protocol supported by their network! (Well, the dedicated server software in Chapter 11 also includes an email protocol and a printing protocol, which the

workstations have the client code for.) This is somewhat disappointing in the 1984–88 context they were designing the code in, which also saw network-transparent windowing in W and X, telephony over Ethernet, and the birth of Novell, and was 16 years after Engelbart's 1968 demo in which he demonstrated screen-sharing of a windowed environment over a long-distance network.

They describe ARP on pp. 222–3, but they're ARPing for a person's initials rather than an IP address.

On p. 231 we begin the description of the dedicated server, and it's astonishing to realize that they don't have a fileserver in the normal sense — a networked facility that appears as a filesystem, just like local disk filesystems, but perhaps supports locking or something, which was the primary service provided in Novell and NFS/NIS networks in the 1980s and 1990s. Wirth had experience with such a system — Cedar's FS was done that way, and its designers report that they only allowed local disks reluctantly as a concession to user demands. Perhaps he didn't think it was a good idea?

They say their server is a Ceres-1 with two megs of RAM (half being the printer's framebuffer) and a 1-MIPS processor on p. 259.

It's amusing that the primary interface between servers is a message queue, though one substantially simpler than AMQP.

On p. 237 we are explained that an Oberon mailbox (for email, not some kind of message queue thing) is structurally restricted to 64 kibibytes and 30 messages; the format is a sort of compound-file-like filesystem-within-a-file with its own block-based free map and directory. I have to say that I think the Berkeley mbox format is both simpler and less limited (though surely less efficient; providing a mailbox directory requires reading the entire file). (Sendmail was a nightmare of complexity compared to the simpler mail server presented here.) Maybe they should have fixed their filesystem so it didn't allocate an entire kilobyte to every file, so they could have just used a file per message. It appears also that it's possible for the mailbox file to be so fragmented that a new message won't fit. (Confirmed on p. 245.)

You could imagine justifying such a complex mailbox file format if it were generalized to support arbitrary tables. For some reason, mailbox formats seem to be an attractive nuisance for ingenious people to overcomplicate, Mork being perhaps the most notorious example but far from the only one.

"Franz" (presumably Michael) appears on p. 238 in an example mail message, as do "Templ" and "Mueller".

On p. 238, we see that for no particularly good reason the mail access protocol seems to be binary: 10H for ACK, 25H for NAK, 26H for NPR, 24H for NRQ, and so on. These are the values of the field SCC.Header.typ.

On p. 252 we see the use of a temporary file (one not registered in the filesystem directory) as a buffer for data:

> it is almost the same as that for handling requests to receive a file. But instead of registering the received file, the file is inserted into *Core.PrintQueue*.

This is the same mechanism I'm planning to use to deliver buffers of pixel data to the Wercam server on Linux.

On p. 253 we see the printer rasterizing operations. (Before PostScript, all raster printers were WinPrinters.) The midpoint circle algorithm makes an appearance (as it does again on pp. 423–424),

though unfortunately the ellipse procedure is omitted.

On p. 258 we see that the cringe-inducing password hash from p. 34 is actually password-equivalent; it has a cringe-inducing justification on p. 260, which also explains that in order to protect the user database from unauthorized access, it's stored outside the filesystem, since the filesystem has no protection. Yet on p. 265, we read, "the impossibility of activating users' programs on the server station significantly reduces the possibilities for inflicting damage from the exterior." But presumably the server stores its software modules in the filesystem for use during boot?

On p. 287 we have RT-11-style command-line switches again (as on p. 192), and specifically they are to disable index and type checks, suggesting that these were major performance bottlenecks at one time. We also see the Algol-68-like expression "the mode of the object", meaning its type.

On p. 288 we see Figure 12.6, the call graph of the Oberon recursive- descent parser, which doubles as a summary of the Oberon grammar.

On p. 289 we finally discover the meaning of the suffixed asterisk on many procedure names at their definition: it indicates that they are to be compiled in such a way that they can be called from another module, so they return with an RXP instruction rather than RET. This implies that even intramodule calls of these procedures must use the slower CXPD; a trampoline function could have been used to avoid this.

In the scanner or tokenizer (module OCS) on p. 307 we have yet another of the tiresome lists of numbered constants, this time in the form of a table of token types: number, NIL, string, ident, ;, and so forth. In Lisp or Cedar you would use atoms (symbols) for this, in modern C you could use an enum, and in ML you would use a sum type. A comment seems like a distinctly poorer form for this information, but I suppose it has the advantage that, being a comment, it doesn't count against the compiler's 4000-line footprint.

The code for OCS is on pp. 307–312.

The hash function described on p. 307 (embedded in the Identifier procedure on p. 308) is very poor, particularly considering that it requires a division by 43 after every identifier character.

On p. 308 we see another peculiar feature of Oberon's grammar: logical AND is written as `&`, while logical OR is written as `OR`, and *binds more tightly than AND*, unless there's an error in the code to scan to the end of the identifier.

It's interesting that Texts has a Read procedure that produces a CHAR rather than a CHAR-with-looks.

Given the simplicity of the compiler as a whole, it seems inescapable that its chief bottleneck must have been this scanner (though the book does not, as far as I can tell, mention a profiler, and even the debugger on p. 368 seems very limited). On that basis it may seem quite surprising that they chose to make a full intermodule function call for every input character; there are 41 callsites for Texts.Read in OCS by my count. An intra-module function call might have been a bit better — both intramodule and intramodule calls had hardware instructions, but I don't know what their timings might have been — but really you'd like to use a piece of code like the following:

```
IF bufPtr = bufLen THEN refillBuffer END;
...buf[bufPtr]...; bufPtr := bufPtr + 1
```

However, you really don't want to put that in the source code itself in 41 places, and Oberon doesn't have a C-like preprocessor, a Lisp-like macro system, or a compiler capable of inlining even an intramodule function call, much less an intermodule function call.

On the gripping hand, the dedicated server in Chapter 11 (p. 231 et seq.) was of their first Ceres-1 generation of workstations and had two megabytes of RAM. The largest module in the system was OCE at 972 lines of code (see p. 18; OCS is listed there as 314 lines, matching well with its 6-page length here). Perhaps 972 lines of code might involve another 972 lines of comments, averaging 60 characters per line, for a total of 117 kilobytes. Reading such a source module entirely into RAM before beginning the scanning process would have used only 6% of the workstation's memory, and would have avoided the problem of file access during tokenization entirely. (Although `*bufp++` is considerably wordier in Oberon than in C.)

On p. 312 we see the initialization of the keyword hash table:

```
BEGIN i := KW;
    WHILE i > 0 DO
        DEC(i); keyTab[i].symb := 0; keyTab[i].alt := 0
    END ;
```

One is forced to wonder how much code could have been eliminated from Oberon by zeroizing module variables, as C does, or all uninitialized variables, as Java and Golang do.

On p. 325 we have the following extremely dubious statement:

> It is the goal of a good compiler to make use of all addressing modes offered by the computer, thereby avoiding the emission of unnecessary instructions for address computation.

Presumably the intent of avoiding unnecessary instructions is either to make the code run faster, to reduce code size, or nowadays to use less energy, but using more addressing modes is not guaranteed to move you toward any of these goals — not in 1984 with microcoded CISCs, not in 1988 with nascent RISC, and certainly not in 2019 with superscalar micro-op-based CPUs.

On p. 330 we see that the author is not American; a procedure is named AssReal.

On p. 345 we find the amusing detail that the NS32k memory block move instruction is called MOVSB, just like the 8088 (though there it's not really a block move without the REP prefix: REP MOVSB).

On p. 355 we see the full Nominal Semidestructor Thirty Two Million CISC instruction set. Sometimes the failure of this chip in the market has been attributed to poor compiler quality: on paper it was just as good as a 68000, but in practice it performed about 25% worse with the vendor's C compiler. It seems to have had a couple of special-purpose base registers (FP and SB), 8 general-purpose registers (like the 8088), and an indirect addressing mode. Unlike the 8088, its instructions are (generally) two-address instructions.

I'm somewhat alarmed by the remark on p. 356, "This solution strictly limits the size of modules" — how small do they have to be?

None of the ones listed on p. 18 was more than about 12 kilobytes.

The code generator itself occupies pp. 357–367, more or less. 56 lines of code seem to fit on a page, but, as commented before, the lines are uncomfortably long.

On p. 368 we are introduced to the very primitive debugger.

Reading p. 369 it occurs to me that perhaps the debug information the debugger reads should be the same information the compiler uses to tell itself where things are stored.

On p. 374 Wirth tells the story of how he saw Chuck Thacker's SIL written in BCPL on an Alto at PARC in 1976 and went to write his own in Modula-2 on a PDP-11, which grew gradually into the Oberon graphics editor.

On p. 385 we have the viewer-broadcast update mechanism from p. 48 explained again.

On p. 388 we have the 1980s view of object-orientation: its virtue is that it allows you to dynamically load machine code you don't have the source code to, as long as it implements a calling interface you do, by means of "the extensible record type and the procedure variable". Yuck.

On p. 407 we find that they encountered the object-graph serialization problem ("pickle") in the graphics editor, solving it with an object table. Later on pp. 435–436 we see that in the ensuing years they generalized this approach to arbitrary object graphs, using it for the repertoire of objects in a text as well.

There is a screenshot from 1995-09-21 on p. 440, showing the state of the Oberon system at that time, with desktop icons, overlapping windows, over 1000 messages in a mailbox, and still regrettable non-antialiased pixel fonts.

Topics

Laser printer oscilloscope

Kragen Javier Sitaker, 2017-04-18 (updated 2017-06-20) (2 minutes)

I've been trying to figure out if there's a way, using salvaged electronics, to store a series of high-speed analog waveforms long enough to digitize them with a low-speed ADC, in order to make a cheap DSO. To be more specific, I want to digitize a channel or two with signals at and above 20MHz, and thus a 40MHz Nyquist frequency, with perhaps 1000 samples, with at least 8 bits of precision, stored following an analog detection of a trigger signal, at varying sample rates, and then digitize them with an ADC of 6Msps or less, promptly enough to update an oscilloscope display many times per second. So those 1000 samples might be 25 microseconds, or 250 microseconds (with elements up to 2MHz), or 2.5 milliseconds (at which point you're only covering up to 200kHz).

I considered a coaxial cable delay line, but 25 microseconds is almost 4 kilometers of coaxial cable, which would impose a fatal degradation of signal-to-noise ratio. I considered driving the deflector coils of a TV tube or computer CRT with higher frequencies, but for 20MHz, the inductance involved would require such high voltages that the coils would almost certainly fail (see TV oscilloscope (p. 1253)). I considered recording the analogue waveform on a hard disk (see Disk oscilloscope (p. 713)) and then digitizing it during succeeding revolutions, which might work, but I don't know enough about the linearity and SNR.

So here's another idea along those lines: use a laser-printer mechanism, electroconductive drum and all. A 10ppm A4-size 600dpi laser printer must go through 210 mm × 297 mm of paper every 6 seconds. That paper works out to almost 35 megapixels, which means that the laser has to be modulated (and successfully recorded on the drum) at 5.8 megapixels per second.

Unfortunately, this is not nearly fast enough, and of course the linearity of the system is terrible.

Maybe the 64-microsecond acoustic delay lines that 1980s PAL electronics used? Those could be hard to find.

Better: see CCD oscilloscope (p. 1861).

Topics

Precisely how is 3 “optimal” for one-hot state machines, sparse FIR kernels, etc.?

Kragen Javier Sitaker, 2014-04-24 (8 minutes)

If you have a state machine with one-hot encoding, perhaps because you're computing your state transitions with non-inverting a logic family such as diode logic, then with two bits of output you have two states; with three bits of output you have three states, with four bits of output you have four states; and with five bits of output you can either have five states, or if you split your state into a two-bit output and a three-bit output, six states. With six bits of output you can have six states, or three chunks of two bits giving you eight states, or two chunks of three bits giving you nine.

This is an interesting thing; by using ternary instead of binary, your state machine gets the power to representation an extra state. It's not much, but it's something. And it compounds: with 24 lines of output split into 8 groups of 3, you get $3^8 = 6561$ states, while if you split it into 12 groups of 2, you get only $2^{12} = 4096$. It's about 5.7% extra information per bit: instead of getting 0.5 bits per bit, you get 0.53 bits per bit.

That is, the optimal number of states per state-machine variable is three.

Suppose you're trying to factor a target finite-length comb FIR kernel into a set of sparse FIR kernels that convolve to produce your target kernel; you want to minimize the number of multiply-accumulates per sample. If you convolve a comb with three unit impulses separated by eight empty points (zero samples) with another comb with three unit impulses separated by two, you get a comb with nine unit impulses separated by eight intervals of two empty points, at a cost of six multiply-accumulates. By contrast, if you convolve three combs with two unit impulses each, separated by two, five, and seven empty samples, you do the same number of multiply-accumulates and get a comb with unit impulses separated by two empty points --- but only eight of them rather than nine. The same is true if you replace two of those two-impulse combs with a four-impulse comb. Things get worse as you move to less-sparse kernels with five, six, or more nonzero points.

That is, the optimal number of impulses per kernel, in some sense, is three.

If you're building a balanced search tree and you want to minimize the number of keys you have to compare your probe key against, binary is the best branching factor: each comparison result gives you exactly one bit of information. But consider, instead, a menu system user interface for an idealized user who makes no errors but must read each menu item before deciding, correctly, whether to select it or not, but who may be seeking an operation not present in the menu tree. How do you minimize the number of menu items the user must read when successfully navigating the system?

If there are two items in each menu, the user must read, on

average, one of them before making a successful selection, so she will need to select lg N menu items to reach her final destination; say, 20 items if there are 1048576 total possibilities. If there are four, she must read on average two items, but need only make half as many selections: 10, while reading 20, just as before. If there are three, she must read on average 1.5, while selecting about $\log_3 N$ times --- in this case, 12.6 selections, reading 1.5 items each time, for a total of about 18.9 menu items read.

So it turns out that the optimal number of menu items, in some sense, is three.

(Real menu systems, of course, have to deal with other cognitive issues: how do you organize hundreds or thousands of things in such a way that users can guess or even remember which category each thing is in?)

Suppose you want to make a program as debuggable as possible. How big should your functions be?

To be a little more concrete, suppose you're trying to track down an incorrect result from the execution of a program that runs for a hundred billion instructions, or about a minute. If the program is built out of functions that call other functions and combine their results, and you can tell from looking at each result whether it's correct or not, maybe you can navigate to the result you want by expanding the execution history as an outline.

If each function calls three other functions, then within 24 clicks, you can make your way from the top-level result to the particular incorrect result; at each step, you have to examine on average 1.5 intermediate results, so you need to look at about 35 results.

If each function calls only two other functions, you have to look at about 37 results, which is slightly worse; and the same is true if each function calls four other functions.

So, in some simplified sense, the optimum branching factor for function-call graphs is three.

The number of nodes you have to traverse to reach one of N nodes with branching factor B is ceil(log N / log B), and if you're examining each of the B branches in the node to figure out which one to follow, you end up examining B ceil(log N / log B) branches. If we remove the discretization, we get log N (B / log B), and it turns out that B / log B has a minimum at B = e, so in practice the optimal branching factor is 3.

That's the phenomenon underlying the above simplified problems. But, as you can see, B/log B is pretty flat over the 2 to 4 region. It jumps up to infinity as you approach 1, and from e on, it grows slowly. It's only 0.4% worse than optimal at B = 3, 6% worse (as seen above) at B = 2 or 4, 15% worse than optimal at B = 5, 60% worse at B = 10, twice as bad at B = 15, and 3.2 times as bad at B = 30. So it's easy for the optimal factor in the real world to be anywhere between 2 and 10, depending on what other things you're trying to optimize.

For example:

- in the case of the state machine, your state-transition logic might be simpler with one-hot encoding among six output lines rather than some kind of conjunction over a group of two and a group of three, and that might pay the cost of the extra line;
- in the factored FIR filter, you might be trying to produce a comb

with eight or ten impulses, in which case convolving two impulses with four or five works, and nothing involving three will;
• in a menu system, you have to make the menu items reasonably comprehensible, which usually requires a larger branching factor.
• while debugging a program, changing stack frames has a cognitive cost, where you have to orient yourself in your new surroundings, and it can be practically very difficult to make a program in which each function calls only three others. The optimum for this is maybe somewhere between 5 and 18.

Topics

DReX and "regular string transformations": would an RPN DSL work well?

Kragen Javier Sitaker, 2016-09-19 (3 minutes)

I found this paper from last year on a thing called DReX, which claims to be "a declarative language for efficiently evaluating regular string transformations". Apparently a "regular string transformation" is a kind of transformational equivalent to a regular expression, and they came up with a linear-time evaluation algorithm for it; "regular string transformations" seem to be a thing they invented more or less for the paper before this one.

I think their notation could use some work though.

One of their examples is a program to delete one-line comments from a (C++) program; here's the original version from DReX paper:

```
del_slashes = split(del('/'), del('/')),
del_non_nl = iterate(del(x ≠ '\n')),
del_comm = split(del_slashes, del_non_nl),
del_comm_line = split(del_comm, del('\n')).
```

Here's an inlined version:

```
split(split(split(del('/'), del('/')), iterate(del(x ≠ '\n'))), del('\n'))
```

Here's an RPN version:

```
'/' del '/' del split x '\n' ≠ del iterate split '\n' del split
```

Here's an infix/postfix version:

```
'/' del, '/' del, x ≠ '\n' del*, '\n' del
```

How is this different from sed or whatever? Would this maybe work as an editor user interface?

They say:

…regular string transformations is a… class that strikes a balance between decidability and expressiveness. In particular this class captures transformations that involve reordering of input chunks, it is closed under composition, it has decidable equivalence…

So DReX is a combinator-based DSL that covers exactly this class with a prototype linear-time implementation (polynomial-time in program size, linear-time in input size).

The DReX combinators are:

- φ → d, where φ is a predicate and d is a function: maps any character a that satisfies φ(a) to d(a);
- split(f, g), which is unambiguous concatenation;
- iterate(f), which is unambiguous Kleene closure;
- combine(f, g), which applies both f and g to the same string and

concatenates their results (unambiguous intersection);
• f else g, which falls back to g if f fails (unambiguous union); and
• chain(f, R), where R is a regexp for tokenizing, which applies f to pairs of tokens.

Split, iterate, and chain also

have a left-additive version in which the outputs computed on each split of the string are concatenated in reverse order.

Apparently the domains of DReX functions are in some sense inspectable and don't include the whole universe of strings. Their only primitive combinator is the → combinator, whose domain I guess is the set of characters for which φ returns true.

They come up with this definition for "consistent" that allows fast (linear-time) evaluation and argue that it "does not sacrifice expressiveness". Interestingly, even though regular string transformations are closed under composition, they don't include composition because it makes the evaluation-problem PSPACE-complete, so their definition of "does not sacrifice expressiveness" may be counterintuitive.

Topics

Fast geographical maps on Android

Kragen Javier Sitaker, 2015-10-16 (9 minutes)

I'm fed up with the terribly slow performance of OsmAnd~ on my phone. But how can I get access to OSM data to reformat in different formats?

Formats and getting geodata

OsmAnd~ uses its own .obf format, and decoding that data seems to involve using their precompiled jar files, since I can't figure out how to get their code to compile.

OSM has two principal formats, an XML format called .osm, and a ProtoBuf format called .pbf, which is a few times smaller, which is documented at http://wiki.openstreetmap.org/wiki/PBF_Format#File_format and implemented in, among other things, Osmosis http://wiki.openstreetmap.org/wiki/Osmosis, which is in Debian. http://wiki.openstreetmap.org/wiki/Osmosis/Detailed_Usage_0.44o#--read-pbf_.28--rb.29 sort of explains how to use it. `osmosis --read-pbf foo.pbf --write-xml` creates a `dump.osm` file full of XML; adding a `-` argument to the end spews the XML out on stdout as Ghod intended. The XML format is documented in http://wiki.openstreetmap.org/wiki/OSM_XML.

For better or worse, the XML format has all the nodes first, followed by all the ways, then all the relations.

GDAL/OGR has support for these formats http://www.gdal.org/drv_osm.html but only from 1.10 on. GDAL in the Debian I'm running is 1.9.

A current PBF of Argentina is only 103MB: http://download.geofabrik.de/south-america/. Planet.osm is 29.3GB http://wiki.openstreetmap.org/wiki/Planet.osm so maybe the 300× smaller Argentina file is better. http://openstreetmapdata.com/ has "generalized" OSM data (sadly, in shapefiles) including coarse coastlines, land polygons, and water polygons (30 MB each).

https://github.com/scrosby/OSM-binary is a PBF library for C.

Arranging data for rapid access

The PBF data is apparently not arranged for rapid access, even though it's divided into independently decompressable PrimitiveBlocks.

My basic thought is that you can fit maybe 20 × 40 roads on my phone display before it becomes too crowded to read, so we ought to be able to produce level-of-detail summaries of the data that allow us to read some constant factor more than those 800 paths when we're drawing a display.

Like, if the display is only 2km tall and 4km wide, or less, we should be able to draw *all* the data in that region, but only access data that is actually in or near that region. If you break the full-resolution data up into 1km×1km tiles, each of which is stored contiguously, then you might need to access 8 to 15 of these tiles, which is probably

okay even on spinning rust, especially if you use Z-ordering or Hilbert curve ordering for those tiles.

Each such tile might have 200 line segments in it, mostly sharing endpoints with other line segments, with 7 decimal places of accuracy in each of lat and lon, and maybe a bit more metadata, like street names. That's probably about one endpoint per segment, and probably about 56 bits (7 bytes) of coordinates, although maybe delta-encoding can shorten that to about 32. $7 \times 200 = 1400$, so each such tile might be 2 to 4 kilobytes. Most will be smaller, a few might be bigger.

If you zoom out to almost 4km tall and 8km wide, then you may have as many as $5 \times 9 = 45$ such tiles in view, and need to access as much as 180 kilobytes of data. This is still doable in a small fraction of a second, even on my cellphone, but for zooms this large and larger, we can do better by making "summary tiles" that include the large-scale-visible features from 16 smaller tiles, but only about 200 line segments in the summary tile; so the summary tiles will also be only 2 to 4 kilobytes each.

The summary tiles should only include the most important ways; the importance ranking for "highway" ways goes something like motorway, trunk, primary, secondary, tertiary, unclassified, residential, service, living_street, pedestrian, track, footway, bridleway, raceway. Perhaps "road" should be near "residential", as should the various "_link" types.

In most areas, data will be so sparse that these 4×4 summary tiles will be able to include all the data.

The summary-tile process should be recursive, so you have metasummary tiles that cover 4×4 summary tiles, and third-level summary tiles that cover 4×4 metasummary tiles, and so on. There will be some duplication of data, but the summary tiles will never add more than a sixteenth (6¼%) to the size of the base tiles, and even an infinite pyramid of such things would only add a fifteenth (6⅔%). But Argentina is only 2.7 million km², so you only need six levels of summary tiles.

In this way, no view will *ever* need to access more than 180 kilobytes of data, plus the bounds as it walks down the tree to find the tiles it needs.

Build process

I feel like even on my netbook the 103 megabytes of Argentina data should fit in my 2 gibibytes of RAM, especially since I don't care about the users, versions, timestamps, or changesets of nodes, just their lat, lon, id, and tags. Generating the XML file on disk, at 150+ bytes per node, is probably not an ultra great idea; it would be about a gig.

For estimating sizes from the XML:

```
perl -lne '$tag{$1}++ while /<(\w+) /g; END { for my $tag (keys %tag) { print "$to
ag $tag{$tag}" }}'
```

```
  <relation id="56688" user="kmvar" uid="56190" visible="true" version="28" chan
   <member type="node" ref="294942404" role=""/>
   ...
```

```
<member type="node" ref="364933006" role=""/>
<member type="way" ref="4579143" role=""/>
...
<member type="node" ref="249673494" role=""/>
<tag k="name" v="Küstenbus Linie 123"/>
<tag k="network" v="VVW
```

Results:

entity	2014 count	est. 2015 count	actual 2015 count
byte	65M		103M
relation	10,913	17,292	24,878
member	228,594	362,233	482,756
way	621,045	984,117	1,169,064
tag	1,666,072	2,640,083	4,468,399
node	6,597,342	10,454,249	10,697,272
nd	8,139,986	12,898,747	12,791,230

So this suggests that the average node participates in ("nd") about 1.2 different ways and thus about 2.2 different line segments; it's almost cheaper to store the nodes redundantly, using an extra 10% (6 bits), rather than indirect node access through some kind of ID, which is necessarily bigger than 24 bits each. The high tag count is a surprise to me, and I assume that it's because ways have a lot of tags, but I should look.

The 65-megabyte 2014 Argentina PBF I downloaded by mistake has only 8.1 million "nd"s in it and takes Osmosis about 5 to 7 minutes to decode into XML. This would put my estimated-8-bytes-per-node optimized binary format at 65 megabytes, or about 103 megabytes for the current dataset, consisting of about 32k to 128k tiles.

The 2015 dataset took Osmosis 11 minutes to decode.

I should be able to do at least the initial build process in Python rather than C, despite it being in-RAM on this netbook, because a dict entry in Python only weighs about 128 bytes; so 10 million nodes will only weigh about 1.3 gigs. I should probably test on some smaller data first. If that doesn't work out, I can probably hack together something in C, or I guess I could put the node data into files and use an external sort, or maybe use SQLite or Postgres.

Reducing the ways to sequences of (lat,lon) tuples, plus a bit of metadata, as a sort-merge job, would involve these steps:

- Reduce the ways to 13 million (nodeid, wayid, seqno) tuples in 325 megabytes. Sort.
- Reduce the nodes to 10 million (nodeid, lat, lon) tuples in 400 megabytes. Sort.
- Merge the two into 13 million (wayid, seqno, lat, lon) tuples in 600 megabytes. Sort.
- Coalesce runs with the same wayid into (wayid, [(lat, lon) ...]) structures.

An alternative multipass strategy would involve decoding the PBF

file several times, each time computing the tiles in a particular geographical area and ignoring the other data. This is a more or less linear time-space tradeoff: I can use 10% of the memory in exchange for running the job in 2 hours instead of 12 minutes.

Topics

- Programming (p. 3658) (286 notes)
- Hand computers (p. 3492) (10 notes)
- Serialization (p. 3707) (6 notes)
- Datasets (p. 3402) (5 notes)
- Geographical information systems (GIS) (p. 3473) (3 notes)
- OpenStreetMap (p. 3615) (2 notes)
- Android (p. 3318) (2 notes)
- Protocol Buffers

Bootstrapping instruction set

Kragen Javier Sitaker, 2018-11-06 (updated 2019-05-03) (19 minutes)

Chifir is inspirational. It's the first archival virtual machine good enough to criticize. You can implement the CPU in a page of C (my version is 71 lines and took me 32 minutes to write), and then all you need is screen emulation, which took me another 40 lines and 28 minutes.†

Archival virtual machines have applications beyond archival. Independent implementations of the virtual machine defeat Karger–Thompson attacks at the virtual machine level and potentially the hardware level as well. And a stable deterministic virtual machine enables the publication of reference implementations of other virtual machines and deterministic rebuilds of derived files, such as compiler output, database indices, or raw pixels output from an image codec.

Alas, Chifir is not good enough to actually use as a *general-purpose* archival virtual machine, which of course is not its intended purpose. Any of the following problems would by itself be a fatal flaw (for a general-purpose archival virtual machine):

• The specification does not include test cases, so there is no way to debug a candidate implementation, and in particular to easily disambiguate the natural-language specification of the instruction set. In fact, as far as I can tell, not even the Smalltalk-72 image described in the paper has been released. This is important because when I added screen output to my implementation of Chifir, my first test program was a single-instruction infinite loop, which had a bug and also found a bug in my implementation of Chifir.
• Also, the instruction set contains unspecified behavior, including access to out-of-bounds memory addresses, execution of any of the 4294967280 undefined opcodes, division by zero, the appearance of pixels that aren't all-zeroes or all-ones, and arguably the rounding direction of division. Unspecified behavior is a much worse problem for archival virtual machines than for the systems we're more familiar with, because presumably the archaeologist implementing the virtual machine specification has no working implementation to compare to.

• The straightforward implementation of the instruction set using `for (;;) { ... switch(load(pc)) {...} }` is unavoidably going to be pretty slow, because the instructions only manipulate 32-bit quantities. Getting reasonable performance would require a JIT-compiler implementation, which is complicated by the need to invalidate and recompile self-modifying code.
• Input is blocking; it is impossible to read keyboard input, which is the only kind of input, without blocking. That means that Chifir will, at any given time, be either unresponsive to the user or unable to execute any code until the user types a key.
• The instruction set has absurdly low code density, requiring 128 bits per instruction. This is a big problem for archival virtual machines because archival media are many orders of magnitude lower density than media customarily used for software interchange.

• Although the specification specifies the bit order of the image when stored as individual bits (as in their proposed physical medium), it does not specify an image file format as a series of bytes. In my implementation, I chose to represent the image in binary big-endian format, with the most significant byte first, but a different implementor might make a different choice, such as representing the image in a sequence of the ASCII digits "0" and "1". Such differences would produce unnecessary incompatibility at the file level between implementations, if not at the micro-engraved disc level.
• Chifir has no provision for access to storage media; the only I/O is the screen and keyboard.
• Because Chifir has no clock, no timer interrupts, and no keyboard interrupts, there is no way within the Chifir system to regain control from an accidentally-executed infinite loop.

However, it's worth pointing out some things Chifir got right:

• Unlike Lorie's UVC, the Chifir virtual machine has well-defined arbitrary implementation limits and overflow behavior, so it should be straightforward for all implementations to have the same arbitrary limits. This will avoid incompatibilities where a virtual machine image works on one implementation of the virtual machine (perhaps the one used by an archivist) but fails for obscure reasons on another (perhaps the one written by an archaeologist.)
• Like BF, and unlike most virtual machines, Chifir does succeed in being implementable in an afternoon, being a "fun afternoon's hack". In fact, it vastly overshoots this goal: an afternoon is three to eight hours, depending on your definition, and it took me 32 minutes to implement Chifir without a display and an hour to implement Chifir with a display. (The COCOMO model used by David A. Wheeler's 'SLOCCount' instead estimates it would take a week, but it's usually wrong in such a way for such small programs.) Chifir could be about three times more complicated without overshooting the one-afternoon budget, especially if the implementor has known-good test programs to work with instead of having to concurrently write and debug their own. Maybe a complexity budget of 512 lines of C, a bit under eight pages, would be a reasonable absolute maximum.
• Chifir's CPU lacks many edge-case-rich features present in other CPUs, which are common compatibility stumbling blocks. For example, it doesn't have floating point, flag bits, signed multiplication and division, different memory access modes, different instruction formats, different addressing modes, for that matter any general-purpose registers, memory protection, virtual memory, interrupts, different memory spaces (as in modified Harvard architectures like the Hack CPU). Similarly, its I/O specifications are ruthlessly simple, leaving little room for compatibility problems. It's very likely that a straightforward implementation of the specification will be, if not absolutely right, at least almost right.
• In part because Chifir lacks such corner cases, nothing stops you from writing a multithreaded program or JIT compiler in Chifir code.

† My implementation of Chifir is available via `git clone http://canonical.org/~kragen/sw/dev3`. It requires the following files:

http://canonical.org/~kragen/sw/dev3/Makefile

http://canonical.org/~kragen/sw/dev3/chifir.c
http://canonical.org/~kragen/sw/dev3/chifir.h
http://canonical.org/~kragen/sw/dev3/chifir_xshmu.c
http://canonical.org/~kragen/sw/dev3/xshmu.c
http://canonical.org/~kragen/sw/dev3/xshmu.h

I'm not counting xshmu.c and xshmu.h as part of the Chifir implementation because I wrote them separately.

Some approaches to a possibly viable archival virtual machine design

How should we design an archival virtual machine to make it easy to write correct and sufficiently fast emulators for existing systems we want to preserve?

Use a register-based instruction set

Register virtual machines are faster than stack machines or belt machines (like the Mill) when all three are implemented in a straightforward way; typically the benefit is about a factor of 2. Register virtual machines have a small instruction density penalty, and their assembly code doesn't factor as nicely as stack machines' and isn't as easy to generate as stack machines'. But I find it easier to write by hand, and in any case, reasonable register allocation can be fairly simple.

Register virtual machines can also, in theory, eliminate the need for jump instructions if the PC is as accessible as any other register, though the overhead on this mechanism is a bit high to use it as a calling mechanism as well.

Look to Thumb (and the curiously similar Cray-1 instruction set) for code density tricks.

Other things we've considered include: MOV machines ("transport-triggered architectures") like my "dontmove"; stack machines like the MuP21; and RTL designs where the only operation was producing new bitvectors from the NAND of slices of existing bitvectors.

Focus the instruction set design on video with SIMD instructions

The most performance-critical part of computers with graphical displays has always been the graphical display. This was true in 1963 when Ivan Sutherland wrote SKETCHPAD; it was true in 1974 when the Alto implemented bitblt in microcode; it was true in 1980 when 6502-based microcomputers competed to provide more hardware sprites on the display at once; it was true in 1996 when Andy Glew designed MMX for the Pentium MMX; it's true today in 2018 when we're using GPUs instead of CPUs to mine Ethereum, crack passwords, and train neural nets.

If updating the screen is slow, everything will feel slow. If updating the screen is fast, probably the other parts of the system's code will be fast enough too.

So probably wide SIMD registers, like SSE, NEON, or AVX, are a good model to follow. Unlike SSE (but like GPUs) it's probably better to have *only* wide SIMD registers. You can always ignore the rest of the vector if you only care about a scalar operation!

GCC offers a particularly simple and reasonably effective way to apply such facilities, documented in its manual in the section "Using Vector Instructions through Built-in Functions". Briefly, it provides only "vertical" operations (those that operate on corresponding vector items), except that you can subscript vector values to fetch individual items, you can broadcast scalars across vectors, and there's a `__builtin_shuffle` function which can rearrange vector items. And of course each operation can operate on any vector type. (Heterogeneous operations, e.g. 4 floats with 4 uint32s, are not supported.) This is mupch simpler than the irregular zoo of 600+ vector instructions provided by Intel, and if it's at times somewhat less efficient, it's still dramatically more efficient than just writing scalar code.

(In C++, GCC also offers `?:`, `!`, `&&`, and `||` for these vector values.)

SIMD instructions provide the opportunity for a naïve for-switch loop to get reasonable performance, because each instruction is doing enough work to pay for the heavy interpretation overhead. This can also helps with the cost of bounds-checking, at least if you don't have scatter-gather — a bounds-check every 16 or 32 bytes is less costly than a bounds-check every byte.

Originally I thought you could get by with only 32-bit unsigned math, like Chifir, and that might actually be true. But then I got a tiny bit of experience writing SSE code, and it sure is nice when you can get 16 byte additions in an instruction instead of 4.

(However, I don't think the saturating arithmetic mode is worth it.)

Or use vector instructions as an alternative to SIMD

"Vector instructions" here is referring to Cray-style or NEC-SX-style vector instructions. I don't think this is a good idea for archival because it's far too easy for implementation optimizations to leak through to the visible semantics, but see A simple virtual machine for vector math? (p. 986) for more details.

Forget about floating point

Floating-point math is profoundly important and very tempting to support, but even with near-universal adherence to the IEEE-754 standard, which requires that the basic arithmetic operations produce results accurate to within half an ulp (which means that all conforming implementations will produce bit-identical results), it's impossible to argue that floating point is easy to get right. Even in the last few years, the introduction of the FMA facility in Intel processors has introduced divergence from bit-identical results in expressions as simple as `a+b*c`, which may produce results differing by an ulp depending on whether your compiler compiles it to use FMA or not.

Aside from that, getting gradual underflow, rounding modes, and NaN handling right requires lots of special cases, and it is often omitted for the sake of performance.

Don't have unspecified behavior

This means no undefined instructions, no undefined video results, no undefined results of out-of-bounds memory, no race conditions, nothing.

Have nonblocking I/O

It's absolutely essential to be able to run code while periodically checking whether the user has provided any input. It may not be necessary to make the input interrupt-driven — modern machines are fast enough to poll a keyboard at 100Hz without producing undue CPU load, even in emulation. A "hardware" input event ring buffer is probably the right mechanism here.

Have a timer interrupt

Without a timer interrupt, you can't do preemptive multitasking, and without either a timer interrupt or a pollable system clock, you can't do animation at a correct speed, because you have no way to tell what speed you're doing it at. The timer interrupt should be able to run at a high enough frequency that you can poll the keyboard at over 100Hz. A fixed 1000Hz is probably fine, but it's possible that it may need to be configurable by the program, like the Unix alarm(2) system call or JS's `setTimeout` and `setInterval`.

This thought is scary because it inevitably introduces nondeterminism into the system, because it potentially provides side-channel communication into the virtual machine, and because bugs in interrupt handlers are hard to debug. But the cost of not having it is just too high.

(With respect to nondeterminism — consider possible implementations in which timer interrupts are handled at the end of each instruction, at the end of each basic block, and at backward jumps or calls. How likely is it that a program that is tested and working in one of the later cases will turn out to have previously undetected heisenbugs when run on a simulator with more possible interruption points?)

For non-interactive tasks or those in which determinism is paramount, this facility can be disabled.

As an alternative sufficient for animation timing, a purely deterministic facility could freeze the virtual machine's execution until a given timestamp had passed, but provide no way for the program to determine the current timestamp. (Waiting for a time in the past would simply return immediately.) Since the program cannot execute any instructions while frozen, it cannot observe whether any time has passed or not. However, this doesn't enable you to implement a preemptively multitasking operating system.

Write a comprehensive set of test cases

You don't need any testing hooks in the virtual machine definition for this, just a breakpoint facility and the ability to look at memory when a breakpoint is hit, which are entirely external to the VM as seen by the program — they are a *deus ex machina* from the program's point of view.

For math operations with up to 32 bits of input, such as 16-bit binary operations and 32-bit unary operations, there's no reason not to exhaustively test them.

Specify a file format

That means a concrete representation as a sequence of 8-bit bytes, not just glyphs engraved on a disk. Provide the test cases in this format.

Don't have memory protection

Memory protection is probably not worth the complexity it would add to the virtual machine.

Minimal set of ALU operations

BF only provides increment and decrement ALU operations, which is insufficient. Chifir provides +, -, ×, ÷ (presumably floor division), modulo, <, and NAND. C provides +, -, *, / (usually floor division), %, ?:, <, !=, ==, >, <=, >=, |, ^, &, ~, <<, >>, and unary -. Golang adds abjunction, &^, which is also in AVX2, PDP-10, and (IIRC) ARM; Java adds >>>, unsigned right shift, as a separate operation, while in C the choice between logical and arithmetic right shift is based on the declared data type. The F18A offers 2*, 2/, +, and ~ (spelled -), &, ^, and +*, which executes an 18-bit multiply if you run it 18 times. Various kinds of vector instruction sets (for example, SSE and APL) also offer min and max; many languages support exponentiation; and lerp, a sort of continuous generalization of ?:, is often offered as a fundamental operation. I saw somewhere an instruction set whose > operation was actually unary; it was just `x > 0 ? x : 0`, so applying it to the result of a subtraction would give you a nonzero value precisely when the result had been positive. CPU instruction sets traditionally have rotation instructions as well as shift instructions, but I find them useful very rarely.

You also have really out-there stuff like CLMUL, AES, CRC32, and the like, the abominable children of the dark-silicon age.

So how much do you really need? For an archival virtual machine, probably the F18A's seven operations (three of them unary) is too minimal, since it means that (in the absence of a sufficiently smart compiler, which is not an afternoon hack) a CPU or FPGA with hardware multipliers must emulate the very common operation of multiplication through an arduous sequence of simulated instructions, and similarly for barrel shifters and multi-bit shifts. On the other hand, the cost of emulating > given < is generally nil, and the cost of emulating >= given < is small. Where it comes to bitwise operations, abjunction (with constants) or NAND alone is sufficient to produce all the others, but producing XOR with either of these alone requires five operations, which seems excessive (thus the F18A's three bitwise operations).

I suggest the following set:

- Unsigned binary integer +.
- Two's-complement unary integer -.
- Unsigned integer *, producing a double-width result.
- Unsigned integer divmod, taking a double-width dividend but producing two single-width results.
- Unsigned integer << and >>> with arbitrary shifts.
- Signed integer >>.
- Bitwise &, ^, |, and ~. (See Performance properties of sets of bitwise operations (p. 636) for the exploration of this.)
- Two's-complement signed integer max, i.e. λ(x, y).(x > y ? x : y).
- ==.
- The ternary operator ?:, which is particularly important with vector registers, since you can't implement it the traditional way using control flow.

This is 14 operations, considerably more than the F18A's 7 but

substantially less than C's 19; they are chosen with the intention that the missing 7 operations, plus many other useful ones, can be constructed in two operations:

- `a - b` = `a + -b`
- `a > b` = `max(0, a - b)`
- `a < b` = `max(0, b - a)`
- `a >= b` = `0 == max(0, b - a)`
- `a <= b` = `0 == max(0, a - b)`
- `a &^ b` = `a & ~b` (or `a ^ (a & b)`)
- `a NAND b` = `~(a & b)`
- `a NOR b` = `~(a | b)`
- `a != b` = `0 == (a == b)`

It's arguable that maybe we shouldn't include both ~ and unary -, since you can get from one to the other with a simple +1 or -1.

Topics

3-D printing by flux deposition

Kragen Javier Sitaker, 2017-02-24 (updated 2019-07-27) (21 minutes)

We've been doing some experiments in 3-D printing of glass parts. It's a classic powder-bed binder-deposition process, except that the binder is another solid powder, and the postprocessing is different — rather than removing a fragile greenware part from the powder bed and then possibly baking, curing, or soaking it, we fire the whole powder bed to activate the binder, which acts as a flux to sinter or even dissolve the surfaces of the powder grains; the binder itself may or may not melt.

We've had success so far with a quartz powder bed and fluxes of soda-lime glass frit and a commercial pottery glaze, both baked at 1020°, in air, at atmospheric pressure. We also expect success with fluxes of potassic feldspar and lead oxide. In all of these cases, the final product is either a glass or a glass with a quartz filler.

However, this process is much more widely applicable.

Unique advantages

The flux-deposition process, unlike most binder-deposition powder-bed processes, involves no extra step of handling delicate greenware extracted from the powder bed. This extra step is labor-intensive and often involves extra equipment such as sandblasting cabinets to remove extra unused powder without damaging the greenware part.

Aside from the reduction in human effort, this should improve the resolution. Typical binder-deposition processes as offered by service bureaus like Shapeways cannot reliably produce walls and wires under about 1000 microns in thickness. By contrast, flux deposition powder bed printing should offer very high resolution — 3D Systems's direct-metal selective laser melting machines can, as of 2015, produce parts with 100-micron wall thickness because they, too, have no greenware-handling step, so the articles are already at their full strength when they are removed from the powder. We think flux-deposition printing can offer the same or better resolution, but without the long build times and inert atmospheres needed for direct metal SLM or SLS.

While we do not expect flux deposition to be able to avoid the issue of dimensional change during sintering, which is a plague for sintering manufacturing processes in general, there is good reason to expect the distortion introduced by the process to be very small compared to fused deposition modeling, selective laser melting, selective laser sintering, or machining of green ceramics before firing them. The binder is deposited when the powder bed is at room temperature, so the average expected final expansion is zero; the article is supported by unused binder throughout the process, limiting forces that could produce deformations; and the process need not involve any strong thermal gradients.

Immediately after sintering, the boundaries between the original powder grains will be enriched in the flux, while their cores will contain none of it; but if the flux and the bulk material are miscible in solid solution as well as liquid, then diffusion annealing can remove

these inhomogeneities. This process can allow the article thus produced to withstand higher temperatures than those used to make it in the first place.

Aside from the microscale inhomogeneities stemming from the union of different materials in powder form, it's possible to use this process to create macroscale inhomogeneities; for example, by smoothly varying the concentration of lead oxide in a glass article, it should be possible to fabricate an article whose refractive index is not constant, but has some controllable gradient. If the microscale inhomogeneities can be eliminated, such graded-index optics can constitute monolithic refractive optical systems that approach diffraction-limited performance arbitrarily closely, lacking as they will refractive index discontinuities to reflect stray light.

Steel and iron alloys

Iron melts at 1538°; due to alloying with 0.18% of carbon, AISI 1018 steel, the most common steel, melts around 1520°. AISI 1070 steel is popular for harder goods such as knives; it contains 0.70% carbon and melts at 1479°. The iron-carbon phase diagram shows a eutectic point at 4.3% carbon at about 1175°, and some cast irons approach that content of carbon, typically replacing some with silicon.

This process is already used industrially to produce sintered steels; for example, Höganäs offers a range of "pure iron powders" for this purpose, which all range from 20–200 μm in particle size, with little variation in that dimension.

As of 1997, iron-powder metallurgy normally takes place at 1120°–1150° for 20'–30' because this allow mesh-belt furnaces to survive, although some processes supposedly may require a sintering temperature between 1100° and 1300°. Even 1120° is enough to sinter plain iron. Carbon is added as graphite; endogas avoids removing the carbon, though we think mere carbon dioxide should be adequate.

Apparently it is common to include up to 2.5% powdered copper in the iron in order to fill pores and produce precipitation-hardenable alloys. So copper is a possible alternative to carbon for this purpose; copper and carbon are usually used together to avoid thermal expansion induced by the copper.

The biggest difference with the process being considered here is that iron powder is normally compacted into a mold at 200 to 800 megapascals before being sintered.

Iron phosphide, though hazardous to handle, is another alternative flux which is known to form a eutectic with iron at 10% phosphorus that melts at only 1050°. The resulting iron-phosphorus alloy is similar to a steel, with final phosphorus concentrations of up to 0.6% being beneficial. (Phosphorus is considered a contaminant in normal steelmaking for reasons that are not relevant here.)

Aluminum is probably not a good bet; it forms no eutectic with iron, its miscibility is low, and the intermetallics are brittle and weak, although there has been promising work by Hansoo Kim in Korea to fix this with manganese, nickel, and carbon.

This makes us optimistic that even a very small amount of carbon, copper, or iron phosphide could work effectively to flux a powder bed of pure iron heated to somewhere below 1120° so as not to sinter the iron itself. If the carbon particles are small enough relative to the

firing time, they will dissolve entirely into the iron particles; further diffusion while hot should gradually eliminate the lowest-melting-point liquid regions. Depending on the size of the iron particles, the result should be either a pure steel article, or an article composed of grains of pure iron cemented together with steel or cast iron, in a bed of loose iron powder.

We don't know if there is a way to get the articles to be fully dense; this would require that the steel or cast iron have greater volume than the iron and carbon that went into them. This seems unlikely, as steels are usually slightly denser than iron.

Korol'kov & Kibak report that iron sinters more easily with 100–250 ppm boron; they mention that the eutectic liquid phase Fe_2B + Fe melts at 1160° to 1180°, and they sintered at 1160°. Also, their 100ppm boron-doped pressed iron powder began to shrink at 450°–500°, suggesting that sintering was already beginning, and hit a discontinuity at about 900°. They suggest that this initial shrinkage was due to cementation of iron grains with liquid oxides of boron.

Aluminum

Aluminum natively melts at 660°, but for casting, it is normally alloyed with silicon, with which forms a eutectic at 12.2% Si, melting at 577°. Silicon itself melts at 1414°. Aluminum-alloy powders are commonly sintered at 590–620°, which is hotter than the melting point of the eutectic. It seems that it should thus be possible to flux an aluminum powder bed with small amounts of silicon, bake the result somewhere between 500° and 590°, and get good "cast" aluminum articles out without sintering the aluminum powder.

Sapphire

Sapphire, Al_2O_3, also known as corundum, aluminum oxide, or ruby, doesn't melt until 2044° or so. It's an extremely hard and refractory material widely used for engineering ceramics as well as bulk refractories.

In 1957, Cutler, Bradshaw, Christensen, and Hyatt showed that the addition of about 4% (wt%) of appropriate fluxes could lower the sintering temperature of finely divided sapphire to 1300° to 1400°. Specifically, they added oxides of manganese, titanium, and copper as fluxes. 2% of either MnO or Cu_2O and 2% of TiO_2 were successful at getting very nearly full sintering to 3.8 g/cc at 1300°. Furthermore, they reported "bleeding" of a black liquid from their pressed pellet specimens onto the coarse sand below, suggesting that the mixture didn't merely sinter — it partly melted. This suggests that sintering should be possible at even lower temperatures, although they did not attempt this.

Indeed, Xue and Chen successfully sintered alumina at 1070° in 1991 with 0.9% CuO (mol%), 0.9% TiO_2, 0.1% B_2O_3, and 0.1% MgO. Full sintering took an hour at 1070°, but only about 15' at 1200°; their result had even higher fracture toughness than undoped alumina. (They also mention as an aside that 1500° to 1700° is adequate to sinter high-purity alumina, and that there's a known 1096° eutectic between Al_2O_3, CuO, and Cu_2O.)

A perhaps even more effective flux for sapphire may be the $MgCl_2$-NaCl mixture in US patent 7,988,763, although this is a flux in the metallurgical sense — it fulfills various functions, including

scavenging contaminants from the metal and preventing oxidation of the molten metal, but also including liquefying metal oxides. The patent mentions that $MgCl_2$ mixes with other salts are already in use to keep the melting point of the flux in the 400–500° range. Unfortunately, the patent doesn't quantify how low it reduces the melting point of the aluminum oxide; it does mention that the mixture was prepared at 550° in an alumina crucible which it apparently did not dissolve in 45 minutes, and that variants of it were used to purify molten aluminum at 850°, 720°, and 700°, but it does not affirm that it liquefied the sapphire dross at these temperatures.

NaCl boils at 1413°, so using it in a pottery kiln salt-glazes both the pottery and the kiln and releases chlorine gas. And that would also be true, and very objectionable, if you heated up sapphire to its normal sintering temperature of 1500–1700°. But if it's possible to use small amounts of these salts to flux alumina at temperatures more like 400° to 900°, no significant amount of salt should boil off.

By far the most famous flux for sapphire, however, is cryolite (Na_3AlF_6, melting point 1009°) with an excess of aluminum fluoride, which is used in the Hall-Héroult process to dissolve sapphire at 950–980° so that aluminum can be electrolytically produced from it. The binary eutectic melts at 962.7° and contains about 22% alumina, and apparently there is no solid solubility between the compounds, so cryolite may not in fact be very suitable for facilitating the sintering of granular sapphire.

Viridian

Chromium(III) oxide, the rare mineral eskolaite, also known as chromia, is the surface film that makes chromed bumpers and common stainless steels stainless. It's a green pigment ("viridian" or "chrome green") and harder than quartz, though slightly softer than sapphire. It's sold for pigment use at about US$25/kg.

Viridian is super fucking refractory, not melting until 2435° or sintering until 1600° in pure form, but you can sinter it at 1280° by adding 1% TiO_2 by weight, according to Callister, Johnson, Cutler (the sapphire dude), and Ure 1979. A tricky problem is that under low pressures it tends to smelt to chromium at temperatures above 1600°, and if exposed to oxygen it tends to not sinter, so the sintering has to be done in a non-oxidizing (endogas or argon) atmosphere. The rutile also may serve to stabilize the oxidation state of the viridian.

Viridian is miscible with alumina in all proportions.

According to Nagai and Ohbayashi (1989), viridian is a P-type semiconductor, but the addition of over 2% rutile makes it N-type.

Rutile

Rutile, titanium dioxide, is commonly used as a white pigment, including in food. In fact, 70% of worldwide pigment production is synthetic rutile. It's a medium-hard mineral, in the same Mohs 6–6.5 range as orthoclase, a bit softer than quartz. It melts at 1800° and costs about US$20/kg on eBay.

Likely fluxes include oxides of iron (FeO), magnesium, and manganese; more improbable candidates include alkali halides and oxides of zinc and vanadium. The spinel group consists of solid solutions of oxides of titanium, magnesium, zinc, iron, manganese,

aluminum, chromium, and silicon crystallized in a cubic close-packed lattice, but titanium spinels are rare, naturally occurring only as ulvite, iron titanium oxide.

Cho and Biswas did find that doping rutile (actually its polymorph anatase) with 1.3% (mol%) vanadium sped up its sintering dramatically, but none of their tests were at a low enough temperature to see if the doping lowered the sintering temperature; their coolest test was at 900°. They do predict theoretically that their "V-TiO_2" should sinter at 800°, while pure TiO_2 basically won't.

PZT

PZT by far the best piezoelectric material, the best high-capacity dielectric, and the best ferroelectric material. It is normally sintered above 1250° but can be sintered at 1000° by doping it with bismuth oxide and carbonates of sodium or lithium (e.g. 0.375% wt% of lithium carbonate and "an equal mole fraction of Bi_2O_3"); this according to Cheng, Fu, and Wei, 1989. Corker, Whatmore, Ringgaard, and Wolny reported in 2000 being able to sinter it at 800° with a "sintering aid" of 5% liquid oxides of lead and copper; the eutectic mixture of PbO (litharge or massicot) and Cu_2O (cuprous oxide or cuprite), which is 80% PbO, melts around 680°, while PbO alone doesn't melt until 888°, and Cu_2O doesn't melt until 1235°. (Given PbO's well-known use as a flux and thinner for glasses and pottery glazes, I suspect this mixture would also work well as a "sintering aid" for quartz, and might result in a transparent article, though Cu_2O alone is so red it's commonly used as a pigment, and might precipitate out as lead silicate forms.)

Corker et al. also briefly survey the overall use of such sintering-temperature-reducing additives in ceramics, typically metal oxides, citing a pair of papers by Kingery in 1959.

Table salt

While probably not useful for any practical use, sodium and potassium chlorides may provide an inexpensive way to test and debug apparatus. The NaCl-KCl system has a eutectic point at about 670° at about 50% KCl; NaCl by itself melts at 801°, while KCl by itself melts at about 770°.

This is not an ideal test system from the point of view of the relative concentrations, the relatively low freezing point depression of the eutectic, the hygroscopic and corrosive nature of the powder, and the purely ionic nature of the bonding in the powder; however, we can easily build a "furnace" that reaches 700°, and the two feedstocks are very cheap, nontoxic, and readily available. Potassium chloride costs US$25 for 10 pounds on eBay (US$5.50/kg); on Mercadolibre sodium chloride costs AR$285 (US$18) for 50kg (US$0.36/kg).

According to US Patent 7,988,763, the ternary $MgCl_2$-KCl-NaCl system has a eutectic point of only 383°, where all three components are equal, and a binary eutectic between $MgCl_2$ and NaCl at 45% wt% NaCl melting at 439°. Magnesium chloride, which melts at 714°, is even easier to obtain than KCl, and on Mercadolibre it costs AR$400 (US$25) for 5 kg (US$5/kg).

Titanium

Amalgams

Applying small amounts of mercury to many different metals can produce stable amalgams.

Brass and bronze

Could you use a powder of some other metal to flux copper powder?

Pure copper, though somewhat expensive, is easily obtained in the modern economy (or from electrical scrap) and does not melt until 1084°; sometimes it occurs naturally allied with arsenic ("bronze"), which both fluxes it so that it can melt at much lower temperatures and hardens it greatly. The humans started using natural bronze some 5300 years ago and synthetic bronze made by mixing separately-occurring tin, rather than arsenic, with copper about 6500 years ago, at first in Thailand and later in the Middle East. Other elements are also widely alloyed with copper, notably aluminum ("aluminum bronze"), zinc ("brass"), zinc and nickel ("nickel silver"), and silicon ("silicon bronze"), with much the same results.

Beryllium forms a precipitation-hardenable alloy with copper that can be harder and stronger than most steels (an ultimate tensile strength of some 1150 MPa at 2% Be), but is little used because of the toxicity of beryllium and beryllia; I do not think it lowers copper's melting point much in the tiny quantities (0.2%–3%) required to make precipitation-hardenable beryllium copper. XXX yes it fucking does

According to 2001 edition of the ASM Specialty Handbook *Copper and Copper Alloys*, the solubility limits of these elements in copper at 20° are 6.5% for arsenic, 1.2% for tin, 9.4% for aluminum, 30% for zinc, 0.2% for beryllium, and 2% for silicon, all by weight; it's miscible in all proportions with gold, nickel, and platinum.

Arsenic is somewhat dangerous, vaporizing at 615° and producing deadly arsenous oxide gas; this may be a significant reason for the switch from arsenical bronze to tin bronze in the early Bronze Age.

Typical bronze is around 12% tin (the rest being copper) and melts around 950°, while tin itself melts at 232°, but remains liquid until 2600°. The tin–copper system is not very friendly for this kind of thing; its eutectic composition is I think Indalloy 244, melting at 227°, which is 0.7% copper, with the rest being tin; the copper is present as a small amount of the brittle intermetallic Cu_6Sn_5 rather than forming an alloy. This suggests that you'd really just be kind of just soldering the copper particles together with the tin, rather than dissolving them fully to make a homogeneous alloy.

Brass — copper fluxed with zinc — melts at 900° to 940°. Zinc melts at 420° and boils at 907°, producing a serious danger of zinc oxide fumes — mixed with air, the zinc vapor burns immediately, and the oxide solidifies as particles with a diameter of around a nanometer, since the oxide is solid until 1975°. So Roman brass was made by the "cementation" process of heating calamine (zinc carbonate or silicate) with copper and charcoal (presumably to scavenge the oxygen), rather than from metallic zinc; or by heating zinc oxide together with copper. This works by producing zinc vapor directly from the calamine, which is taken up by the solid copper, but the reaction is slow; it typically takes several hours. Roman brass is typically 20% to 28% zinc, but some modern brasses have even higher levels, up to 40%.

So, again, the likely outcome of trying to flux copper powder with zinc metal would be just sort of soldering the copper particles together with zinc.

Brasses and bronzes increase in strength as the degree of copper is reduced; when 60% cold-reduced, 8%-tin bronze has an ultimate tensile strength of some 750 MPa, and 30%-zinc brass is nearly as strong at 600 MPa, compared to some 400 MPa for pure copper or 225 MPa for pure annealed copper. But the objects produced by this flux-deposition process would be fully annealed, so their strength would be more like 350 to 400 MPa even at those high alloying levels

Topics

Debokehfication

Kragen Javier Sitaker, 2019-09-01 (updated 2019-09-12) (4 minutes)

Bokeh preserves sharp edges; it just spreads them out. The circularly-symmetric boxcar filter of an ideal bokeh has a circularly-symmetric sinc frequency response in two-dimensional frequency space, and sinc's falloff is pretty slow, just 6 dB per octave. So even a 32-pixel-wide bokeh is only attenuating single-pixel detail by about 30 dB.

(Of course, boosting single-pixel noise by 30 dB will add nontrivial graininess; that's 5 bits of precision lost, after all. But some photos have that degree of precision.)

Indeed, as suggested in Starfield servo (p. 1709), under some circumstances such a bokeh can actually *enhance* sensor precision, and it might be preferable to use a large pinhole in front of a sensor rather than a lens on a camera, both because it gives you a more precise reading on the position of bright impulses in the visual field, and because it does a better job of taking advantage of the limited dynamic range of common image sensors. The principle is the same as how R's default graphic for plotted points is a circle, not just a point, and in fact a disc in the center of the pinhole would probably work even better for extending the sensor dynamic range and precision. A more elaborate shadow mask such as a Hadamard matrix could improve this further.

Real camera bokehs tend to not be perfectly flat, even at small scales; although they don't include Hadamard-matrix shadow masks, in addition to spherical aberrations, they do include tiny imperfections on the lens and lens filters that only add a tiny amount of stray light to a focused image, but are easily visible in unfocused images of (near) point source lights.

N.B.: This effect might be useful for getting lensless optical transmission microscopy out of commonplace digital cameras without taking the lenses off of them: put the slide reasonably close to the camera, illuminate with one or more out-of-focus point sources, ideally with some non-overlapping or mostly non-overlapping images on the focal plane.

These imperfections provide additional high-frequency information that could permit improved estimates of the unblurred image; moreover, in images that contain at least some bright points, they can provide much tighter estimates of the defocus of a particular region of the image. Also, if they are sufficiently strong, they can disambiguate behind-focal-plane defocus from in-front-of-focal-plane defocus. (Any kind of half-turn asymmetry in the bokeh can provide such disambiguation, including the common feature of approximate polygonality with an odd number of sides.)

Aside from the obvious approach of removing bokeh by applying Wiener filters selectively to parts of the image, it might be worthwhile to try not only convolution with an estimated bokeh shape but also morphological erosion with it, to identify candidate bright points — both to improve the estimate of bokeh shape and to measure the scale of the bokeh in different parts of the image.

Image-processing tricks on the bokeh are not limited to removing it and doing microscopy with it; you can also do camera

identification (from the lens imperfections) and depth estimation. You might be able to correct chromatic aberration.

Bokeh can vary over the focal plane due to, for example, occlusion from a shroud extending in front of the lens. If this is recognizable it should be relatively easy to correct, but more general occlusion effects cannot be corrected in general — you might have a single light that's half-covered by a finger halfway between the camera and the light, giving a sort of partially-eclipsed-moon bokeh not shared with much else in the scene.

Topics

- Algorithms (p. 3310) (123 notes)
- Graphics (p. 3483) (91 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Optics (p. 3609) (34 notes)
- Microscopy (p. 3583) (3 notes)
- Bokeh (p. 3346) (3 notes)

Dercuano backlinks

Kragen Javier Sitaker, 2019-05-22 (7 minutes)

I want to add backlinks to Dercuano articles.

Backlinks

Backlinks are the Wiki feature which shows you all the pages that link to a page; they make links in some sense bidirectional, and they allowed us to categorize pages on Wiki years before Wikipedia — you would just create a page called CategoryWhatever and then link to it from whatever pages you thought should belong to it, and look at its backlinks page.

I think having backlinks will improve Dercuano's navigability dramatically.

Desired implementation in Dercuano

In Dercuano I'd like to display the backlinks on the same page as the rest of the article, maybe at the end, and ideally with a bit of context. This is especially helpful when I have a later note that essentially obsoletes an earlier one.

Implementation techniques

Dercuano goes to some amount of trouble to avoid rerendering notes unnecessarily, because rerendering all the notes takes 28 seconds on my laptop, while just regenerating the categories and the index page takes only 6.3 seconds. A more detailed timing breakdown (as of dercuano-20190522):

	sec
rerendering all the notes	28
just regenerating categories and index	6.3
instantiating Bundle and all Notes	0.5
reading `triples`	0.065
bundle.generate_index()	2.0
bundle.generate_categories()	2.9

Actually about a quarter of this time to generate categories and indices is finding note titles, and another 10% is finding the notes in a category; most of the rest is generating HTML, though some of it is querying the unnecessarily inefficient triple store, which is also a factor in the note titles being slow. But 77% of the execution time is rendering notes, which is mostly a matter of running Markdown, though it actually spends about 25% of its time finding the notes in a category too.

So Dercuano is a lot more usable when it doesn't rerender notes when it doesn't have to, although it produces incorrect results when it fails to rerender notes whose titles and categories have changed. The trouble is, the point at which we learn about the backlinks is when we render the bodies of the notes. (At present, repl() on line 469 inside `replace_links` is getting called 3095 times, some of which it finds a backlink.)

One of the things I'd like to do is do the actual rendering in

subprocesses, so that I can run four or so of them; ideally, this should speed up the 22 seconds of rendering to 5.5 seconds, although lazy memoization may result in some duplication of work between the subprocesses.

Without caching, the minimal-work way of handling the unknown-backlinks problem would be to do it in two phases: one phase where we render all the bodies, and a second phase where we render all the backlinks. However, this adds complication if we want to farm the rendering out to subprocesses, since they would then need to deliver both the backlinks and the bodies back to the main process, probably via the firesystem.

A different approach would be to store the relevant backlinks in the filesystem somehow, in the output directory probably, and rerender the page when its backlinks change. Most of the time, they wouldn't change even when some other note that links to it changes, but when they do, this could require you to render the same note twice in the same run. In particular this results in rendering nearly all the notes twice when starting from a missing output directory, since at first each note is rendered with no backlinks, and then later it will be rendered with backlinks.

…this suggests I should optimize the triple store a bit and profile my execution times again. So I did. Upon adding code to index the triple store on startup, the Bundle instantiation time ballooned from 0.50 s to 0.55 s; upon using it, the build-from-scratch time went down from 28 s to 18 s, and the just-rebuild-index-and-categories time went from 6.3 s to 2.5 s. Now the top time consumers are all Markdown rendering functions, except for 650 ms waiting for tar to finish.

This in some sense makes the case for avoiding and parallelizing Markdown rerendering more urgent than before: previously Markdown took 75% of the runtime, and now it takes 85%.

So, my thought is to do the following:

• At program startup, read all the backlinks.
• For each note:

• Generate a glob of metadata for a note from the triple store (for example, the titles and note counts of categories) and the backlinks.
• Compare the glob of metadata to the previous glob of metadata for that note, stored on disk. If it's the same and the note source file hasn't changed, continue to the next note.
• Render the note and generate backlinks.
• Store the glob of metadata and the backlinks.
• Update the backlinks store and do #2 a second time. A second time should be sufficient to reach a fixed point. This should be done automatically, rather than requiring the user to do it as L^AT_EX does.

This requires separate stores for the glob of inputs for a note and the glob of outputs (including backlinks). I'm thinking this can be in places like .meta/cheap-frequency-detection.inputs and .meta/cheap-frequency-detection.outputs for the note notes/cheap-frequency-detection.html.

The code path actually needs to be a little bit more complicated, because the set of other notes' titles that the note HTML depends on is a function of the note source file's contents, and finding out what

that set is presently would require us to rerender the Markdown, which is what we're trying to avoid. But we don't actually need to go that far — if the source file has changed, we need to rerender it anyway.

We could imagine storing the titles of notes parsed from the HTML in the same store as the backlinks (thus the filename .output rather than .backlinks) in order to avoid opening and parsing the same files multiple times.

This is of course very much the kind of thing discussed in A minimal dependency processing system (p. 911), Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870), Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257), and Dehydrating processes and other interaction models (p. 3208), and to a lesser extent Simple dependencies in software (p. 2447). I haven't written any of the systems described therein, so I need to do it by hand.

Topics

- Performance (p. 3621) (149 notes)
- Caching (p. 3361) (25 notes)
- Dercuano (p. 3406) (16 notes)
- Hypertext (p. 3512) (13 notes)
- Dependencies (p. 3405) (7 notes)

Reducing nighttime bedroom CO_2 levels

Kragen Javier Sitaker, 2019-07-08 (updated 2019-07-09) (14 minutes)

My girlfriend's bedroom is cold in winter, so we sleep with the door closed and the electric heater turned on, which pretty much eliminates air circulation. But sometimes we wake up out of breath, perhaps because so much CO_2 has accumulated in the air. Recent research suggests that this is a problem for a lot of modern sleeping arrangements, but it doesn't usually rise to levels that cause perceptible discomfort. This note explores the possibilities for solving this problem.

(See Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) for more information on the air changes per hour needed for clean air.)

The CO_2 level can get really fricking high without ventilation

The bedroom is 3 m × 3 m × 4 m, and pretty much at sea level, where the density of air is 1.2 g/ℓ (at STP, according to Millikiln (p. 2581)), so it contains about 43 kg of air. The window seals pretty well. We are two people, and we commonly spend 12 hours straight in there; at 2000 kcal/day/person that's 4000 kcal/day or 2000 kcal without opening the door.

Carbohydrates, our normal energy source, are about 5 kcal/g (21 MJ/kg), and carbohydrates all have the empirical formula CH_2O, thus the name. (Glucose, for example, is $C_6H_{12}O_6$.) Multiplying by the atomic weights, that's 12 parts carbon, 2 parts hydrogen, and 16 parts oxygen, so carbohydrate is 12/30 carbon by mass. (Protein is similar; fat is about twice as high in carbon and also about twice as high in energy, so the result is approximately the same.) So that's about 12.5 kcal per gram of carbon (52 MJ/kg). CO_2 is 12 parts carbon to 32 parts oxygen, so that's about 3.4 kcal per gram of CO_2 (14.3 MJ/kg).

So 2000 kcal is 590 g of CO_2, which would amount to 14000 ppm by mass, 1.4% of the air, 35 times the concentration in the outside air. No wonder we get uncomfortable!

If we wanted to use ventilation to limit the CO_2 to only 2½ times the 400 ppm in the outside air, we'd need to have only an extra 600 ppm in the air, or 26 g CO_2, or 370 kJ, or, in archaic units, 88 kcal, which our two human bodies emit every half-hour. This would require roughly two or three air changes per hour.

You can suck it up with succulents, if your house doesn't suck

If you put houseplants in your bedroom to keep the CO_2 levels down when you sleep, you may be disappointed if you are one of those people who sleep at night, rather than during the day. Plants convert CO_2 to oxygen (and sugar) through photosynthesis, which

only happens when the sun is shining on them. At night, plants oxidize sugar to survive, just like the humans do.

Plants using crassulacean acid metabolism are commonly recommended to help with this; these are typically succulent desert plants. Normal land plants keep their pores ("stomata", literally "mouths") open during the day, allowing water to evaporate from their leaves and keeping the plants cool. CAM plants instead tolerate the high temperatures, keeping their stomata tightly closed during the day. But, like normal plants, they need sunlight and CO_2 for photosynthesis. So the motherfuckers open their stomata *at night*, releasing the excess oxygen from photosynthesis, and store the CO_2 as malic acid until daylight comes around.

Cacti, aloe, and pineapples are among the crassulacean acid metabolism plants, although none of those are actually Crassulacea. Because most CAM plants are specialized to very dry climates, they grow very slowly, prioritizing surviving over thriving. This means, I suspect, that they tend to photosynthesize very slowly. Aloe vera might be the best choice of CAM plant for this kind of thing.

The amount of CO_2 consumed is dependent on the amount of sunlight and the photosynthetic efficiency of the plant. Typical photosynthetic efficiency is 3%–6% at 114 kcal per mole of CO_2, which is to say, it instead takes 1900–3800 kcal of sunlight (8–16 MJ in SI units) falling on typical plants to reduce a mole of CO_2 (44 g).

So how much aloe do you need, or how much sunlight do your aloes need to soak up during the day, in order to suck up 590 g of CO_2 per night? Well, 590 g · 12 MJ/44 g = 161 MJ, and divided by a day, that's 1900 W. But how much sunlight is that?

Let's figure that the "capacity factor" of photosynthesis is close to the capacity factor for utility-scale solar photovoltaic energy in the US, which is about 25% — that is, for every 100 watts peak of installed solar photovoltaic generation capacity, you get on average 25 watts, due to things like the sun being on the wrong side of the Earth half the time, and clouds, and a suboptimal sunlight angle on your panels. Or leaves.

The "solar constant" against which those theoretical numbers are calibrated is 1000 W/m², which is about how much sunlight gets to the bottom of Earth's atmosphere where the humans live. At a 25% capacity factor, this is 250 W/m². So you need almost 8 square meters of aloe plants. (And 8 square meters of sunlight to put them in. You might want to put them in some Radio-Flyer-style wagons so you can trundle them out onto the patio in the daytime.)

So basically this approach can work, but it requires on the order of 4 m² of sunlit leaves per person, so you kind of have to design your house around it. I can't do it here in my apartment (not enough sunlit area) and neither can she.

If we had that much sunlight available, we might be able to store much of its energy thermally (see Household thermal stores (p. 1533)) which would reduce our desire to keep the door closed.

Scrubbers

I've written previously in Notes on a possible household air filter (p. 1961) and House scrubber (p. 248) about mechanical means of removing CO_2 from the air in your house; I will not go into more details here except to mention that it does seem feasible, and could be

retrofitted to existing sun-poor housing.

A heat exchanger — a nose for the house

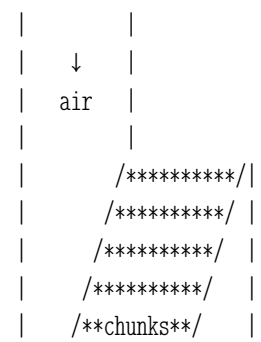
As David MacKay (RIP) pointed out in Without Hot Air, Section III, chapter E, the humans' nose is a heat exchanger which reduces heat and humidity losses to your breath, and they can install a countercurrent heat exchanger on their houses, with the same purpose of transporting air in and out without transporting the precious heat along with it. (Or, in the summer, the odious heat.)

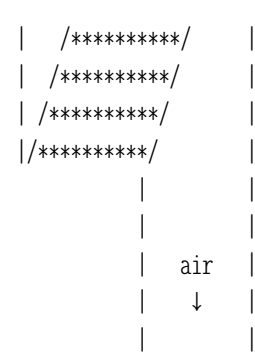
Countercurrent heat exchangers ("recuperators") have a couple of major differences from the "regenerator" design used in Stirling engines and the noses of the humans. One is that, for the same efficiency, they must be much larger, because the two working fluids in a recuperator are in much less intimate contact than the fluid and reservoir in a regenerator. (However, see Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) for a design that fixes this.) The other is that regenerators transfer not only heat but also material between the two directions of flow. For example, water condenses in your nose as you exhale, then evaporates again as you inhale. For some applications, such as your nose, this mass transport is essential; for others, such as transferring heat from a sodium coolant loop to a steam coolant loop, mass transport would be fatal. (Regenerators also have a third difference, which is that their flow is unidirectional and alternating rather than continuous, which is somewhat awkward for rigid structures such as most human bedrooms; often they are used in pairs to compensate for this.)

Ventilating your house is an intermediate case: it's not essential for the water vapor that condenses from the outgoing air to be infused into the incoming air, but it would be desirable, because otherwise an aqueous human is going to want a humidifier in the bedroom. (Maybe hundreds of kilograms of aloes would solve this problem as well.) This suggests that regenerators would be a better choice for this application than recuperators.

Suppose we want to achieve 10 air changes per hour, a relatively normal number of air changes for spaces designed for the humans, and one which in this case would exceed the two or three air changes per hour computed above. This is 100 liters per second in the bedroom I described. If we want to keep air speeds below about 10 m/s to avoid thunderous wind noise, we need to use a duct of at least about 100 mm × 100 mm in each direction.

This presumably means that the regenerators themselves, the pebble beds, will be somewhat wider than 100 mm in order to have comparable air resistance despite being full of chunks, but perhaps not much wider:





To prevent infestations like Legionnaires' Disease, some kind of anti-bacterial and anti-fungal treatment might need to be applied to the chunks. For example, you could use steel ball bearings plated with nickel and then with copper, or (at the risk of deliquescence) rock salt. Some kind of gravel — say, non-clumping clay kitty litter — that's been somehow copper-plated or infused with blue vitriol would also work. Maybe chunks of portland cement would be natively sufficiently alkaline to keep mold or bacteria from growing. (The specific heat of portland cement is 0.880 J/kg/K. Probably the specific heat of clay kitty litter is the same as brick: 0.84 J/kg/K.)

(Shah and Sekulić tell me that paper, wool, hygroscopic nylon, and polypropylene are common materials for such regenerators in HVAC applications, and that the cycle time is frequently measured in seconds.)

How small could we make the pebble beds?

The total thermal mass of each regenerator needs to be much larger than the thermal mass of the air sent through it between reversals of direction. The amount of air sent between reversals of direction needs to be much larger than the amount in the duct, which can be reduced to just over the amount of air in the regenerator pebble beds themselves by doing the reversals of direction close to those beds. This way, if the air ducts from the regenerator to the bedroom has to run a long distance through the house, the air in those ducts can continue moving in the same direction.

Air's specific heat of 1.01 kJ/kg/K is a bit higher than steel's specific heat of 0.47 kJ/kg/K, but steel's density of 7.9 g/cc is much higher than air's 0.0012 g/cc. So if the volumes of air and steel in the pebble bed were equal, the thermal mass of the steel would be about 3100 times greater.

So, presumably, whatever their size, you could send air through them in one direction until you had sent through 150 times their volume of air, then reverse. So far this isn't getting me any closer to conclusions about how big or small the pebble beds have to be if they're just serving to transfer heat and humidity from outgoing air to incoming air.

To take a particular size, suppose one pebble bed is 100 mm × 400 mm in cross-section and 400 mm long, and it's mostly full of steel shot plated as above, close-packed as spheres, which means the spheres occupy $\pi/(3\sqrt{2})$ or 74% of the space. This is 11.9 ℓ of steel, weighing 94 kg, and holding 44 kJ/K, the same thermal mass as 44 kg of air, roughly the entire bedroomfull. If $\Delta T = 22° - 10° = 12$ K, this is 530 kJ, equivalent to running the 2 kW electric heater for four or five minutes. So this is probably overkill, by at least one and possibly three orders of magnitude.

Making the regenerator matrix finer increases airflow resistance but also increases heat conduction between the mass of the matrix and the air passing through. Probably using parallel fins, as in corrugated cardboard, improves this tradeoff. Actual corrugated cardboard itself may be a reasonable material; its specific heat should be close to that of wood, 1–3 kJ/kg/K.

Adapting a regenerator to temperature changes

Sometimes we might have the heater turned off, for example because we're not there, and then want to turn it on.

Topics

Recurrent comb cascade

Kragen Javier Sitaker, 2018-11-09 (updated 2018-11-10) (2 minutes)

A cascade of recursive (i.e. feedback) comb filters at subharmonics of a desired frequency should be able to provide a high-Q bandpass filter at very low computational cost. For example, to isolate a 256-Hz signal with a sampling rate of 1024 Hz, you can subtract the sample 2, 6, 10, 14, 18, or 22 samples ago. Suppose you start by subtracting the sample X[i-14] 14 samples ago, which is 3½ cycles; this produces a new signal A. Now add the sample 16 samples ago from that new signal A[i] + A[i-16]; this produces a new signal B. Now subtract the sample 18 samples ago from that new signal B[i] - B[i-18]. This produces a new signal C. Now add the sample 20 samples ago C[i] + C[i-20], producing a new signal D.

This signal C is the input signal filtered with the product of the frequency responses of the three component filters; that filter has exact nulls at every place any one of the filters had an exact null, including DC, and it has peaks at multiples of the least common multiple of their periods, which I think is well above the Nyquist frequency in this case.

Wait, I'm confusing feedforward and feedback implementations.

Hmm, with unit impulse amplitude this ends up being a linearly growing signal if you make it feedback. You probably need to apply some kind of Hogenauer-style limit thing to keep its amplitude below a limit.

Topics

Notes on local file browsing

Kragen Javier Sitaker, 2019-09-15 (updated 2019-09-28) (4 minutes)

I've been struggling with local-files WWW browsing in Dercuano since its inception, and I've learned a few things in the process.

Android, the special shitty snowflake

As described in How to make Dercuano work on hand computers? (p. 1371), I think that probably PDF is a more feasible solution for *delivery to* hand computers, but I also want to do *development on* hand computers.

I can build Dercuano on Android in Termux with no difficulty, especially now that I've spent the hour to port it to Python 3. However, browsing the built HTML tree is more difficult.

Firefox Lite on Android can't browse local files at all. Chrome on Android can, using the normal file:///sdcard/whatever URLs, but not the entire filesystem. So it's necessary to put the files you're browsing in /sdcard, even if it's "emulated", so that the browser has access to them.

Opening HTML files in Android from Termux with `termux-open-url` does not work well, because it doesn't use the file:// but rather a content:// URL which could perhaps in theory export the whole tree of HTML files (and CSS, and JS) but in practice doesn't. It's rather fiddly to get the correct directory in Termux if you got there via symlinks; I've resorted to `termux-open-url "file://$(readlink -f "$(pwd)")/index.html"`, though maybe there's an easier way.

Chrome on Android is capable of saving HTML files you're viewing, which it does in a MIME multipart/related document. Firefox Lite does not deign to permit mere users such abilities.

Same-domain problems in filesystem browsing

Normal Firefox of course can browse local file URLs; however, it has some rather strange restrictions. dercuano-20190915/index.html (p. 1) can successfully load dercuano-20190915/liabilities/style.css, which can then successfully load, for example, dercuano-20190915/liabilities/et-book-roman-old-style-figures.ttf, and if you click from dercuano-20190915/index.html to, for example, dercuano-20190915/notes/notes-on-local-file-browsing.html, it can also load these files. Similarly, both can load addtoc.js from the liabilities directory. However, if you open dercuano-20190915/notes/notes-on-local-file-browsing.html directly (for example, by pasting the URL, launching it from the command line, or clicking a bookmark), it can load the CSS and the JS but not the TTF, which gets an error message about cross-origin requests in the JS console. So as long as you're clicking around from the top level, you're fine, but entering through another path will result in a mysterious font failure.

Chromium also fails to load the fonts under similar but not identical circumstances.

Whatever concept of "origin" the browsers are using isn't exported

to JS as `window.origin` or `document.domain`, which are consistently `"null"` (a string) and `""`, respectively.

localStorage

Somewhat surprisingly, both Firefox and Chromium support localStorage from local files; Firefox, for one, seems to use the same notion of "origin" that sometimes prevents it from loading the fonts. In both cases localStorage survives browser restarts. This suggests that it should be possible to use localStorage as a persistent cache, but it's not safe as the only way to store important data.

data: URLs

Data URLs such as data:text/html,hello are apparently not supported by Firefox Lite, but they are supported by Android Chrome, and they support JS and base64. Unfortunately I don't think there's a way to get them to be gzipped. This is maybe a useful delivery mechanism for apps that can be encoded in a single QR code (size limit: 2953 bytes without chaining multiple QR codes, which is maybe enough for a decompressor).

Topics

A proposal to support hypertext links in ANSI terminals

Kragen Javier Sitaker, 2013-05-17 (updated 2019-12-26) (13 minutes)

We should totally have a way to render HTML-style links in terminal emulators.

Why this is a good idea

Lots of terminal emulators already have a way to recognize URLs in free text so that you can visit them (ctrl-click in gnome-terminal, an option on the dropdown menu in both gnome-terminal and Konsole), which is useful. But if you're chatting over XMPP with someone who's using HTML (like HipChat), they're probably going to embed links from time to time. And they expect that when they say:

> We'll go from my house to Facebook at around 10:00.

where "my house" is linked to http://maps.google.com/maps?client=ubuntu&channel=fs&oe=utf-8&q=1456+Edgewood+Dr.,+Palo+Alto,+CA&um=1&ie=UTF-8&hq=&hnear=0x808fbb0d745a65e9:0xfde4b05151805922,1456+Edgewood+Dr,+Palo+Alto,+CA+94301&gl=us&sa=X&ei=bZNCUbbuI62g4AOPyYDQDw&ved=0CDAQ8gEwAA and "Facebook" is linked to http://maps.google.com/maps?q=facebook&hl=en&sll=37.45392,-122.13933&sspn=0.007495,0.013797&gl=us&hq=facebook&t=m&z=16, then you will see something like

> We'll go from *my house* to *Facebook* at around 10:00.

and not

> We'll go from http://maps.google.com/maps?client=ubuntu&channel=fs&oe=utf-8&q=1456+Edgewood+Dr.,+Palo+Alto,+CA&um=1&ie=UTF-8&hq=&hnear=0x808fbb0d745a65e9:0xfde4b05151805922,1456+Edgewood+Dr,+Palo+Alto,+CA+94301&gl=us&sa=X&ei=bZNCUbbuI62g4AOPyYDQDw&ved=0CDAQ8gEwAA my house to http://maps.google.com/maps?q=facebook&hl=en&sll=37.45392,-122.13933&sspn=0.007495,0.013797&gl=us&hq=facebook&t=m&z=16 Facebook at around 10:00.

which is pretty annoying and hard to read.

The proposed solution: `ESC [ _` and `ESC [ U < url >`

We define two new ANSI-compatible escape sequences, which ought to be proposed for the next version of ECMA-48 and the corresponding ISO standard:

- `CSI _`, or `ESC [ _`, "LNK": to indicate the beginning of a hyperlink to an URL. All characters produced before the next URL sequence form part of the link.
- `CSI U`, or `ESC [ U`, "URL": to indicate the end of the link text begun by LNK. This sequence is followed by the unencoded text of the

URL to which clicking the link should take the user, preceded by "<" and terminating before the next ">" or " ", which is part of the escape sequence and not displayed.

As an example, a link labeled "Canonical Hackers" linking to http://canonical.org/ could be represented as follows, with `ESC` representing the ASCII escape character:

```
ESC[_Canonical HackersESC[U<http://canonical.org/>
```

Rationale for the design of the escape sequence

Above and beyond the typographical possibilities of whitespace, we already have boldface, underlining, and different colors in our terminal emulators. These are produced by "escape sequences", invisible sequences of characters typically beginning with the ESC character (character 27) followed by a "[", which change the state of the terminal so that subsequently displayed characters will have a different effect. The `ESC[` sequence is called "CSI", "Control Sequence Introducer".

So I think we should define an escape sequence to make the following (or preceding?) sequence of characters into a clickable link to a given URL.

It would be desirable if the escape sequence degraded into something that was human-readable when displayed on terminals that didn't support it. This is only possible to a limited extent, since programs that try to be aware of screen layout will necessarily be confused about where the text is on the screen, but it is still somewhat possible.

ANSI escape sequences can't contain sequences of arbitrary characters, while URLs can contain most characters. Consequently the escape sequences for setting the terminal title aren't ANSI escape sequences: `ESC]0;new title^G`, where `^G` is the BEL character control-G (character 7) and can also be `ESC\`, and `0` can be replaced with `1` or `2`. The `ESC]` sequence is known as "OSC" or "Operating System Command".

xterm's parsing of the above sequence seems to consume anything following `ESC]` until the next `^G` or `ESC\`, even if it doesn't start with a digit, which is pretty unfriendly. This suggests that if we wanted to use the same `ESC]` introduction sequence, incompatible xterm implementations would eat the URL if we terminated it with the same `^G`, and other things as well if we used a different terminator. Even if most people are now using other terminal emulators, this seems like a compatibility trap.

Putting the link escape sequence *after* rather than *before* the text to be marked up offers a certain kind of safety; it's quite easy for an unterminated color escape sequence to set the next few pages of output to white on white, or flashing, or underlined, or whatnot. It also probably improves matters on terminals that don't yet understand the escape sequence, as they could see "Canonical http://canonical.org/" rather than "http://canonical.orgCanonical". Since you presumably need two escape sequences anyway to indicate the boundary of the link, you might as well put the URL in the final

one. So you have one escape sequence to indicate "beginning of link" and another one to indicate "end of link; URL is xyz".

For reasons of tradition and URL-safety, I would like the delimiters (particularly in the gracefully degraded form) to be <>. Ideally the beginning and ending sequences would also have a pleasing visual and memorable symmetry, and would disappear from view entirely in old terminals.

(In the following, ESC means the ASCII character 27, escape.)

This suggests using one of the ASCII matching delimiter pairs []()<>{}``. [] are right out, since they're already in use. ESC( and ESC) cause rxvt to swallow the following character; I'm not sure what their meaning is, but they seem to be used. Treating `` as a matched pair is sadly out of style, due to the unfortunate but now nearly universal adoption from Microsoft Windows fonts of a vertical '.

ESC{ and ESC} disappear in rxvt and xterm, render as literal ESC character glyphs in gnome-terminal, overlaid on top of the {} in my font, and disappear in konsole while producing warning messages on its stderr. ESC< and ESC> disappear in rxvt; the second disappears in gnome-terminal, while the first renders as a literal ESC character glyph; they disappear in konsole. This suggests that perhaps ESC< and ESC> are already taken.

A little looking around suggests that ESC> is "set numeric keypad mode" aka DECKPNM on VT100s, ESC< is "exit ANSI mode" on VT52s, while ESC(and ESC) are used by VT100s to change character sets (setaltg0, setaltg1, etc.)

ESC} on VT100s is "invoke the G2 character set", according to http://rtfm.etla.org/xterm/ctlseq.html, although that's ignored in xterm and probably all other modern terminal emulators.

So this suggests the following syntax:

```
ESC{Canonical.ESC}<http://canonical.org/>
```

Actually, though, you could use valid ANSI escape sequences instead of ESC{ and ESC}. I wanted to use ESC[a for the "begin link" sequence (like <a>), but rxvt already uses it for a nonstandard "move right" sequence, an alternative spelling of ESC[C. (See rxvt-2.6.4/src/command.c:2652, inside process_csi_seq.) The full set of CSI-ending codes handled by rxvt seems to be iAeBCaDEFG`dHfIZJK@LMXPT^Scmnrshltg^W, which is to say, @ABCDEFGHIJKLMPSTWXZ^``acdefghilmnrst.

Argh, so what to use for the other escape sequence? Wikipedia says:

> For two character sequences, the second character is in the range ASCII 64 to 95 (@ to _). However, most of the sequences are more than two characters, and start with the characters ESC and [(left bracket). This sequence is called CSI for Control Sequence Introducer (or Control Sequence Initiator). The final character of these sequences is in the range ASCII 64 to 126 (@ to ~).

This suggests that we could use "ESC[*", which konsole reports as an "Undecodable sequence" and drops, rxvt apparently drops (and isn't in rxvt's list of CSI codes), xterm and screen drop, and gnome-terminal displays literally. "ESC[*" is nice and mnemonic: links are normally underlined.

ESC[U would work for the "end link, begin URL" sequence, which could be followed by the URL wrapped in <>. If the URL is specified

to end at the next space or >, then this sequence would be unlikely to inadvertently gobble up a large quantity of text when random data is sent to the terminal that randomly happens to include ESC[U. So that would give us:

```
ESC[_Canonical.ESC[U<http://canonical.org/>
```

Implementing the escape sequence

You could write your own terminal emulator to support links, but it probably makes more sense to implement the feature in existing terminal-emulation software. The popular free-software terminal emulators are tmux, screen, gnome-terminal, konsole, rxvt, xterm, Emacs, and whatever Apple ships, plus perhaps implementation in ncurses is necessary for much application software to use it.

I looked through the available file on my Ubuntu box to see what other terminal emulators there are. The relevant popularity metrics from http://popcon.debian.org/main/by_vote are:

```
 #rank name                      inst  vote   old recent no-files (maintainer)
 34   libncurses5               137051 119786  7528  9710    27 (Craig Small)
 448  libvte9                    65767 30960 26691  8033    83 (Debian Gnome Maintainers)
 499  gnome-terminal             54405 27886 21381  5118    20 (Debian Gnome Maintainers)
 771  xterm                      77540 17077 50118 10317    28 (Debian X Strike Force)
 918  screen                     45241 12867 30602  1758    14 (Axel Beckert)
 1078 libvte-2.90-9              27674  9519 11363  5591  1201 (Debian Gnome Maintainers)
 1182 konsole                    16489  8229  6662  1590     8 (Debian Qt/kde Maintainers)
 1434 emacsen-common             28896  5699 19961  2805   431 (Rob Browning)
 1609 xfce4-terminal              9368  4110  4360   896     2 (Debian Xfce Maintainers)
 2109 tmux                        6908  2154  4244   508     2 (Karl Ferdinand Ebert)
 2285 lxterminal                  5290  1803  2946   541     0 (Debian Lxde Maintainers)
```

```
 2739  terminator                2158  1274   764   119     1 (Nicolas Valcá
orcel Scerpella)

 2766  yakuake                   2324  1242   972   110     0 (Ana Beatriz G
uerrero Lopez)

 3073  guake                     1499   972   449    78     0 (Sylvestre Led
ru)

 4913  tilda                      724   324   367    33     0 (Davide Truffa
)

 4920  eterm                     1189   322   787    80     0 (Debian Qa Gro
up)

 5130  terminal.app               830   294   509    26     1 (Debian Gnuste
p Maintainers)

 5289  rxvt                      2032   274  1634   124     0 (Jan Christoph
 Nordholz)

 6432  aterm                      972   180   761    31     0 (Debian Qa Gro
up)

 6948  cutecom                    796   148   596    50     2 (Roman I Khimo
v)

 7207  gtkterm                    781   135   619    27     0 (Sebastien Bac
her)

 7299  sakura                     299   132   155    12     0 (Andrew Starr-
bochicchio)

 7376  ajaxterm                   251   128   116     7     0 (Julien Valrof
f)

 7855  mrxvt                      452   110   320    22     0 (Jan Christoph
 Nordholz)

 7768  picocom                    535   113   378    43     1 (Matt Palmer)
 

 7975  mlterm                     628   106   448    73     1 (Kenshi Muto)
 

 8386  fbterm                    1392    95  1097   199     1 (Nobuhiro Iwam
atsu)

 8947  roxterm                    470    82    79     5   304 (Tony Houghton
)

 10260 pterm                      345    59   231    55     0 (Colin Watson)
 
```

```
 10458 jfbterm                  1142    56  1007    78     1 (Debian Qa Gro
oup)

 10771 kterm                     560    52   497    11     0 (Ishikawa Muts
umi)

 13843 evilvte                   117    27    87     3     0 (Wen-yen Chuan
g)

 14563 microcom                  187    24   150    13     0 (Alexander Rei
chle-schmehl)

 14898 xvt                       212    23   177    11     1 (Sam Hocevar)
o

 15979 termit                     71    19    49     3     0 (Thomas Koch)
o

 19919 vala-terminal              44    10    32     2     0 (Debian Freesm
artphone.org Team)

 24553 xiterm+thai                36     5    28     3     0 (Neutron Soutm
un)

 24634 bogl-bterm                 41     4    32     5     0 (Samuel Thibau
lt)

 25809 pyqonsole                  31     4    24     3     0 (Alexandre Fay
olle)

 45654 libterm-vt102-perl          5     0     5     0     0 (Debian Perl G
roup)
```

(I thought Text::CharWidth might be relevant, but it doesn't seem to handle escape sequences anyway.)

Now, libvte9 or libvte-2.90-9 is the actual terminal emulator library that powers a number of the above terminal emulators, at least gnome-terminal, sakura, xfce4-terminal, vala-terminal, tilda, lxterminal, gtkterm, and evilvte. But from looking at the code of gnome-terminal and xfce4-terminal, each separate application built on top of libvte would probably have to write some code to handle clicks on link regions.

It seems that once you add the feature to libncurses5 (in the termcap, at least), libvte9, and gnome-terminal, it's available to 20% of users of Debian and similar systems; if you add it to xterm, you get another 12%, although it's perhaps dubious whether upstream xterm will accept such a patch; adding the feature to screen makes it available to another 9%, although some of them will still be using other terminals (such as MacOS X terminal) to connect to their screen; konsole gets another 6%; emacs ansi-color.el 4%; xfce4-terminal 3%; and tmux 1.6%.

Security

Some escape sequences of the past have been disabled for security

reasons in modern software. However, in general, it's safe to launch arbitrary URLs in a normal browser at the moment, or so we believe; so this should be safe. It may, however, result in terminal users getting rickrolled. It would be useful to have a way to see what the linked URL is before you click on it.

Topics

Radiant heating

Kragen Javier Sitaker, 2018-05-20 (3 minutes)

Humans are most comfortable when the air they are breathing is substantially cooler than the radiant temperature of their environment. If both are at 19–24°, humans can be reasonably comfortable, but it's better to have the radiant temperature at some 40° and the air temperature closer to 20°.

This, of course, implies some kind of continuous energy input — supplied by the sun in our natural environment, whose "solar constant" is 1000 W/m². If the radiant temperature is 40°, without any cooling, things will gradually heat up to 40°, and humans will feel like shit. The air inside a building itself doesn't absorb a substantial amount of radiant energy at thermal IR wavelengths, but surfaces will — even with low-emissivity coatings, they will absorb at least 10% of the incident radiation, which is the radiation they would be emitting at 40° — that's 545 W/m² for a black body, 54.5 W/m² for one whose emissivity at the relevant wavelengths is 0.1. And the 1000 W/m² of the solar constant works out to a radiant temperature of some 91°.

54 W/m² is a relatively reasonable amount of heat to remove through airflow and air conditioning, especially compared to the 1000 W/m² supplied by the sun shining through an open window. What does it look like to supply it through radiant heaters?

Note that you have to have the whole 545 W/m² incident on your body for the radiant temperature to be 40°. But only some of it has to come directly from the radiant heaters; other parts can be reflected, or even emitted, by other materials in your environment. So let's consider what it takes to reach 150 W/m² on the floor directly from ceiling-mounted radiant heaters radiating down.

High-emissivity heating elements that can reach 900° without damage are commonplace. Higher-temperature materials, like the Kanthals, are somewhat expensive. 900° is 107 kW/m², while 1400° (about as high as any Kanthal goes) is 440 kW/m². So each cm² of 900° heater radiator can cover 720 cm² of floor with 150 W/m². To cover a whole 40m² efficiency apartment, you need 560 cm² of 900° heating elements to emit the requisite 6 kW (requiring 25 A at 240 V to produce it through Joule heating).

If you instead use coolant from a phase-change reservoir at, say, 500° (not sure what material melts at 500°, but I'm sure there's something), it only emits 20 kW/m², so you need 3000 cm² (0.3 m²) of emitting panels.

Topics

A note on meditation

Kragen Javier Sitaker, 2019-04-20 (1 minute)

From *The Mind Illuminated*, p. 117:

Meditation is a series of simple tasks, easy to perform, that only need to be repeated until they bear fruit. So where is the sense of difficulty and exertion coming from? We usually describe a task as difficult because we're dissatisfied with our performance, which means we've started judging. Your expectations haven't been met, and maybe you're starting to doubt whether you'll ever succeed, which can sap your motivation. You're not actually struggling with meditating, you're struggling with unrealistic expectations and an idealized image of what you think "should" be happening. As a result, it feels like you're forcing yourself to do something you think you aren't very good at. If you believe those feelings, the ego-Self naturally wants to avoid blame. If you can convince yourself that you've been trying really hard, then the ego-Self doesn't feel guilty for not meeting its own self-imposed expectations. You can blame the teacher, the method, or concoct a story about how meditation isn't right for you. The real issue isn't that meditation takes too much effort, or that something is innately wrong with you, it's your judgment and expectations.

Useful to keep in mind.

Topics

Dercuano formula display

Kragen Javier Sitaker, 2019-04-30 (5 minutes)

Dercuano contains a bunch of text, which can be more readable and comprehensible and navigable if properly formatted (see Dercuano stylesheet notes (p. 374)), but a lot of that text also contains mathematical formulae, things like ½LI² or k (2 cos θ_0 cos² ω - cos θ_0 - 2 sin θ_0 sin ω cos ω) or log cos x ≈ -x²/2 - x^4/12 - x^6/45 - 17x^8/2520 - … when $x^2 < \pi^2/4$. One problem is that Unicode's ability to represent mathematical formulae is pretty limited; you can do x^i but not vice versa, you can't do simultaneous subscripts and superscripts (for things like $\int$), and any kind of two-dimensional layout is pretty limited. Another problem is that the spacing and fonts are all wrong; that last formula, for example, should look more like

$$\log\cos x \approx -\frac{x^2}{2} - \frac{x^4}{12} \ldots$$

only with better fonts and no misalignment between the minus signs and fraction bars. (That'll depend on your fonts.)

The standard approach to solving this problem on the web is MathJax, which scans your HTML after page load for TEX formulas tagged with \(\) and renders them into nice readable formulas. I downloaded a copy, and it doesn't look like it sends the formulas to a server for rendering (the way Wikipedia's alternative does) but it's 33MB. And it looks like a big chunk of that 33MB is necessary to use MathJax in the usual way.

MathJax can generate SVG and HTML with CSS, though, so maybe I could somehow get it to compile my formulas to HTML or SVG at build time. Although, to bring up problems reminiscent of those in Dercuano drawings (p. 64), here's the bletcherous markup it spat out for `$\sim\bigoplus_{i=0}^7b_{n,i}$`, generating screen output which looks absolutely gorgeous:

```
<span class="math" id="MathJax-Span-9" role="math" style="width:
6.018em; display: inline-block;"><span style="display:
inline-block; position: relative; width: 4.959em; height: 0px;
font-size: 121%;"><span style="position: absolute; clip:
rect(1.17em, 1004.96em, 2.78em, -1000em); top: -2.302em; left:
0em;"><span class="mrow" id="MathJax-Span-10"><span class="mo"
id="MathJax-Span-11" style="font-family:
MathJax_Main;">~</span><span class="munderover"
id="MathJax-Span-12" style="padding-left: 0.278em;"><span
style="display: inline-block; position: relative; width: 2.334em;
height: 0px;"><span style="position: absolute; clip: rect(3.087em,
1001.05em, 4.441em, -1000em); top: -4.014em; left: 0em;"><span
class="mo" id="MathJax-Span-13" style="font-family: MathJax_Size1;
vertical-align: 0em;">⊕</span><span style="display: inline-block;
width: 0px; height: 4.014em;"></span></span><span style="position:
absolute; clip: rect(3.359em, 1000.43em, 4.207em, -1000em); top:
-4.491em; left: 1.111em;"><span class="mn" id="MathJax-Span-14"
style="font-size: 70.7%; font-family: MathJax_Main;">7</span><span
style="display: inline-block; width: 0px; height:
4.014em;"></span></span><span style="position: absolute; clip:
rect(3.366em, 1001.22em, 4.207em, -1000em); top: -3.729em; left:
1.111em;"><span class="texatom" id="MathJax-Span-15"><span
```

class="mrow" id="MathJax-Span-16"><span class="mi"
id="MathJax-Span-17" style="font-size: 70.7%; font-family:
MathJax_Math; font-style: italic;">i<span class="mo"
id="MathJax-Span-18" style="font-size: 70.7%; font-family:
MathJax_Main;">=<span class="mn" id="MathJax-Span-19"
style="font-size: 70.7%; font-family:
MathJax_Main;">0<span style="display:
inline-block; width: 0px; height:
4.014em;"><span class="msubsup"
id="MathJax-Span-20" style="padding-left: 0.167em;"><span
style="display: inline-block; position: relative; width: 1.369em;
height: 0px;"><span style="position: absolute; clip: rect(3.143em,
1000.42em, 4.202em, -1000em); top: -4.014em; left: 0em;"><span
class="mi" id="MathJax-Span-21" style="font-family: MathJax_Math;
font-style: italic;">b<span style="display: inline-block;
width: 0px; height: 4.014em;"><span style="position:
absolute; top: -3.864em; left: 0.429em;"><span class="texatom"
id="MathJax-Span-22"><span
class="mi" id="MathJax-Span-24" style="font-size: 70.7%;
font-family: MathJax_Math; font-style: italic;">n<span
class="mo" id="MathJax-Span-25" style="font-size: 70.7%;
font-family: MathJax_Main;">,<span class="mi"
id="MathJax-Span-26" style="font-size: 70.7%; font-family:
MathJax_Math; font-style: italic;">i<span
style="display: inline-block; width: 0px; height:
4.014em;"><span style="display:
inline-block; width: 0px; height:
2.302em;"><span style="display: inline-block;
overflow: hidden; vertical-align: -0.436em; border-left: 0px
solid; width: 0px; height: 1.662em;">

That gzips to 643 bytes. Without using the MathJax fonts or `position:`, you might try doing something like

$$\sim\oplus \quad 7 \quad b_{n,i}$$
$$i=0$$

Markdown won't let me treat a `<table>` as an inline element, but CSS will, and the idea would be to hack the math stuff in further down the processing chain anyway. The source code to that looks like this, reformatted:

```
<table cellspacing=0 cellpadding=0>
    <tr><td rowspan=2 style="font-size: 2em">~⊕</td>
        <td style="font-size: .71em">7</td>
        <td style="padding-left: 3px" rowspan=2><i>b<sub>n,i</sub></i></td>
    </tr><tr>
        <td style="font-size: .71em"><i>i</i>=0</td>
    </tr>
</table>
```

It's 253 bytes, gzipping to 180 bytes — more than the 50 bytes of the original formula, but not the 3 kilobytes gzipping to 643 bytes (not counting the fonts!) generated by MathJax.

So one alternative, which also might be reasonably okay for some

things, is to hack together some kind of thing to spit out some HTML. It won't look as good as MathJax or real T_EX, but it's potentially a lot more capable than raw Unicode, and it should be able to express most of what I want to express.

Certainly when I'm *editing* formulas I want to see them with MathJax, though. And if someone is online and willing to take advantage of MathJax, they should be able to.

Topics

- Math (p. 3564) (78 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Dercuano (p. 3406) (16 notes)
- HTML (p. 3508) (6 notes)
- CSS (p. 3398) (3 notes)
- MathJax (p. 3567) (2 notes)

Sorting in logic

Kragen Javier Sitaker, 2019-12-28 (2 minutes)

I was thinking about Prolog as I lay in bed last night, and I came up with this:

```
%% Sorting a sequence of numbers: a constructive definition.

%% Choosing an element from a nonempty sequence:
choose([X|Xs], X, Xs).
choose([X|Xs], Y, [X|Ys]) :- choose(Xs, Y, Ys).

%% Permutations of sequences.
perm([], []).
perm([X|Xs], [A|As]) :- choose([X|Xs], A, B), perm(B, As).

%% Ordering of sequences.
ordered([]).
ordered([_]).
ordered([X, Y | Xs]) :- X =< Y, ordered([Y | Xs]).

%% Sorting.
sorted(Xs, Sorted) :- perm(Xs, Sorted), ordered(Sorted).
```

This can, in fact, be used in an ordinary Prolog system to sort a sequence of numbers:

```
?- sorted([5, 1, 3, 7, 8, 9, 2, 6, 4], X).
X = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;
false.
```

However, Prolog's standard evaluation order makes this a ridiculously inefficient way to sort, taking a factorial number of steps.

What would you do if you wanted to evaluate this definition efficiently? `perm/2` with its first argument instantiated generates its permutations left to right, and `ordered/1` inspects permutations left to right; it is clear that no sequence of the form `[2, 1 | As]` can ever pass `ordered/1`, so there is no need for `perm/2` to recurse to generate alternatives for `As` in that case. Suppose you could propagate that nogood set from one branch of the program to the other; would that give you an $O(N^2)$ sorting algorithm?

I don't think so, because in the example above, there is only one correct initial sequence of 2 items, and 35 other pairs of 2 items that must be tried and rejected --- but rejecting them involves trying everything that can follow them. I'm not totally sure.

This is a reasonably efficient insertion sort with Prolog's usual semantics, but I think it's still $O(N^3)$:

```
isort([], []).
isort([X | Xs], Ys) :- isort(Xs, Sx), choose(Ys, X, Sx), ordered(Ys).
```

This is vaguely related to Generic programming with proofs, specification, refinement, and specialization (p. 958).

Topics

Studies in Simplicity

Kragen Javier Sitaker, 2007 to 2009 (5 minutes)

In the last year or so, without quite knowing what I was doing, I've been constructing a series of "studies in simplicity". I've been inspired by apostles of simplicity such as Niklaus Wirth and Chuck Moore; by friends from whom I've learned a lot such as Dave Long, Darius Bacon, Norm Hardy, and Aristotle Pagaltzis; and by current projects such as VPRI's "Reinvention of Computing" effort and the "NAND to Tetris" or "The Elements of Computing Systems" course.

I've been doing this to construct working, useful systems with small amounts of code, with the functionality of the aspects of computers that used to scare me, or seem magical, or seem impossible to understand. For a long time, I've known that there aren't *really* magic elves in the computer, but I've always felt pretty uncomfortable with the handwaving I have to do to explain how, say, a compiler works.

Part of my objective has been to enable other people to share my newfound fearlessness. I haven't gotten much feedback, so I can only assume I haven't succeeded in that yet.

To a great extent, this stuff is nothing special; I think it's stuff that any computer science undergraduate does, or should do, in a compilers or operating-systems class, and indeed the "TECS" course covers most of it in a single semester.

Tinylisp (2007–09)

Tokthr (2007–11)

An interactive Forth interpreter under 2kiB; 1000 lines of code; unfinished.

One of the first things I tried was implementing a tiny token-threaded interactive Forth system. I worked on this for about a week, in November 2007. It's not quite complete, and it's 1534 bytes in size when compiled, from about 1000 lines of assembly.

The primary objective is to see if it's possible to build a usable interactive development environment in two kilobytes or so, so that software development doesn't have to be limited to the wealthy and the fortunate. I'm pretty confident that this shows that it's possible, at least for people who like Forth. Here's my basic rationale, quoted from the source:

> There are still a lot of computers out there that have tens of kilobytes of memory or less, and they cost a lot less than, say, a cellphone. Cellphones are apparently still too expensive for half the world's population. I want to see how close I can get to having a comfortable programming environment in a smaller device.
>
> Some smallish microcontroller chips from five different manufacturers, with current Digi-Key prices:
>
> Name bytes RAM bytes ROM MHz price
> ATtiny2313 128 2048 20 US$1.38
> ATMega48-20AU 512 4096 20 US$1.62
> MSP430F1111AIPW 128 2264 8 US$2.43
> LPC2101 2048 8192 70 US$2.52
> H8/300H Tiny 1536 8192 12 US$3.58
> M16C/R8C/Tiny/1B 1024 16384 12 US$3.54
> SX28AC/SS 136 3072 50 US$2.79

In a sense it's a fairly slow interpreter; it needs about 100ns per

bytecode on my 700MHz laptop, about 70 clock cycles per bytecode. That's about as slow as Python 2.4's bytecode interpreter, but the bytecodes are much lower-level, so it's actually slower than Python.

If I (or somebody else) should finish Tokthr, it will be one of the smallest Forths, and indeed the smallest interactive interpreters, ever created; it should provide an interactive development environment in under 2kiB of program with a few hundred bytes of RAM.

Tokthr draws on, among other things:

- My cheap electronics dissection project, where I found out that you could buy a variety of electronic devices for under US$10, some with enough space and computational power to be usefully programmable, but none that were actually programmable (2006);
- Richard W. M. Jones' JONESFORTH, a Forth implementation in x86 assembly language (2007);
- C. H. Ting and Bill Muench's public-domain eForth Model 1.0, which shows that you can build a practical Forth in very little space and very little code (1992?).
- The inspiration of Chuck Moore, obviously, and Jeff Fox's evangelical and sometimes slightly unhinged zeal in telling the rest of us what Chuck had achieved.

Ur-Scheme (2008-02)

Compiler from a subset of Scheme to x86 assembly, written in itself; 1600 lines of code; finished.

This was my first real compiler. It compiles a subset of R5RS Scheme big enough to write a compiler in, which I know because I wrote it in that subset.

Although it takes a very naïve approach to code generation, the code it produces runs surprisingly fast, only about a factor of 5 slower than C compiled with GCC. In part this is due to its omission of first-class continuations and garbage collection.

It does implement closures, and the assembly it generates compiles to statically-linked, standalone executables.

Ur-Scheme draws on many inspirations, which are listed on its home page.

Topics

Amnesic hash tables for stochastically LRU memoization

Kragen Javier Sitaker, 2017-04-03 (1 minute)

A policy rarely used for handling collisions in hash tables: discard the old colliding entry! This is appropriate for using a hash table as a cache, e.g. for memoization. Or push the oldest, or one of the older, entries out of a small bucket, of four or eight items, like a 4-way or 8-way set-associative CPU cache.

If you hash the key twice, so that it hashes into two separate buckets — rather like cuckoo hashing — then the chance of survival goes up significantly. You check both buckets before declaring that it's not found. In this case, you should probably rewrite the entry into both buckets when read-accessing it, so that more-often-read entries will be erased with lower probability. This is an alternative to using 4-way or 8-way set-associative caches; I suspect it will turn out to be more efficient.

Topics

Matrix memory

Kragen Javier Sitaker, 2016-07-27 (1 minute)

If you want 4096 bytes of memory as a minimum, and you're somehow storing the data in some kind of cheap (or even homogeneous) passive elements activated by intersections of wires, and you want to minimize the number of wires, you need ⌈√(4096*8)⌉ = 182 wires in each direction, a total of 364 wires. If you do it in the more usual way, you need 128 wires in one direction and 256 in the other, for a total of 384 wires.

Ferrite cores are the most typical way of storing data in such a memory, but there are other possibilities as well, even at the ferrite-core macroscale. For example, you could imagine a ferroelectric material sandwiched between two layers of indium tin oxide electrodes, one vertical and one horizontal, like an LCD but with PZT or whatever replacing the liquid crystals; or an e-ink (electrochromic) display.

Topics

the oversold-as-low-power Renesas RL78 microcontroller line

Kragen Javier Sitaker, 2019-08-27 (10 minutes)

The RL78/G12 datasheet describes a low-power 16-bit 31-MIPS 24 MHz microcontroller, supported by GCC. At first glance, **it appears to run under 100 pJ per instruction, but this is not true**; my calculations from inspecting the datasheet and the Hardware User's Manual suggest more like 450 pJ per instruction, nearly twice the 230 pJ/insn that excited me about the STM32L0 in Notes on the STM32 microcontroller family (p. 3176) — and the STM32's instructions are 32-bit instructions!

These are my notes from reading the datasheet and some other sources.

Wikipedia tells me it descends from the NEC 78K0R, which descends from the 8-bit 78K/0, which NEC launched in 1986. The "G" in "RL78/G12" is "general-purpose" as opposed to LCDs, industrial, or automotive use.

The R5F10278ANA#W5 member of the family costs US$2.01 in quantity 1, US$0.986 in quantity 1000, at Digi-Key, with almost 5000 in stock. This is the version with 24 pins, 8 KiB of program Flash, 2 KiB of data Flash, and 768 bytes of RAM.

Supporting hardware

It runs on a single power supply, which can be 1.8 to 5.5 V, and includes a ±1% on-chip oscillator, which can do up to 24 MHz. So you don't need a ceramic resonator; you need a crystal if you need tighter timing than that. (The lower-end R5F103 line is ±5%.)

If you do use an external crystal, you're limited to 20 MHz, or 8 MHz if you're under 2.7 V. I don't see anything about being able to use a 32.768 kHz crystal, which would be ideal for precisely timed sleeps.

CPU

It's a 16-bit architecture with a 1-megabyte address space, 2–16 KiB of program Flash (rewritable 1000 times), in some cases 2 KiB of data Flash (rewritable 1,000,000 times if you only care about 1-year data retention, or 10,000 times for 20 years), and 0.25 to 2 KiB of on-chip RAM, and 8 8-bit general-purpose registers — comparable to an AVR in power and the 8086 in convenience, so programming it in C seems like a reasonable thing to do. (And there are four register banks, presumably like the Z80, for interrupt handling and other task switching.) It says it's CISC, but also that it has a 3-stage pipeline.

Oh geez. The registers are "AX", "BC", "DE", and "HL". Then it has ES, CS, 8-bit registers "A", "X", "B", "C", "D", "E", "H", and "L", a carry flag, an auxiliary carry flag (!), a PC, and an SP. The memory map starts with interrupt vectors. This thing is damn close to being an 8080. There's an "ES:" instruction prefix (a la 8086 segment overrides) and a register bank select (a la Z80). Unlike the 8080, it has indexed addressing modes (no Z80-style IX and IY, though). And, yes, there are instruction timings — mostly 1 or 2

clocks, or 4 or 5 clocks if you're accessing Flash.

It's maybe not totally surprising NEC copied Intel; I had an NEC V20 IBM-compatible once.

Unlike the 8086, the ES and CS registers are shifted by *8* bits before forming the effective address, not *4*.

The INC and DEC operations don't affect the carry flag, but they do affect the zero flag.

The instruction set listing (without opcodes) takes 17 pages, but mostly that's because they listed the large number of addressing modes separately (since they vary by instruction), so MOV takes up 2½ pages. Seems like it's probably on the order of a Z80 in complexity.

The general-purpose registers (all four banks!) are mapped into memory. Also, code and data are in the same memory space — no AVR-style Harvardry here.

It includes a 32×32→32-bit multiplier but no floating-point. It's mapped into memory as a device. (All the devices are memory-mapped; there are no IN and OUT instructions.)

The smallest chips in the family, the ones with only 2 KiB of program Flash, can't reprogram their own Flash under program control. The others can.

The interrupt system has multiple priorities, four hardware and one software.

There's a *separate* "RL78 Family User's Manual: Software", which I haven't read; it has instruction maps in it.

Power usage

This is the main attraction! But it turns out to be false. It's actually specified to be worse than an STM32L0!

It says "basic operation" at 24 MHz is 1.5 mA, but "normal operation" is 3.3–5.0 mA; these numbers do not depend on voltage, except that running at 24 MHz at all requires you to be at 2.7 V or more. 3.3 mA ÷ 24 MHz is 138 µA/MHz, twice the 63 µA/MHz touted on the datasheet's front page, but maybe they were talking about "basic operation", which is not defined, but apparently only possible at 24 MHz. Lower-speed numbers do not use lower currents per MHz, but they do use lower voltages; low-speed 8-MHz mode is 1.2–1.8 mA, at either 2 V or 3 V. This is slightly worse — 150–230 µA/MHz — but uses less power (300 pJ/cycle rather than 370) because of the lower voltage.

Moreover, the "31 DMIPS at 24 MHz" on the front page is also misleading. **This is not a superscalar chip**; at best it can run one instruction per cycle, and many instructions take 2 cycles. It might really manage 16 million instructions per second.

That means those 300 pJ/cycle are really 450 pJ/insn! That's almost *twice* what the STM32L0 claims in its datasheet. (I probably ought to verify its claims.)

They also have modes HALT (110–1210 µA, leaves clocks running) and STOP (0.19–2.20 µA, doesn't leave clocks running). Restarting the clock after STOP can take a while.

These power numbers generally don't include peripherals, including the timers needed to wake up. The low-speed on-chip oscillator uses 0.20 µA, the interval timer 0.02 µA, and the watchdog timer 0.22 µA. The ADC uses 500–1700 µA.

Leakage current is specified as ±1 µA per pin, which is enough to be worrisome. (Presumably you could use external multi-megohm resistors to cut down on that.)

"RL78/G12 User's Manual" documents SNOOZE mode, which, using the on-chip oscillator, makes it possible to trigger ADC conversions from a timer without waking the CPU, or to receive UART or SPI data.

In STOP mode you can use the interval time and watchdog timer, but not the PWM-generating timer array. Ports remain latched.

As for "Basic operation"? Not documented anywhere. Maybe they're running a NOP loop or something.

Peripherals

It has three timers (an interval timer, a timer array with 4–8 channels, and one that is a watchdog), I²C, a 4Mbps UART, 12Mbps "CSI" (which appears to be Renesas's name for SPI), and a rather sad ADC. The timers can run from a low-speed 15-kHz (±15%) on-chip clock, which I assume is to save power while sleeping for an interval. Some of its pins are N-channel open-drain for 5-volt tolerant I/O.

There's a DMA controller that can be used for the ADC, SPI, and even GPIO ports.

The ADC has 11 channels, but only 10 bits, and it needs 39 µs to complete a conversion (except in low-voltage mode, when it needs 95). The internal voltage reference is specified as 1.38–1.50 V, which is I guess ±9%, and it has a temperature sensor that's 3.6 mV/°C, which I guess you can select as an ADC input, with 25° being 1.05 V. It seems like this means that measuring the temperature against the internal reference has an error of ±24°, which is unusable except for emergency shutdown.

It has a lot of interrupt lines, like, 15 or so. Presumably this is also to facilitate waking from sleep.

It can source 20 mA or sink 10 mA per pin (40 mA absolute maximum), up to a maximum of sourcing 140 mA or sinking 100 mA (170 mA absolute maximum), plus some other minor restrictions. Also it has loosely specified pullups (10–100 kΩ) on chip.

The timers can be clocked from off-chip data.

Development tools

FreeRTOS and ChibiOS/RT run on it.

GCC supports the R78 line. There's a note in the datasheet about an "on-chip debug function" in the datasheet but it just says it's "provided"; I don't know if it's provided with free software.

There is some documentation of programming waveforms. It has a "TOOL0" pin with a dedicated UART which can be used to reprogram the device via serial data at 115,200 baud, 250 kbaud, 500 kbaud, or 1 Mbaud. The hardware user's manual refers me to "RL78 Microcontrollers (RL78 Protocol A) Programmer Edition Application Note (R01AN0815)," which presumably means that the programming protocol is public and so probably someone has implemented it.

Conclusions

The RL78/G12 looks like a reasonable 16-bit microcontroller, if a bit anemic; it's broadly comparable to an AVR, but faster, and uses

much less power than an AVR does. In a lot of ways it looks like "the 8088 done right", and if there's a version of the chip with more RAM, it might be reasonable to do a self-hosting development environment on it.

However, in the usual cross-compilation context, it seems hard to justify choosing an RL78 over an ARM Cortex-M chip like the STM32L0, which uses half the power, runs more than twice as fast, has first-class debugging support in free software (maybe the RL78 has that too?), and has more pins and an ADC that runs 110 times as fast.

Topics

Interactive geometry

Kragen Javier Sitaker, 2018-04-26 (1 minute)

To make an interactive geometry application, maybe when a point or line or arc or whatever is selected, then the things you can construct from there should be immediately visible and selectable, with a little pushbutton to create each one. Maybe things like midpoints and intersections should be created as soon as segments and lines are, as in Laszlo Pandy's (and Bret Victor's?) dynamic drawing application.

To clean up clutter, you might want to have some constructions that are a little more at arm's length; given a library of them, you could choose one to instantiate a cartoon of it with a little pushbutton on each of its inputs. Then you can constrain it to objects in your scene one by one.

Additionally, you may want to put some things in another layer, then hide that layer. In multitouch, this can be done with a layer quasimode.

(See Two-thumb quasimodal multitouch interaction techniques (p. 1765) for more details.)

Topics

Transmission line computer

Kragen Javier Sitaker, 2016-07-11 (updated 2019-07-23) (7 minutes)

Reading *The Art of Electronics* I was reminded of a rumored unpublished result I vaguely recall from somebody at Bell Labs in the 1990s to the effect that you can build a universal computer if you have a single NAND gate. Could you do something like this in practice using a big coil of coax?

(Coaxial and fiber-optic delay lines are an existing commercial product.)

Table 2.1, on p.74 (the 107th page) of the current edition of *The Art of Electronics* is labeled, "Representative Bipolar Transistors". Almost all of them can switch at 100 MHz; the 2N3904/2N3906 "jellybean" is 300MHz. But at c, an oscillation at 300MHz is 999mm. Typical coax propagates signals almost nondispersively (is this close enough?) at about half of c, so each meter of it could hold about four bits of storage. My 32768-bit benchmark for a practical computer would then require about 16 kilometers of cable. At 2.95mm (the diameter of RG58 instrumentation cables) that's 109 liters of cable, which is a completely reasonable amount of cable to manage, but somewhat costly.

(Some RF-amplifier transistors in the table reach up to 4GHz, reducing all of this by an order of magnitude.)

(With magnetostrictive torsion delay lines, you could reduce the size of all this stuff by many orders of magnitude; if your wire transmits shear waves at 2km/s, which I think is a reasonable ballpark, and is 0.3mm wide, each bit takes up about 300µm×300µm×3.33µm, so 32767 bits is 109mm instead of 16km, nearly a milliliter of wire instead of nearly a cubic meter. It should be practical to have megabytes of memory. But nondispersive magnetostrictive delay lines require special stress-relieved spring wire and magnetostrictive transducers, both of which sound tricky to me.)

http://www.digikey.com/product-detail/en/tpi-test-products-int/508-1200-1M/290-1020-ND/268032 is a hundred-foot (30.5 m) length of 50Ω 30V RG58 coaxial cable with BNC connectors on the ends. They sell it for US$57.49, which would set the price of the above quantity of cable at US$30200. I hope it's cheaper in bulk!

http://articulo.mercadolibre.com.ar/MLA-610317920-cable-rg-58-cuerda-foam-nuevos-_JM is a roll of 100m of 50Ω RG58 coaxial cable without connectors for AR$1583, which is just over US$100, so the cost would be about half of what I said above.

If all of this cable were in a single length, the bit-circulation latency would be about 55 µs (32768 bits of 1.67 ns each), but in practice you would probably want to have at least some modest amount of parallelism; some sort of ideal compromise would be to have something like 128 cables each containing 256 bits with a circulation time of about 427 ns. You probably also want to have some diversity of cable lengths; for example, if the main cycle time is 256 bits, you might want some cables of 255, 254, 252, 248, 240, 224, 192, 160, 96, and 32 bits delay times, allowing nonlocal interaction.

Another reason for wanting shorter delay lines is that the lines are lossy; the cable I linked above is specced to attenuate at 138dB/km at 100MHz. That means that the 100-foot (200ns, say) cable attenuates by about 4.2dB. Your jellybean transistor might have a β between 25 and 150, which means that a single one of them can recover from over 30dB of signal loss and maybe up to 44dB. So 200ns or 400ns of delay line at a time is fine, but if you get up over about 200 meters (700 ns) you might start to need a repeater. At some point you're going to start having SNR problems.

What would the logic design for such a machine look like? I'm thinking of the delay line as containing a number of separate state machines (something like 128 to 4096 of them), each of whose has a state represented in a set of parallel bits in the different cables, one bit per cable; each of them goes through a single state transition every time it cycles through the state-transition active circuitry that they all share. The circuitry is entirely capable of latching some amount of state from one machine to the next; this constitutes an output from the previous machine, and an input to the next.

By itself, this is adequate to implement simple machines like the rule-33 or rule-110 cellular automata. But we can surely build a machine that's easier to program and more efficient than those are. The "nonlocal communication" strange-length links I mentioned above are one useful enhancement: they link these virtual state machines into a much more densely connected topology than the ring topology inherent in the temporal sequence of the delay lines. (If you could only pick a single extra length, rather than the eight I suggested, it should be the square root of the total number; for example, if there are 1024 state machines, each with 32 bits of state, this extra network link should have a delay time of 32 or 992, enabling a message to be routed from any machine to any other in at most 64 hops with an average of 32. The logarithmic network would cut this to a maximum of 10, at the cost of needing ten odd-sized links instead of one.)

(I should read about what Turing's Pilot Ace and the LGP-30 were like, since they had some of this nature.)

On the Pilot Ace:

> The main store of the machine used ten delay lines each holding 32 words of 32 bits. There were also six temporary stores implemented as short delay lines each capable of holding a 32-bit number.
>
> The operations of the Pilot ACE allowed the programmer to specify move operations from one delay line to another. This was achieved by waiting for the number to come round and then "gating" it into the data flow of another delay line. Because it was arranged so that the numbers emerged from the delay lines at the same moment you could only move the nth number in a delay line to become the nth number in another delay line.
>
> If you wanted to change the order of numbers in the long delay lines you had to first transfer the number to a short delay line and then wait for the position in the destination to come round. This made programming more like juggling.

(
http://www.i-programmer.info/history/9-machines/11-an-ace-of-a00-machine.html?start=1)

32 to 128 bits is not enough space to program a very complicated state transition function.

Topics

- Electronics (p. 3430) (138 notes)
- Pricing (p. 3646) (89 notes)
- History (p. 3500) (71 notes)
- Physical computation (p. 3631) (26 notes)
- The LGP-30 computer (p. 3547) (3 notes)
- Cellular automata (p. 3367) (2 notes)
- Delay lines

Differential spiral cam

Kragen Javier Sitaker, 2017-07-19 (9 minutes)

To make an automaton like the Jaquet-Droz artist, you need to program a relatively extensive series of paths.

One mechanically very simple way to do this is with a differential spiral cam gantry, like a couple of Chinese windlasses driving a vertical plotter.

The first step is to wind a couple of threads — one for X, one for Y — around eccentric spools mounted rigidly to a common shaft. The rates at which the threads are taken up by the spools will vary depending on the rotational position of the spools, which can be out of phase. If you hook up the threads to a pen in X and Y, as you wind, the pen will trace a wavy line across the paper as the phases vary.

This is kind of lame, though, because not only does the motion repeat on every turn, but also, in a given direction of windings, the pen is limited to a fairly small range of directions: up, left, or somewhere in between. Moving it right or down is impossible.

The solution to the direction limitation is to make the motion differential: instead of directly driving the pen with the thread, run the thread around a pulley on a long loop between a supply spool and a takeup spool mounted rigidly to the same shaft. Each of the spools can be eccentric or oddly shaped. The difference between their radii at the tangent points of the thread at a particular rotational position results in a net lengthening or shortening of the loop, and thus extension or retraction of the pulley. If they have different overall circumferences, the loop will have a net extension or retraction for each revolution.

Now we have four spools mounted on the same axis and two pulleys connected to the pen; each revolution of the spools moves the pen through a fixed pattern, which may not end where it began, and so repeating the rotation will draw the pattern repeatedly at slightly different positions.

Suppose that we don't want to simply repeat the pattern, though. Then instead of simple drum-shaped spools, we can use spools with helical grooves to entrain the thread; perhaps a 140-µm thread (Dyneema 10-pound fishing line, say) could run in the bottom of a 2mm-deep wide. Now, each rotation of the spool can vary its radius independently of previous and subsequent rotations, as long as you can confidently shape the 2mm-wide groove "above" or "below" the previous and following windings of the groove. You're still limited in how quickly you can increase or decrease the radius — you need a quarter turn to ramp all the way up from or down to zero — but hopefully the differential principle keeps that from being a huge problem.

Now let's suppose we have a reasonable-sized RepRap-printed version of such a doohickey. Let's say, four spools with 2mm grooves, max 100mm radius, and 200mm allocated among the four, 50mm each, printed at 100-µm precision. 50mm of a 2mm-pitch spiral gives you 25 turns, a total of 7.9 meters at the maximal 100mm radius. It probably makes sense to think of the actual motion as being

half of that: about 4 meters. I'm not quite sure how quickly we can change speed, but if we figure that we can control more or less each 100μm unit separately, then we have about 40 000 line segments here to draw with, which is plenty for encoding a Jaquet-Droz-Artist-like picture.

Some final details and variations:

• It might make sense to "gear up" the output so that the detail is rougher than 100μm, but the total line length is more than the 6 or 8 meters you get with this design. For example, you might use an extra pulley or two so that the pen moves 2mm or 4mm every time the thread difference changes by 1mm.
• Rather than a gantry system (like what you see in a construction crane), you could use the standard vertical-plotter configuration, where the two dimensions are the distances from two reference points.
• By using three pairs of spools instead of two, you could overconstrain the pen, keeping it under tighter control; and you could selectively relax that constraint for when, for example, you'd like to lift the pen from the paper. Indeed, this gives you a deltabot, with its attendant three degrees of freedom.
• You could print each spool as two separate prints, instead of all as one piece, so that you have four times the length. And you could probably space the grooves closer than 2mm, though 1mm might be pushing it.
• Instead of using the threads under tension as linear actuators, you could use them to rotate things. I kind of feel like cords under tension eliminate most of the usual problems with linear motion, though, except for lack of precision.
• You could use it to position things other than pens. Markus Kayser used a vaguely similar system in his SunCutter to control the MDF-burning effect of a magnifying glass, although he was using timing-belt-linked big acrylic cams pushing on cam followers on a linear slide gantry moving a table around. You could position machine tools, position squirting nozzles of various materials, trace lines in wet sand, trace lines in wet frosting or other plastic material, move an electro-etching or electrodeposition head around to selectively etch or deposit metal, cut foam with hot wires, cut things with lasers or cutting torches or sparks, etc.
• You could use thicker thread. 140-micron Dyneema fishing line is capable of supporting loads of a few kg; if you were to go up to 1500-micron Dyneema braid, that would hold loads of up to 400 kg or more, 800 kg in each dimension considering it's going around the differential pulley twice. This would also provide greater rigidity, but probably much greater than is necessary; if we want 1mm positioning precision over a 1-meter distance, and our pen or whatever weighs 500g and accelerates at 1 gee, we need to have less than 1 millistrain of deformation at 4.9 N. Doubled-up 140-micron Dyneema with a Young's modulus of 100 GPa would hit a millistrain at about 3 N, so it's already close. Doubled-up 1500-micron Dyneema doesn't stretch a whole millistrain until 350 N; at 5 newtons over a meter, it stretches 28 microns.
• The Grail, of course, is to make the device programmable at run-time, instead of requiring a new set of spools for every new

picture. This can be accomplished by making the spools cylindrical and not eccentric, with a constant radius, but no longer rigidly fixed to the same axis; instead, their axes should be linked with some kind of continuously variable transmission, probably one of the positive-displacement ratchet-controlled types rather than one of the friction-controlled types. The setting of the CVT then determines whether the thread is being extended or retracted.

You could imagine using a lightly angled cone spool instead of a cylinder, then controlling where along the cone the thread winds onto it. This might work, but I fear that the thread might slip too easily along the cone after being wound; maybe a sufficiently bumpy cone surface would prevent that problem, at the cost of less precise positioning.

Eventually, with any of these variations, you run out of thread in one direction and have to change directions. If you're driving the spools with a powerful motor, this may require slowing down and stopping before you hit the end and break the thread. In fact, this is likely to be a showstopper unless you have several hundred meters of thread.

• By threading the thread through various kinds of paths, we can actuate things other than just a pen floating around in space. For example, three threads running through a universal joint are sufficient to pull the joint into any position.

Topics

String cutting cardboard

Kragen Javier Sitaker, 2016-06-30 (5 minutes)

Recently I bought some Dyneema (UHMWPE) fishing line, a 0.5 mm diameter four-strand braid rated for 50kg; there are thinner grades down to about 0.2mm. Among the first things I did with it were to cut my hands a bit (accidentally) and neatly cut some sandwiches (on purpose).

This Saturday I visited the Mini Maker Faire Buenos Aires and talked with a designer who makes cardboard furniture. She gave me a little laser-cut and laser-engraved heart magnet made from cardboard.

I'm interested in self-replicating machinery fed from cheap materials, and in particular fabrication by planar cutting. It occurred to me that you could almost certainly cut cardboard with this string, although I haven't tried yet.

In particular, I think that you can make a kind of string bandsaw out of this string that will cut cardboard with a precision of about a third of its diameter, which would be about 60µm with the 200µm braid. Laser cutters typically manage about 100µm, but cannot be made out of cardboard themselves.

The pressure of the string along the edge of the cardboard needs to be high enough to cut into it rather than just drag over the edge. But the crucial feature here is the pressure, not the force; the pressure is determined by the force and the string's bending radius. In the other parts of the machine where cutting is not desired, the string can be run around a large-radius pulley made from the same cardboard without cutting it, because the larger radius reduces the pressure on the pulley proportionally.

Thinner strings at the same stress will produce less pressure on the edge at a given bend radius, directly proportional to the diameter ratio; if their bend radius is proportionally smaller, they will produce the same pressure. This suggests that using thicker string will provide a larger margin between the pressure needed to cut into the cardboard and the tension the string can withstand, at the cost of less precise cuts and larger radii. Laser cutters are typically capable of 100µm-wide kerfs, which will be difficult to achieve with off-the-shelf Dyneema.

(A 490-newton force reaches the rated stress of 2.4 GPa in this 500µm braid; such a string wrapped around a 3mm-radius half turn will be exerting 980N net on it, spread over 6mm diameter and 500µm diameter, for a final pressure of 330 MPa. By contrast, a 200µm-diameter braid stressed at 2.4 GPa bears only 79 N; wrapped around the same 3mm-radius half turn, it exerts 130 MPa. But if instead it is wrapped around a proportionally smaller half turn with a 1.2mm radius — the thickness of a sheet of cardboard — it exerts 330 MPa again.)

However, the cardboard, too, is experiencing bend radii that depend on the string used to cut it. I don't know enough to know whether this will matter in practice.

The 500µm-diameter braid is comprised of four tows braided together, each of some hundreds of fibers, so each fiber might be 10µm or 20µm across. Rebraiding the braid into smaller braids should

make it possible to get sub-100μm kerfs, at least if the resulting string is still capable of cutting cardboard — the analysis above suggests it may not be.

The same kind of string bandsaw could melt through PVC pipe and perhaps plastic sheets, but it may be better not to use UHMWPE for this. UHMWPE is commonly listed as having a softening point or maximum service temperature of about 82° to 90°, barely above PVC's temperature of 54° to 80°. Spun nylon is the standard string used for friction-melting through PVC in construction; I find different sources giving its softening temperature as 75° or 180° (for nylon 6, polycaprolactam, which is not polycaprolactone) and as 76°, 110°, or 230° (for nylon 6,6), so I don't really have any idea.

Tying knots in the string might help to provide enough irregularity to ensure cutting rather than just sliding.

It should be possible to achieve tight radius turns by rotating the workpiece around the place where the string runs through it.

With the addition of abrasive, this bandsaw could also cut other materials, like metal and glass, but the string won't last long.

Typical lightweight cardboard box material can withstand an edge crush test of 32 pounds per inch ("32ECT"); over 500 microns, that would be only 2.8 N, while this string is capable of withstanding 490 N on each side, almost 400 times as much. This bodes well for the ability to cut cardboard cleanly by this method. Note that by this measure, thicker string is better in proportion to its diameter; halving the diameter halves the force needed to crush the edge, but cuts the strength of the string by four. The longitudinal motion of the string should reduce the effective strength of the paper further.

Topics

Approximate optimization

Kragen Javier Sitaker, 2019-11-13 (3 minutes)

Many successive-approximation algorithms for mathematical optimization --- including the variants of gradient descent that are currently fashionable for training ANNs, but especially things like Newton-Raphson iteration, the method of secants, and the method of Halley --- are quick to converge to a global optimum once they are close to it, but can take a long time to get anywhere close to that optimum.

This suggests that it may be advantageous to do the first iterations of these algorithms using relatively imprecise but cheap means, then do only the last few iterations using higher precision. What kinds of imprecise means might work for the early iterations?

Analog computation

Differential analyzers and similar analog "computers" are commonly accurate to 2% or so, although it depends on the computation, since errors in numerical integration of ODEs, for example, can grow exponentially with time (and this applies equally to analog "computers", even though no actual numbers are involved). Successive-approximation algorithms are something of a best case, though, since rather than growing over time, errors tend to die out over time.

So it seems likely that a properly configured analog computation circuit could approximate the solution within the limit of whatever its signal-to-noise ratio is; commonly 60dB (0.1%) to 80dB (0.01%) is reached. You could imagine some kind of crossbar interconnect made out of CMOS analog switches --- multiplexor/demultiplexors --- to interconnect op-amps, resistors, capacitors, OTAs, and entire analog processing blocks.

0.01% is close enough that two further iterations of a quadratic-convergence method such as Newton-Raphson iteration would reach the precision limits of double-precision floating point.

One difficulty is that, with the exception of integration over time provided by capacitors and differentiation over time provided by inductors or gyrators or the like, analog circuits normally have no memory --- their output is a function of their input, not a function of all their past inputs. This means that you need separate circuits for each scalar variable in your problem state; a single vector equation like $a = b + c$ might require hundreds or thousands of op-amps, unless you can arrange for those vector variables to be indexed by time.

Analog delay lines of various kinds can improve this situation considerably; the old standards for slow signals were thermostatically-controlled piezoelectric mercury delay lines and later torsional magnetostrictive delay lines, with Pupin-like cascades of inductors and capacitors or coaxial transmission lines for faster signals. The approach of using a CCD or similar camera sensor described in CCD oscilloscope (p. 1861) is a potential pure-analog solution that permits some degree of out-of-order access.

In this context, though, it may not be necessary; maybe we can offload the problem of memory onto digital circuitry and feed it

through a DAC feeding the analog circuitry, then digitize the result with an ADC to store it again.

Low-precision floating-point and fixed-point

"Half-precision" 16-bit floating-point is commonly available in GPUs, "TPUs", and modern SIMD instructions; typically it runs at the same cycle time as double-precision floating point, but four times the parallelism. So it's an obvious candidate.

Even lower precisions may be useful for computing initial approximation. Some kind of 8-bit floating-point format, for example, or 8-bit saturating fixed point.

Topics

A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination

Kragen Javier Sitaker, 2017-06-21 (updated 2019-12-27) (3 minutes)

I previously calculated (in Fast sea salt evaporator (p. 1087)) that the enthalpy of vaporization of water (2.26 MJ/kg) and terrestrial mean insolation (180–280 W/m² at temperate latitudes) make solar evaporation a relatively inefficient method of desalination: 80–120 mg/m²/s of water. This also makes sea salt production relatively inefficient. Indeed, these efficiencies are so low that even normally abundant solar thermal energy only permits their use in niche cases.

By comparison, the Sorek reverse-osmosis desalination plant forces water through semipermeable membranes at 7.1 MPa, producing drinking water at a total cost of US$0.58/kℓ. 7.1 MPa × 1 kg/ℓ is 7.1 kJ/kg, or 0.0071 MJ/kg, about 320 times less energy than the enthalpy of vaporization. 7.1 kJ/ℓ works out to 2.0 kWh per kiloliter, which costs between US$0.08 and US$0.24 at 4¢ to 12¢ per kWh, current US electrical prices. By contrast, 2.26 MJ/kg works out to 630 kWh/kℓ, which would be US$25 to US$75 per kiloliter, compared to Sorek's US$0.58. (The reason solar desalination happens at all is that you don't have to buy the energy.)

However, vaporizing water once with solar energy is a silly way to do things. When that water condenses, it releases its enthalpy of vaporization again. If you could harness that heat released in condensation to vaporize more water, even a few stages of the process would improve the efficiency of the system dramatically, although 17 stages would be needed to bring it closer to Sorek's consumption. However, 5 stages would reduce the energy consumption to 450 kJ/ℓ, or, say, increase to 500 mg/m²/s.

A tricky problem is that you need to deliver this condensation heat into water that is cooler than the condensing steam or water vapor, for example by running the steam through coils cooled by water that is evaporating. It would be sufficient, if perhaps not necessary, for the coolant water to be boiling — but at a lower temperature. This approach requires that each successsive stage of the desalination apparatus operate at a successively lower pressure — exponentially lower: 85 kPa for 95°, 70 kPa for 90°, 58 kPa for 85°, 47 kPa for 80°, 39 kPa for 75°. The water thus produced needs to be pumped out of these partial-vacuum chambers against this pressure — 62 kPa for the 75° chamber. While this is less than 1% of the pressure used in the Sorek reverse osmosis plant, and therefore will not add a significant *energy* cost, it still represents machinery that adds significant complexity to the apparatus.

I suspect that this quintuple-acting vacuum cascade approach will make solar water distillation sufficiently more inexpensive to permit its use in a wide variety of settings where it is currently too expensive.

Update: this method is called "multi-stage flash distillation" and is currently in widespread use; reverse osmosis typically uses less energy, as described above, so few new MSF plants are being built. The margin is sufficiently large that you can desalinate a larger amount of water per sunlight joule even by powering a reverse-osmosis plant from photovoltaic panels.

Topics

Intermittent fluid flow for heat transport

Kragen Javier Sitaker, 2019-07-10 (4 minutes)

As mentioned briefly in Heating my apartment with a plastic tub of hot water (p. 1310), intermittent fluid flow can transport heat better than continuous fluid flow with the same average speed. The case there is that heating my apartment with the shared hot-water heater would require 19 mℓ/s, but drawing that continuously would just heat up the pipes in the wall, not the apartment. If, instead, I draw 300 mℓ/s for two minutes, I can put 36 liters of hot water into a bucket or radiator, which can then release the heat into the apartment over the next half hour. This eliminates more than 85% of the heat loss — although the half-liter or so of water left in the pipe for that half hour will lose nearly all of its heat, that's only 3% of the total heat.

This is a general property of heat transport through thin, lossy tubes: if the amount of time the transport fluid spends in the lossy tube is pushed to extremes by using pulsing flow, the overall system efficiency improves, sometimes dramatically. I speculate that, in the mass-transport realm, this might be a reason vertebrate hearts use the shocking, violent pulsing motion that they do, incurring an obvious waste of energy from unnecessary viscous energy losses, rather than the peristaltic motion used by earthworm hearts and insect hearts: that way, most of the blood can spend more of its time in the capillaries and less of its time uselessly losing oxygen in large vessels, and furthermore even the venous part of the non-pulmonary capillaries gets fully-oxygenated blood, and even the venous part of the pulmonary capillaries gets fully-deoxygenated blood to oxygenate and decarbonate.

This has a direct application to ice vest design (see Ice pants (p. 298).) Here you have a substantial amount of coolant (salt water, say) sitting in tubing inside the vest, absorbing heat from your body, and a substantial amount of coolant sitting in tubing inside an ice pack, releasing heat into the melting ice. If you use a continuous slow flow, all the coolant passing through the transitional tubing between these two points is going to lose an unnecessarily large amount of heat, and additionally, part of the vest will be ice cold, while another part will barely be cold at all, decreasing heat transfer; an analogous phenomenon will impede heat transfer within the ice pack, where the coolant near the exit is already too cold to release much additional heat. If, instead, you use periodic sudden pulses that replace most of the vest coolant at once, the whole vest will be a spatially uniform (but temporally oscillating) temperature, increasing heat transfer substantially.

Probably the way to do this electrically is to use a small electric motor to wind up a spring, then release a brake on the spring to drive the pump. This avoids the need for a large electric motor.

I think I hadn't noticed this previously because of an aesthetic or philosophical inclination toward symmetrical, steady-state solutions, which are easier to analyze, rather than oscillatory or asymmetric

solutions: wheels rather than legs, turbines rather than pistons, linearity rather than nonlinearity, flat rather than curved walls, continuous rather than crenellated or fractal webs. I speculate that this inclination is shared by much of modern science and engineering and represents a significant blind spot.

Topics

Microprint visor

Kragen Javier Sitaker, 2016-09-07 (2 minutes)

I'm trying to figure out how to do a paper microprint system for my notebook.

If I print text out at a line height of six 600dpi pixels, it's like uh 100 lines per inch so 0.72 point. For text to be comfortably readable with the naked eye it needs to be 12-point, which is the standard line-printer 6 lines per inch; that's 16⅔× bigger. So a head-mounted magnifying apparatus of between 16× and 30× is probably adequate to make it comfortably readable.

The column width shouldn't be more than about 0.3 radians in order to be comfortably readable. At 30×, an apparent 0.3 radian image is something that was originally 0.01 radians; this means that if it's at a distance of 500mm (from the lens), the column width is then 5mm, which is 118 pixels at 600dpi; at 3.5 pixels per character, that's only 33 characters, about 6 words. So being towards the lower end of that range is probably desirable for ergonomic reasons.

4.8× handsfree magnifying visors are readily available, but 15× and 30× are harder. I did find a 20× one with LED illuminators and a 10× monocle one each for about US$27.

If we stick to 10×, we want 1.2-point fonts, which are 10 pixels high on a laser printer, or maybe 8 or 9 pixels with 1 or 2 pixels of leading. If the glyphs occupy 5 pixels horizontally, then a 600×600 pixel square inch can contain 7200 characters of text, or about 1440 words; a 4×5 inch A6-size page with some margins can hold twenty times that, 144 kilobytes, 28800 words or about 27 standard line-printer pages. Call it 32, then 16 pages (8 sheets of paper) holds 512 standard pages.

(There's still the unknown of whether I can properly align the pixel grids and fully exploit the 600dpi theoretical capability of the printer, or whether I have to stick to Nyquist, taking a 4× areal density hit.)

Some ideas of what to put in it:

http://piratepad.net/4e311Fk9yz

Topics

Storing CSV records in minimal memory in Java

Kragen Javier Sitaker, 2015-09-03 (6 minutes)

Objective: store CSV records in minimal memory in Java. Main sub-objective: minimize number of heap allocations without costing too much runtime.

Brief overview

A class per CSV format, with field types defined and fields named. Private booleans to indicate whether the fields have been parsed. Store CSV line in a char array. Store comma offsets in int array. Actually, use the same array for both. Loop over the line in getters to set the requested field when it's unknown. Weakly reference String fields to avoid either retaining no-longer-used Strings on the heap or multiplying a String on the heap because it's gotten many times. Optionally intern the Strings in a (possibly weak) hash table to handle the common case where many lines have the same value for a string field.

In more detail

From a list of CSV fields and their types, we generate a class automatically with a getter method for each field. Instantiating this class with a line from the file leaves behind only two heap allocations: one for the instance, and one for an array containing the line's characters and the offsets into them where each field starts.

We could find the field boundaries lazily; in that case, the method template code looks something like:

```
boolean hasQuarter = false;
int quarter = -4242;
int getQuarter() {
    if (hasQuarter) return quarter;
if (knownOffsets < 4) findFieldOffsetsUpTo(4);  // ensure offsets[3] ok
int q = quarter = parseInt(data[3]);
hasQuarter = true;
return q;
}
```

Thus when you invoke getQuarter() the usual case will be to check the boolean, find that it's true, fetch the field, and return it. The next most common case will be to fetch `knownOffsets`, compare it to 4, find that it's not less, and jump ahead to the call to `parseInt`, which iterates over the characters from the relevant point in the character array; hopefully that will get inlined.

It might make more sense to always find the field offsets when the object is created (i.e. not find them lazily, but rather eagerly), because that would avoid storing knownOffsets and conditionally testing it every time a field is parsed.

Laziness

Most of the time that I'm processing CSV files, I don't process all the fields. I might just want the maximum value of a single field, for example.

(I'd actually like to be able to avoid doing even character set conversion, since it's both potentially lossy and also uses up CPU time for no good reason on fields that I'm not parsing. Just keep the data in byte arrays. This is inconvenient to do in Java.)

Strings

Strings are a little more difficult than primitive types like floats and doubles, because strings are heap-allocated, so if we're not careful, they can eat up a lot of memory and also put pressure on the garbage collector. There are two strategies to deal with this:

• Weak references. A weak reference doesn't prevent an object from being collected by the garbage collector, but if the object hasn't been garbage-collected yet, it allows you to avoid recreating the object. So the idea is that when we lazily parse a string field, we store a weak reference to the created String value, and return a strong reference to it. As long as the client program holds on to that String, we can keep returning references to it, but if it releases it, it becomes fair game for the garbage collector, and then we may have to recreate it if the program asks for it again.

• Interning. Consider this CSV data from the R distribution:

```
Name,Country,City,OK,CountryCode
"Argentina (La Plata)",Argentina,"La Plata",1,ar
"Argentina (Mendoza)",Argentina,Mendoza,1,ar
"Australia (Canberra)",Australia,Canberra,1,au
"Australia (Melbourne)",Australia,Melbourne,1,au
```

Some of its text columns contain data that will occur only once, but other columns are drawn from a small set of alternatives, like the world's countries. The most effective way to reduce memory usage for those other columns is to ensure that every CSV object returning the string "Argentina" from `getCountry()` is returning a reference to *the same* String object, so that it only has to exist once on the heap. To achieve this, we'd maintain a hash table of existing "interned" strings, and when we want to parse out a field, we check to see if it's already in the table before we allocate it.

This is a little bit tricky because using the standard Java HashMap or even Hashtable objects requires that we have already allocated the String, but a custom hash table implementation avoids that problem.

The potential problem with interning is that if you intern values in a unique field, your hash table retains and eventually tenures strings that will never occur again and do not need to be retained.

These two strategies can be combined to get the best of both worlds: a custom hash table containing weak references to Strings both avoids duplicate allocation and avoids retaining data that doesn't need to be retained.

Weak references are a little less efficient than strong references. I don't yet know enough about Java's implementation of them to know whether they impose an extra cost on the garbage collector (push notifications) or on the mutator (pull notifications) or both.

Line parsing

One very common CSV format, Excel's, allows newlines inside of fields as long as they are inside of double quotes. This implies that you cannot simply walk into Mordor by reading a line and turning it into a CSV object; you need to involve the CSV-parsing stakeholder earlier in the process so that it can tell you when you're done parsing a record. And, as long as it's doing that, it might as well remember where the fucking commas are.

External buffers

Going a little bit further to reducing allocations, copying, and bounds-checking, we could avoid creating a separate char array for each CSV object.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Java (p. 3531) (5 notes)
- Comma-separated values (CSV) (p. 3399) (2 notes)

Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder

Kragen Javier Sitaker, 2013-05-17 (5 minutes)

Lots of people suffer from "seasonal affective disorder", or just think they'd function better, because they spend a lot of their awake time in environments illuminated at 50–500 lux, which is hundreds of times dimmer than full daylight. 10klux is the low end of full daylight; 32klux is the low end of direct sunlight; 130klux is the high end.

What's the minimal amount of equipment I'd need to illuminate my body (including eyes, but other things just in case) at 10klux?

A lux is a lumen per square meter. At 555nm, the most efficient wavelength, a lux is 1.5 mW/m², so 10klux would be 15W/m². I might have a surface area of about 2m², so I'd need about 30W of light, or 20klm. The most efficient light source is probably an LED; red-orange and green LEDs are around 95 lumens per watt, which means I'd need (20klm/(95 lm/W)) = 210 watts of LED power to reach that. Common LEDs are closer to a third of that efficiency, so I'd need about 630 watts, and about US$600 worth of LEDs. Cree Inc. supposedly has 200lm/W white-light room-temperature LEDs available, but Digi-Key only has up to 153lm/W.

Cree's CXA2530-0000-000N00S40E7 ("Cree XLamp CXA2530 19mm white") LED array, in stock at Digi-Key, costs US$25.50 in quantity 1 and delivers 4643 lm at 1.5 amps at 37 volts, for 84 lm/W; it gets above 100 lm/W at half the current. (That's about 15% efficiency: 150mW/W.) Getting 20klm out of these guys would require five of them, costing US$127.50; they could then deliver 23klm at max current (278W), or 15.5klm at optimal efficiency (148W). Linearly interpolating, that gives you 20klm at 226 watts (+ 148 (/ (- 20000 15500) (/ 1 (/ (- 278 148) (- 23000 15500.0))))).

So you could probably cure seasonal affective disorder for the cost of US$130 plus a three-hundred-watt 37-volt power supply (say, US$50), plus 226 watts. You make a chamber of mirrors with all-transparent furniture, stick the brilliant lightbulbs into it (maybe with some frosted glass to make them less harsh), and go sit in it, naked, for some 16 hours a day. (You could imagine that this might cause some other kinds of disorders related to social interaction, though.) 226 watts for 16 hours a day is 151 watts or 1320 kWh/year, or US$132 per year, assuming a 100% efficent power supply.

Given the high power cost, so it probably makes sense to use six of the Cree 19mm LED arrays instead of five to improve efficiency: the cost for the LEDs jumps from US$127.50 to US$153, but the necessary power drops: if the output light from a single array in the 0.8 to 1.5 amp range is 3095lm plus (/ (- 4643 3095) (- 1.5 0.8)) = 2211 lumens per amp, and 20klm is 2860lm per array, we can interpolate to (+ 0.8 (/ (- (/ 20000 6.0) 3095) 2211)) = 0.91 A per array, or (* 37 6 0.91) = 202 W. That means we can get by with a two-hundred-watt power supply and also reduces our power usage by more than 10%, to

1180 kWh/year, US$14 less. The extra array pays for itself in energy savings in less than two years, not even taking into account the likely parts savings from the smaller power supply, the increased resilience to part failure from having more redundant LEDs, and the lower junction temperature.

From the datasheet, it doesn't look like you can significantly improve efficiency by lowering the current further.

20%-efficient photovoltaic panels covering the roof of a 10-meter by 10-meter house in a zone with 300W/m² average day/night insolation would give you 6kW average. If you were powering your LEDs with photovoltaic panels, you could provide light for about 40 people. Of course, during summer daylight hours, it would be more efficient to just put the daylight inside the house directly; then, instead of 20% efficiency at the panels times 15% efficiency at the LEDs, for a total of 3% efficiency, you'd have more like 80% efficiency. Too bad that only works a small amount of the time.

If you only want to illuminate your eye pupils at 10klux, well, your pupils are about 4mm in diameter, so they have a total area of about 25 millionths of a square meter; a quarter of a lumen would suffice to illuminate them to 10klux. You could get that out of 2.5 milliwatts, a hundred thousand times less than what I'm proposing above.

Topics

Vectorized prefix sum

Kragen Javier Sitaker, 2017-07-19 (5 minutes)

You can apply the parallel prefix sum algorithm in a serial, vectorized way using a logarithmic number of Numpy's vector operations. The parallel prefix sum algorithm puts the vector elements into the leaves of a complete N-ary tree (here I use N=2) and propagates partial prefix sums first up and then back down this tree.

Here you have the sum of 16 numbers in four elementwise-parallel operations using strided arrays:

```
>>> import numpy
>>> x = numpy.array([1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8])
>>> a = x[::2] + x[1::2]
>>> a
array([ 3,  7, 11, 15,  3,  7, 11, 15])
>>> b = a[::2] + a[1::2]
>>> b
array([10, 26, 10, 26])
>>> c = b[::2] + b[1::2]
>>> c
array([36, 36])
>>> d = c[::2] + c[1::2]
>>> d
array([72])
```

You can then use reverse the ordering to propagate the prefix sum back down to the leaves. The 72 in `d` is of course the prefix sum at the end of the right half of the tree, and `c` already contains the prefix sum of the left half of the tree. For reasons that will become somewhat less obscure below, we can transpose and ravel to perfect-shuffle them into the right order:

```
>>> cp = numpy.array((c[::2] + numpy.concatenate(([0], d))[:-1], d)).T.ravel()
>>> cp
array([36, 72])
```

The corresponding full prefix sums at the points represented by `b` are half elements of this `cp` and the other half composed from the even-indexed elements of `b` added to the previous element, or 0 if none.

```
>>> bp = numpy.array((b[::2] + numpy.concatenate(([0], cp))[:-1], cp)).T.ravel()
>>> bp
array([10, 36, 46, 72])
```

Then we can do the same trick to get an expanded version of `a`:

```
>>> ap = numpy.array((a[::2] + numpy.concatenate(([0], bp))[:-1], bp)).T.ravel()
>>> ap
array([ 3, 10, 21, 36, 39, 46, 57, 72])
```

And similarly to get the whole prefix sum:

```
>>> xp = numpy.array((x[::2] + numpy.concatenate(([0], ap))[:-1], ap)).T.ravel()
>>> xp
array([ 1,  3,  6, 10, 15, 21, 28, 36, 37, 39, 42, 46, 51, 57, 64, 72])
```

A thing to note about this expression is that it has only two O(N) operations in it: the prepending of the 0 and the addition. The stride-indexing, transposition, and raveling operators are all constant-time.

In a sense, this is not very interesting, because numpy already has a prefix-sum operator:

```
>>> numpy.cumsum(x)
array([ 1,  3,  6, 10, 15, 21, 28, 36, 37, 39, 42, 46, 51, 57, 64, 72])
```

But the above transformation can be generalized to any monoid, not just integer addition, and in particular to function composition, which means it can in theory be used to generate the sequence produced by any definite loop that can be expressed with these operations. It's a little bit tricky in that the monoid elements are functions representing the action of an iteration of the loop and their composition takes the place of addition in the above calculations. If you're going to represent them with numpy array elements, they need to be in some sense constant-space, and you may need many arrays.

As a useless and less trivial but still comprehensible example, consider the action taken by a loop decoding a digit string in some base b:

```
result = 0
for i in range(k):
    result = b * result + digits[i]
```

The action taken by any n iterations of the loop is to multiply the previous result by b^n and add some number m to it. To compose two such actions, (n_0, m_0) followed by (n_1, m_1), we compute $(n_0 + n_1, m_0 \cdot b^{n_1} + m_1)$; processing a number m merely produces (1, m). So we need one array (at each level) for n and one for m. Moving up the tree:

```
>>> na = numpy.ones(len(x))
>>> ma = x
>>> nb = na[::2] + na[1::2]
>>> nb
array([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
>>> mb = ma[::2] * 10**na[1::2] + ma[1::2]
>>> mb
array([ 12.,  34.,  56.,  78.,  12.,  34.,  56.,  78.])
>>> nc = nb[::2] + nb[1::2]
>>> mc = mb[::2] * 10**nb[1::2] + mb[1::2]
>>> nc, mc
(array([ 4.,  4.,  4.,  4.]), array([ 1234.,  5678.,  1234.,  5678.]))
>>> nd = nc[::2] + nc[1::2]
>>> md = mc[::2] * 10**nc[1::2] + mc[1::2]
>>> nd, md
```

```
(array([ 8.,  8.]), array([ 12345678.,  12345678.]))
>>> ne = nd[::2] + nd[1::2]
>>> me = md[::2] * 10**nd[1::2] + md[1::2]
>>> me
array([  1.23456781e+15])
```

If we move back down the tree, we get all the intermediate results. This is somewhat tricky, though conceptually straightforward, and is left as an exercise to the reader, or to me at a later date maybe.

Note that everything in the above has used only the monoid operation and the identity element 0. But we don't actually need the 0; the algorithm is still applicable to semigroups lacking an identity element. A practical example of this is finding the minimum or maximum of the elements seen so far.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Python (p. 3671) (27 notes)
- Prefix sums (p. 3645) (18 notes)
- SIMD instructions (p. 3711) (10 notes)
- Parallelism (p. 3616) (8 notes)
- Numpy (p. 3600) (6 notes)

Absurd household materials

Kragen Javier Sitaker, 2018-04-26 (updated 2018-05-18) (8 minutes)

Suppose we wanted to take advantage of modern materials, manufacturing, and pricing to make high-quality household goods. What would that be like?

How about a copper bowl? A 2mm thick copper bowl, maybe alloyed with a bit of some cheap metal to harden it, would be quite solid. Luxurious, even. What if it were a 300mm hemisphere? That's 570 mℓ of copper. Copper's density is almost 9 g/cc, so this would be 5 kg of copper, which would make it a rather heavy bowl. But copper only costs about US$2.20 per pound, according to the USGS (US$4.85/kg), so this is about US$25 of copper. Not a terribly cheap bowl, but that's like US$25 for your entire life.

Even if US$25 and 5 kg is too much for a bowl, maybe it would be reasonable to copper-plate other things. Copper-plating is very cheap, and reduces infection risk, because copper kills germs. Bathtubs in particular ought to be ideal.

How about making your glasses from synthetic sapphire instead of soda-lime glass? High-purity fused aluminum oxide costs US$1440 per ton (US$1.59/kg), and about 140,000 tons of it are consumed by the US each year (as abrasives), a fraction of the worldwide use. So price and supply are not problems, but of course processing requires high temperatures and/or hard cutting tools. But the glasses would be very difficult to break or scratch and would have a totally different ring to them. Granular aluminum oxide would make a nice powder-coat enamel for materials that can resist the temperature — much like the alumina ceramic we use for spark plugs.

High temperatures probably call for molybdenum, of which the world mines some 200,000 tons per year at a price of US$14.50/kg. This leads me to believe that refractory molybdenum crucibles for growing synthetic sapphire crystals would probably only contain US$100 or so of molybdenum. However, the only sources of molybdenum I can find locally sell small reels of 110-micron molybdenum wire for manual cutting of cellphone glue, at prices like AR$35 (US$2) per meter — that's about 100 mg (molybdenum weighs 10.3 g/cc), working out to about US$20000/kg.

(This molybdenum wire might be useful for heating elements? Molybdenum doesn't melt until 2623°. But it oxidizes at much lower temperatures with a volatile oxide, which I suppose is to say that it catches fire.)

Pyrolytic graphite has a nice sheen to it, and although it's brittle, it could probably be deployed advantageously to substitute for metals in many places.

Dichroic iridescent surface coatings would be a nice addition to many materials, and indeed iridescent coatings are widely available in decorations, but are currently treated as a sign of poor taste. Titanium dioxide makes the brightest colors, due to its high index of refraction.

Aerogel would be a nice substitute for fiberglass in many applications, and allows transparent high-value insulation. You could imagine the glass in the front of your oven, for example, being a silica aerogel sandwich between two plates of monocrystalline synthetic

sapphire. But silica aerogel is brittle, and the powder is irritating to human skin. (It ought to be safe from silicosis by virtue of being amorphous, though.) Aerogels made from other materials, such as gelatin, might also be useful for transparent high-value insulation, even if they don't resist such high temperatures.

Near-microscopic flexures ought to be able to make comfortable cushions, with tailored stress-strain curves, from heat-resistant, water-resistant materials — and, unlike glass fiber or ceramic fiber, it should be feasible to make them safe for human skin contact. Even a silicone or Teflon layer would aid greatly in making it possible to clean waterproof cushions.

An annoying problem I'm confronting at the moment is that available floor materials are either combustible or uncomfortably hard. An aluminum honeycomb sandwich, perhaps with aluminum mesh on the surface, could perhaps solve this problem. The aluminum has enough compliance to bounce like a wood floor, without being combustible like a wood floor. And then you can paint it with a surface treatment that reduces slipperiness and changes the color without adding too much combustibility — perhaps some kind of sand in a sodium-silicate or portland-cement binder, although that itself might suffer from cracking and flaking (perhaps with sharp, glassy edges!) if the surface is as flexible as a wood floor. Alternatively, a thin layer of plaster might work — either plaster of Paris, sand in plaster of Paris, or sand in lime. Gypsum is comfortingly soft to sit on, porous to absorb sweat, and cool without the coldness of glazed ceramics.

A bit of mica would be a nice addition to many surfaces.

Doorknobs have no need to be hollow or have sharp edges. This is not so much an issue of materials as it is of shaping. Carved soapstone would be fine.

Polycarbonate, or perhaps polycarbonate with a harder surface to resist scratching, would be a nice alternative to glass for tabletops. Candidate harder surface materials might include potassium-nitrate-hardened soda-lime glass, fused quartz, synthetic quartz crystal, and synthetic sapphire.

Garbage cans should have nonstick coatings: Teflon or silicone, for example. Moreover, organic garbage should be shredded and desiccated before it has a chance to decay, rather than dumped into a garbage can to potentially decay anaerobically. (You can add water to compost it later if you like.)

Teflon coatings would also dramatically improve zirconia kitchen knives.

There's a lot we can do for soundproofing: isolating walls from small vibrations with the same kind of path-lengthening tricks we use for high-efficiency window frames, filling walls with fibers and foams, angling walls and ceiling to reduce standing waves, punctuating sound-absorbing walls with many small holes, and so on. None of these are novel, but they are rarely used in the home.

Sometimes it is nice to have lighting coming from small pointlike sources — when you want to bring out the sparkle of water drops or a gem, for example, or emphasize the surface profile of a sculpture you're working on. Other times, it's nicer to have ambient light, a desire incandescent lights were poorly fitted for. However, with LEDs, light diffuser panels, and electroluminescent materials, it's

relatively easy to put light in as many different places as you like: inside the cabinets and drawers when you open them, in a reproduction of the constellations all over your ceiling, in light diffuser panels covering the whole ceiling, inside your tabletop. Moreover, you can make the lights of whatever color you like, even changing them from moment to moment.

Teflon fabric is a really remarkable material. It cannot be stained, is undamaged by sunlight, does not decay, and is nontoxic, being entirely biologically inert.

UHMWPE is also a really remarkable material. It nearly cannot be stained, does not currently decay, and is nontoxic, but additionally it's as strong as steel, and only slightly more compliant. This offers a number of possibilities for lightweight, free structures in furniture and the like.

Topics

Minimum hardware and software to get a flexible notetaking device running

Kragen Javier Sitaker, 2017-04-28 (4 minutes)

I was thinking about the minimum hardware and software needed to get a flexible notetaking device working on an AVR microcontroller or something similar. There are a variety of old Nokia cellphone displays out there that can be accessed via SPI, such as http://articulo.mercadolibre.com.ar/MLA-647101617-lcd-nokia-5110-_JM (84×48 pixels, monochrome, SPI, AR$99). PS/2 keyboards (http://articulo.mercadolibre.com.ar/MLA-624534188-teclado-everte-c-kb-12u-bk-ps2-teclas-resistentes-compacto-_JM AR$59, a few blocks from my house) are easy to interface. Supposedly 4GB MicroSD cards are easy to interface via SPI (http://articulo.mercadolibre.com.ar/MLA-648833744-memoria-micro-sd-4gb-micro-sdhc-nueva-sandisk-micro-hc-_JM $49). And the ATMega328 is readily available (http://articulo.mercadolibre.com.ar/MLA-609412290-atmega328-atmega328p-pu-dip28-bootloader-arduino-nubbeo-_JM#D[S:ADV,L:VQCATCORE_LST,V:3] $70).

So the basic hardware costs (+ 99 59 49 70) = AR$277 = (/ 277 15.8) = US$17.53. Plus batteries I guess. And the result has 105 full-sized keys, (* 84 48) = 4032 pixels (8 lines of 24 4×6 characters, say, although originally it was used for 4 lines of 13 characters), and 4 gigabytes of permanent storage — storage you can remove and read and write elsewhere!

I actually already have an old PS/2 keyboard lying around, reducing the cost to AR$218. It weighs about 800 g, and there's a space containing the LEDs with no keys of about 85×45mm where maybe the LCD could be mounted. The Nokia 3110 from 1997, featuring the display I mentioned above, is only 45 mm wide, so the screen must be about 40×25 mm.

But what do you do about the software? At a minimum you need a font, some screen update logic, editor buffer update logic, search, and probably some kind of snapshotting, ideally using the firesystem other machines will expect on the SD card.

The ATMega328 has 32K of Flash and 4K of RAM. You could probably fit all the code necessary for the editor and the LCD, filesystem, and keyboard drivers into that 32K. Ant's Editor, a simple buffer-gap vi clone written for the IOCCC around 1993, is a bit under 3K compiled to amd64 machine code. But you probably want to do paging from the SD card as much as possible; in particular you don't want the 4K of RAM to have to contain the whole document you're editing.

None of these three devices requires extremely tight timing. The LCD is clocked from the microcontroller (its five lines are reset, chip select, clock, data, and "data/command", of which I think the last three can be multiplexed with other things as long as chip-select is

held high), the SD card in SPI mode is also clocked from the microcontroller (ideally at 20 megabits per second, 2.5 megabytes per second, using four lines: clock, data in, data out, and chip select, so again it only needs one dedicated pin), and the keyboard generates its own clock but only clocks at 10–16.7 kHz, which means that each phase of the clock lasts 30–50 µs (according to http://www.computer-engineering.org/ps2protocol/). That means that, although there is a deadline, the deadline is very generous: 10 or 20 microseconds, which is 160 to 320 clock cycles at 16MHz, 200 to 400 at 20MHz, or 80 to 160 at 8MHz if I want to save myself a crystal. And all we have to do at that time is sample the DATA line.

Also, if we miss the deadline, all that happens is that we miss a keystroke.

The whole microcontroller interface needs eight pins: the LCD reset line (maybe optional), two chip select lines for the LCD and SD card, three clock and data lines shared between the LCD and SD card, and two lines for the keyboard. That leaves the other 15 GPIO pins of the ATMega328 free.

Topics

How to get 6 volts out of a 7805, and why you shouldn't

Kragen Javier Sitaker, 2019-06-08 (updated 2019-06-10) (8 minutes)

Lots of discarded electronics has things like 7805s and white LEDs in it. White LEDs need about 3 volts to light, and the bright illumination ones have a substantially less exponential V–I curve than ordinary indicator LEDs (because of a larger ohmic resistance component and special magic I don't understand to prevent thermal runaway) so it's typical to hook two of them in series up to a 6-volt power supply. But they're not *so* ohmic that it's a good idea to hook them up to, say, a 9-volt or 12-volt supply, or that they'll work on a 5-volt supply. (8 volts — 4 volts per LED — does seem to work with the 6-volt lighting panels I picked up off the sidewalk. At 12 volts — 6 volts per LED — my 12-volt 500-mA switching power supply detects an overload and turns off before burning out any LEDs.)

There *is* a 7806 6-volt voltage regulator, but it's much less common than the 7805 and the 7812. But it's not actually that hard to get a regulated 6 volts out of a 7805.

The 7805

The 7805 is a three-pin regulator which reduces, according to Fairchild's datasheet, 7–35 Vdc to 5 Vdc ±4% at up to 1 A (1.5 A from TI) as long as it's below 125° (150° from TI), with 5°/W of thermal resistance from the junction to the case (3°/W from TI), and so with adequate heatsinking it can dissipate 20 watts or more. It regulates its output pin to 5 V above its ground pin, which sources up to 8 mA. Bypassing with 0.33µF on input and 0.1µF on output is suggested but not generally needed.

The hack

The idea is that you float the 7805's "ground" pin 1 V above the real ground, using a 5:1 voltage divider. This is shown on p. 23 of Fairchild's datasheet. If we're satisfied for the 8 mA regulator current through the voltage divider to produce an error of 0.2 V, which is comparable to the 4% error of the regulator itself, the divider circuit needs to be fairly low resistance: 25 Ω down to real ground and 125 Ω up to the output pin. The 125-Ω resistor will be dissipating 200 mW, so you need to use at least a ¼-watt resistor. (Alternatively, you can use an op-amp buffer to set the "ground" voltage for lower power consumption and error, as shown on p. 24.)

In this case, though, you could probably tolerate a larger error and use, say, a 100-Ω resistor and a 330-Ω resistor. 15 mA through the 330-Ω resistor plus 4–8 mA of bias current through the ground pin put 19–23 mA and thus 1.9–2.3 V of offset on the 100-Ω resistor, giving 6.9–7.3 V on the output, or 6.7–7.5 V if we include the 7805's own error too.

Power dissipation

If 500 mA (a guess) is running through the 6-volt LED strings,

they'll be dissipating 3 watts, which is not a big problem since they're spread out over a large area. If the 7805 is dropping 12 volts (–1) down to 5 volts, it will also be dissipating 3 watts. Though it's specced to operate at 125°, I'd kind of want to keep it below 70° so it will last longer and I'm not at risk of burning my hand on it, and in environmental temperatures up to 35° here in Buenos Aires, that only leaves $\Delta T = 35°$, so we only have a thermal resistance budget of 11.7°/W, of which the junction-to-case resistance already eats up 5°/W. So we need 6.7°/W or less between the heatsink and its coupling to the TO-220 case.

The heatsink tab is, for better or for worse, connected to the package's ground pin, since that's the most negative voltage in the circuit.

In 2019, you should probably just use PWM for multi-watt loads, though

A transistor switch (maybe controlled with a second transistor), a small inductor and a small capacitor for an LC filter, a microcontroller, and possibly a resistor or two for analog feedback from the output is a better solution. You can totally run the microcontroller off your 7805, and you get 95+% efficiency (instead of the 50% described here). The ±4% precision of the 7805 may or may not be available from a microcontroller without trimming, but it's plenty good enough for this. For example, the ATMega328P datasheet specifies its internal bandgap voltage reference as 1 V ±10%, and its ADC contributes about an additional ±1% error; the STM32F103xx is specified to be better with only ±3% error on its reference and ±0.3% ADC error.

Additionally, this gives you programmability and dimming up to a few hundred kHz for free. The dimming is only voltage dimming, and so highly nonlinear for LED illumination.

The downside of this is that you lose the overheating, current-limiting, and foldback protections built into the "virtually indestructible" 7805.

LEDs can general tolerate higher pulsed power than constant power, so the LC filter might seem unnecessary. But if you're running off a 12-volt or 19-volt DC supply, I'm not confident that even short current pulses through the LEDs won't destroy the LEDs or the switching transistor, and of course the shortness of the pulses is a function of the microcontroller software.

Power dissipated by PWM

If you're dimming at 200 kHz via a IRLML6402 (a 40¢ P-channel power MOSFET rated for 20 V and 3.7 A; see My attempt to learn about jellybean electronic components (p. 1974)), you're dumping 12 nC from its gate to ground every time you turn it on; that might be about 12 volts. That's 2.4 mA of gate switching losses at 12 V: 30 mW, about 100× less than the linear 7805. Its 65 mΩ on-resistance will dissipate another 16 mW at 500 mA. So you don't gain much from using a substantially lower frequency like 20 kHz, right?

No, wait — the datasheet also lists a 48 ns rise time and a 381 ns fall time, so each 5-µs-period pulse includes an 0.43-µs transition time during which the on-resistance is significant. If we simplify its

behavior to a linear ramp up of current from 0 to 500 mA during that time, while its drain–source voltage linearly drops from 12 V to 0, its linear power consumption follows a parabola from 0 to 0 during that time, with a peak at 1.5 W in the middle. So it might also have some 90 mW or so of switching losses just from being slow. (Its real behavior is somewhat more complex, but that toy model is probably adequate for our purposes here.)

So, although power consumption is acceptable at around 125 mW, you actually might improve it substantially by dropping to 100 kHz or 50 kHz or something, or using a beefier MOSFET.

Topics

Low-carbohydrate diets are ecologically sustainable

Kragen Javier Sitaker, 2018-04-27 (2 minutes)

Quite aside from the question of whether low-carbohydrate diets are good for people, bad for people, or good for some people and bad for others, there's the question of whether they're economically sustainable — agriculture produces much more energy stored as carbohydrate than in other forms. So, some people have claimed that there isn't enough land in the world for everyone to survive on a low-carbohydrate diet.

Soybean yields average 2.93 metric tons per hectare per year; world arable land is currently a bit over 48 million km², which is 4.8 billion hectares, about 11% of the world's land area. Dry soybeans are 37% protein by weight. Multiplying these three figures together, if all current arable land were continuously planted in soybeans at current average soy yields, the world would produce 1.4×10^{13} kg of soybeans per year, containing 5.2×10^{12} kg of protein, 680 kg of protein per person, or 1.87 kg per person per day — about 7500 kcal of protein per person per day.

However, soybeans are also 18% fat by weight, which would add another 910 g of fat per day per person, 8200 kcal, for a total carbohydrate-free yield of 15700 kcal. This is about six or seven times the human daily caloric requirement.

It seems likely that the yields from other arable land not currently cultivated in soy would be lower than current average soy yields, and also that sustainable farming practices would reduce yields modestly further. However, it seems unlikely that this would reduce yields by the factor of six that would be needed to make a zero-carbohydrate diet economically unsustainable.

Of course, soybeans also contain carbohydrates (30% by weight), but you can remove those before you eat them.

The reason that agriculturally produced food appears scarce is that most of it is fed to livestock, which converts most of it into manure. There are more efficient ways of removing carbohydrates from soybeans than feeding it to chickens or cows.

There may be a way to produce more protein per hectare than growing soybeans, but I don't know of it; the reason soybeans are currently such an important crop is that they are the favored protein source for fattening up livestock.

Topics

Notes on reading eForth 1.0 for the 8086

Kragen Javier Sitaker, 2007 to 2009 (5 minutes)

These are notes on the original 8086 eForth model, eForth 1.0 by Bill Muench and C. H. Ting, 1990.

The assembly-language parts of eForth are:

- the colon header on each colon word: NOP; CALL DOLST
- the $NEXT macro: LODSW; JMP AX
- the boot code ORIG, 14 instructions
- The following 31 primitive FORTH words; their instruction counts include $NEXT where applicable:

```
;   BYE     ( -- )
;       Exit eForth.
        1 instruction.

;   ?RX     ( -- c T | F )
;       Return input character and true, or a false if
;       no input.
        16 instructions.

;   TX!     ( c -- )
;       Send character c to the output device.
        8 instructions.

;   !IO     ( -- )
;       Initialize the serial I/O devices.
        2 instructions.

;   doLIT   ( -- w )
;       Push an inline literal.
        4 instructions.

;   doLIST  ( a -- )
;       Process colon list.
        6 instructions.

;   EXIT    ( -- )
;       Terminate a colon definition.
        5 instructions.

;   EXECUTE ( ca -- )
;       Execute the word at ca.
        2 instructions.

;   next    ( -- )
;       Run time code for the single index loop.
;               i.e. FOR-NEXT, usually known as DO-LOOP.
        9 instructions.
```

```
;   ?branch ( f -- )
;       Branch if flag is zero, sometimes called _IF or
;       (IF)
        9 instructions.

;   branch  ( -- )
;       Branch to an inline address, sometimes called
;       _ELSE or (ELSE).
        3 instructions.

;   !       ( w a -- )
;       Pop the data stack to memory.
        4 instructions.

;   @       ( a -- w )
;       Push memory location to the data stack.
        4 instructions.

;   C!      ( c b -- )
;       Pop the data stack to byte memory.
        5 instructions.

;   C@      ( b -- c )
;       Push byte memory location to the data stack.
        6 instructions.

;   RP@     ( -- a )
;       Push the current RP to the data stack.
        3 instructions.

;   RP!     ( a -- )
;       Set the return stack pointer.
        3 instructions.

;   R>      ( -- w )
;       Pop the return stack to the data stack.
        4 instructions.

;   R@      ( -- w )
;       Copy top of return stack to the data stack.
        3 instructions.

;   >R      ( w -- )
;       Push the data stack to the return stack.
        4 instructions.

;   SP@     ( -- a )
;       Push the current data stack pointer.
        4 instructions.

;   SP!     ( a -- )
;       Set the data stack pointer.
        3 instructions.

;   DROP    ( w -- )
```

```
;       Discard top stack item.
        3 instructions.

;   DUP     ( w -- w w )
;       Duplicate the top stack item.
        4 instructions.

;   SWAP    ( w1 w2 -- w2 w1 )
;       Exchange top two stack items.
        6 instructions.

;   OVER    ( w1 w2 -- w1 w2 w1 )
;       Copy second stack item to top.
        4 instructions.

;   0<      ( n -- t )
;       Return true if n is negative.
        5 instructions.

;   AND     ( w w -- w )
;       Bitwise AND.
        6 instructions.

;   OR      ( w w -- w )
;       Bitwise inclusive OR.
        6 instructions.

;   XOR     ( w w -- w )
;       Bitwise exclusive OR.
        6 instructions.

;   UM+     ( w w -- w cy )
;       Add two numbers, return the sum and carry flag.
        9 instructions.
```

That's 157 instructions worth of primitives in all, in 31 primitives, plus 14 more instructions in the boot code. The rest of the system is written in Forth. It would be about 31 fewer instructions if there were a central NEXT instead of an indirect jump on the end of each word.

I don't have an assembler that can assemble it or a disassembler that can disassemble the compiled version, but if it's similar to Bill Muench's updated "8086 eForth ITC16i 971014.1", there should be about 4.3 bytes per instruction, which would make this a 675-byte kernel of an interpreter. (That later Forth also has CHAR+, CHAR-, CHARS, CELL+, CELL-, CELLS, and REDIRECT as primitives, but omits lower-case next.)

The 161 high-level FORTH colon definitions in EFORTH.SRC are another 3078 words of text (according to wc) and so are probably about another 6000 bytes. There's an 8814-character span of NULLs in the middle of the .COM file, which totals 15600 bytes, leaving 6786 bytes that might be meaningful; this is pretty close to my estimate of 6675 bytes. (Which leaves out the user variables and so on.)

If we use token threading, 3078 words of text are probably closer to

3000 bytes, but the token table is another 1024 bytes.

To implement the I/O stuff on Forth on Linux, we'd probably want to implement a "syscall" word in machine language taking up to 6 arguments (because we need select() for ?RX, and it takes five arguments) that makes an up-to-5-argument system call. Apparently the parameters go into %ebx, %ecx, %edx, %esi, and %edi, in that order, according to my disassembly of select.o from my libc.a.

In a direct-threaded system, there's a few bytes extra penalty for colon definitions --- probably five.

"unnecessary" primitives here include: "next" (the lowercase looping one) R> 1- >R I 0= (IF) R@ R> DUP >R or : R@ R> R> TUCK >R >R ; over : over >R DUP R> SWAP ;

That is, it would be possible to avoid defining them as primitives.

The 24 different instructions used are NOP, CALL, LODSW, JMP, MOV, CLI, STI, INT, CLD, IRET, XOR, JZ, OR, JNZ, PUSH, POP, CMP, XCHG, JC, SUB, ADD, CWD, AND, and RCL.

Topics

- Small is beautiful (p. 3714) (40 notes)
- Assembly language (p. 3328) (25 notes)
- Forth (p. 3461) (19 notes)

Fast secure pubsub

Kragen Javier Sitaker, 2019-02-04 (updated 2019-12-03) (2 minutes)

Suppose you want to implement a fast, secure publish-subscribe ("pub-sub") service. What if you package your message transformation and filtering code into individual processes that run and produce a result?

The idea is that Turing-complete rules provided by the sender, the receiver, and possibly intermediate routing entities run in very-short-lived ephemeral processes which have relevant information mapped into their memory space. Even on Linux, it's possible to spawn off 7000 processes per second; we should be able to do better with a custom kernel, and for example Fastly's Lucet WebAssembly runtime "can instantiate WebAssembly modules in under 50 microseconds", thus potentially permitting the creation of some 20,000 "processes" per second per core. Processes that exceed their CPU allocation are ruthlessly killed.

Processes running sender-supplied code can be run with access to information about potential recipients to prevent insensitive recipients from seeing sensitive information, and then they can be killed to prevent them leaking information about the potential recipients either to the sender or to other recipients. Processes running receiver-supplied code can inspect relevant aspects of the message to decide whether or not to pass the message along — either whole or in some summarized form.

If access to the message is provided not via memory-mapping but via some kind of recordable API, a recipient-provided Turing-complete selection function that is careful not to inspect more message fields than needed can implicitly produce a memoized filter rule in a supervisory process. For example, if the filter function inspects a "newsgroup" field, finds that its value is "alt.sex", and then rejects the message without inspecting further fields, then the supervisory process can memoize a filter rule: `[{"newsgroup": "alt.sex"}, "reject"]`. Perhaps it can construct a trie on known values of "newsgroup" and only actually invoke filter functions whose results are not already recorded in the trie — perhaps they inspected an additional field, for example, or perhaps there are values of "newsgroup" that have not previously been seen.

This is very similar to how transactional memory observes the reads being executed by the code in a transaction in order to be able to safely detect update conflicts.

I think this is a particular application of Umut Acar's "self-adjusting computation".

Topics

- Performance (p. 3621) (149 notes)
- Systems architecture (p. 3691) (48 notes)
- Operating systems (p. 3608) (18 notes)
- Transactions (p. 3755) (14 notes)

Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels

Kragen Javier Sitaker, 2019-08-31 (updated 2019-09-08)
(28 minutes)

On the bus, thinking about vision, there occurred to me a simple way to convolve an image with a fairly precisely approximated (real) Gabor filter or Morlet wavelet, in fixed point, with no multipliers, and a cascade of a small number of sparse FIR and IIR filters. This is probably known, but not to me.

Gabor filters

The general form of a Gabor wavelet in two dimensions has five parameters: an angle θ, a wavelength λ, a phase ψ, a radius σ, and an aspect ratio γ.

WIKIPEDIA SAY

$$g(x, y;\ \lambda, \theta, \psi, \sigma, \gamma) = \exp(-(x'^2 + \gamma^2 y'^2)/(2\sigma^2)) \exp(i(2\pi x'/\lambda + \psi))$$

where

$$x' = x \cos\theta + y \sin\theta$$
$$y' = -x \sin\theta + y \cos\theta$$

Here the first exp gives you the Gaussian envelope and the second exp gives you the oscillation. As you might or might not guess, its Fourier transform is *also* a Gabor wavelet.

You convolve this thing with an image and it detects edges at its given angle at or near its given frequency, in the area selected by its Gaussian window, which is nearly zero when you reach displacements of several times σ, or σ/γ^2 in the y' direction.

Sparse approximations

As mentioned in Sparse filters (p. 834), I'm looking for ways to get good approximations of convolution filters using networks of small, sparse kernels, to reduce the total amount of computation. In particular I'm interested in Hogenauer filters, which is discussed in Some speculative thoughts on DSP algorithms (p. 2590), Recurrent comb cascade (p. 483), and Cheap frequency detection (p. 3026), The Bleep ultrasonic modem for local data communication (p. 966), and similar things. In Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) I describe how you can use precise resonators that are not only non-FIR but non-BIBO to very efficiently compute precise finite infinite response filters, and I'll use that here.

I think a sparse Gabor-filter approximation is particularly interesting because of the filter's importance and because of its relatively great computational cost. (But maybe someone already knows this algorithm.)

Overview

The filter is a composition or pipeline of five major stages, each composed of a cascade of small, sparse linear filters. Because of the convenient properties of convolution, the order of all of the individual pieces can be reordered as desired, and in particular if you're computing several different Gabor filters over the same image with some parameters in common, it may be desirable to move stages that are in common between them to the beginning of the processing pipeline.

The stages are a resonating feedback comb to create the oscillations, an oscillation window to confine them to a Gaussian along the direction of oscillation, a low-pass filter along the direction of oscillation to remove frequencies higher than the desired frequency, an antialiasing filter to deal with pixels a fraction of a sample off the oscillation axis, and a transverse window blur to spread the impulse response along a Gaussian at right angles to the direction of resonation.

In what follows, I will suppose that the angle θ can be adequately approximated as a ratio of small integers, as examples of which I will use 3 and 4: we are looking for waves whose phase varies not at all in the direction $(x+3, y-4)$, and varies fastest in the direction $(x+4, y+3)$. Below I will refer to this (4, 3) displacement as the "stride". And I will suppose that the wavelength λ we're looking for can be adequately approximated as an integer multiple of *twice* that displacement; that is, at some integer n, $(x+8n, y+6n)$ has a phase precisely 2π advanced from the phase at (x, y). There are some tricks to handle waves of other wavelengths, but they are not as well developed.

The resonating feedback comb and oscillation window

To window the resonance over a given distance, we first use a feedback comb filter whose impulse response is an infinite non-decaying oscillation:

$$y[i, j] = x[i, j] - y[i-4n, j-3n]$$

That is, the pixel (i, j) of the output y is the corresponding pixel $x[i, j]$ of the input, minus a *previous output pixel* positioned exactly half an oscillation away along the line of oscillation, at $(i-4n, j-3n)$. You can easily verify that the impulse response of this filter is an alternating sequence of positive and negative impulses leading away from the impulse in the desired direction of oscillation.

To tame this wildly unstable filter, we simply use a window over some finite number m of oscillations, a *feedforward* comb filter; here our x input is the y output of the previous filter:

$$y[i, j] = x[i, j] - x[i - 8nm, j - 6nm]$$

When using exact arithmetic, the composition of these two filters has a finite impulse response; in its impulse response, the alternating positive and negative impulses generated by the first filter continue on for m oscillations, then are canceled completely and precisely by the subtraction.

This alternating impulse train contains our desired spatial frequency as well as all of its odd harmonics, and the second filter just brutally windows it with a rectangular window after some integer number of oscillations. The first, feedback, comb filter plays the role

of the integrator in a Hogenauer downsampling CIC filter, while the second, feedforward, comb plays the role of the comb in the Hogenauer filter.

Unlike the integrator in a Hogenauer filter, the response of the feedback comb at DC is identically 0, and its amplification factor for frequencies that don't precisely align with its resonant frequency is finite; it only fails to act BIBO if the input data actually does contain nonzero energy at precisely the resonant frequency (necessarily over infinite space, at least in one direction). So you might be able to get reasonable results using floating point, but there are no guarantees there.

That window doesn't look very Gaussian yet, so to fix that, we repeat the process two or three more times, for a total of six to eight atomic kernels each consisting of a single subtraction. The window will have the most Gaussian shape if m remains the same in all of the stages, but this trades off against undesirable quantization in the available window sizes.

This is one of the places where you have a little flexibility to go beyond the limitations of the basic method; you can use resonators in slightly different directions and frequencies to get an intermediate overall direction and frequency of oscillation. But the windowing functions must be matched to those precise strides to prevent the overall system impulse response from becoming infinite.

The oscillation-axis low-pass filter

The above is not yet satisfactory for two reasons. First, we still have the odd harmonics to deal with — the third, fifth, seventh, and so on — which are exactly the same amplitude as the fundamental, since it's an impulse train. Second, aside from those harmonics, looked at precisely along the oscillation axis, the two stages described above give us a very nice Gabor wavelet, but at any other angle it looks like an impulse — all frequencies pass unchanged.

For these two purposes, we need to do some low-pass filtering, some of which is transverse to the oscillation axis and is what gives us the roundness or ellipticity of our two-dimensional Gaussian window, and some of which is parallel to it. And in particular, if it's possible, we may want to use another feedforward comb filter to stab the third harmonic in the heart precisely, because it's going to be the most troublesome harmonic to handle with general-purpose low-pass filtering, since it's only 1.58 octaves above the fundamental, and a filter with its first precise zero at that harmonic will suppress it completely, as well as most of its spectral leakage. The fifth harmonic is 2.3 octaves above the fundamental, so a generic low-pass filter can separate it from the fundamental pretty easily.

Our fundamental has a period $(8n, 6n)$, so our third harmonic has a period $(⅓8n, ⅓6n)$. We can either subtract pixels at this displacement or add pixels at half this displacement $(¼4n, ⅓3n)$ to completely suppress the third harmonic. If these offsets are not precise, for example because $4n$ isn't divisible by 3, the suppression won't be precise either, and a lot more of the third harmonic will survive to be dealt with by the other more generic low-pass filtering. (But we'll get a little bit of bonus transverse low-pass filtering.)

So the simplest form of our heart-stabbing filter would look like this:

$y[i, j] = x[i, j] + x[i - \frac{1}{34}n, j - \frac{1}{33}n]$

That taken care of, we can proceed to the oscillation-axis low-pass filter, which is mostly important if n is larger than 2 or 3. Suppose $n = 5$; now a period of the full oscillation is (40, 30). We can use an orthodox Hogenauer filter along the axis of oscillation to suppress harmonics higher than the third; first an integrator (calculating a prefix sum along each strided diagonal, also known as a sum table, scan, or integral image):

$y[i, j] = x[i, j] + y[i - 4, j - 3]$

and then a feedforward comb of, for example, two strides:

$y[i, j] = x[i, j] - x[i - 8, j - 6]$

This amounts to a rectangular window, whose frequency response is a sinc; its first null is where a full oscillation fits precisely into the window, which in this case would be a period of (12, 9). It has a 6 dB per octave frequency rolloff.

We probably want a better rolloff than that; if we repeat it two more times, we get 18 dB per octave, which attenuates the fifth harmonic by almost 42 dB, and higher harmonics by more.

The usual CIC-filter concerns about passband flatness don't apply here, since we are only trying to select a single frequency and frequencies very close to it.

At higher spatial frequencies, as long as the stride is a pair of integer number of pixels, the higher harmonics disappear because they alias harmlessly back down into the lower harmonics.

So the oscillation-axis low pass filters end up being seven more filter stages each consisting of a single addition or subtraction. But, as we see in the next section, we will reduce this to five.

The antialiasing filter

So far, pixels have only ever been combined with other pixels at a multiple of the basic stride (4, 3) from them. This means that every pixel in the first three lines of an image is in a separate, noninteracting signal, so far; even very high-frequency components will survive and may be aliased down. We'd like to sort of "fill in" the other pixels, at least along the axis of oscillation, rather than skipping over them completely as if they belonged to an entirely different image.

There are a variety of different ways this can be achieved. For example, we could use a couple of simple feedforward combs to get reasonably good fill-in without widening the line of the OTF much:

$y[i, j] = x[i, j] + x[i-1, j-1]$
$y[i, j] = x[i, j] + x[i-2, j-1]$

But let's not do that; we can perhaps get a bit more mileage out of the antialiasing filter in the case where the oscillation wavelength and both dimensions of the Gaussian window and are much larger than this; we could use, for example, a simple Gaussian blur, which also takes some of the load off the low-pass filter along the oscillation axis. Again, this can be done as an orthodox Hogenauer or CIC or box filter, but this time on the usual pixel rows and columns:

$y[i, j] = x[i, j] + y[i-1, j]$
$y[i, j] = x[i, j] - x[i-8, j]$
$y[i, j] = x[i, j] + y[i, j-1]$
$y[i, j] = x[i, j] - x[i, j-6]$

The cascade of those four filters has an impulse response of an 8x6

constant-1 rectangle, and in particular it has the same low-pass effect along the axis of oscillation as the $-x[i - 8, j - 6]$ filter proposed in the section above, as well as providing antialiasing fill-in along the axis of oscillation and some amount of transverse window. If you were to iterate this filter two more times you would have a second-order approximation to a squished Gaussian, but let's not — let's just run it once more and be satisfied, and reduce the high-pass filtering from the previous section by one filtering stage.

So this stage is a cascade of eight tiny kernels, each consisting of a single addition or subtraction.

Since this, in effect, reduces the resolution of the image, it might be wise to do it early on in the pipeline and then decimate the image so that later stages can run much faster, operating on a reduced-resolution image.

The transverse window blur

So at this point our impulse response is a fairly precise sinusoidal oscillation along the correct axis, with a fairly precise Gaussian envelope along that axis, and some sort of relatively crude smooth falloff about 10 pixels to either side of that axis. Now we want to widen out that transverse axis into a Gaussian envelope, either round or elliptical; the existing falloff may help us a bit, but we still need to widen it out considerably.

We can do this with, again, a strided CIC filter, consisting of a cascade of an integrator and a feedforward comb, but this time along the transverse axis, using a stride rotated 90 degrees:

$$y[i, j] = x[i, j] + y[i - 3, j + 4]$$
$$y[i, j] = x[i, j] - x[i - 3p, j + 4p]$$

Here p gives the number of strides in the width of (one stage of) our window, as nm gave the dimension of the Gaussian window in the perpendicular direction.

Because of the existing falloff, we may be able to get away with one more stage of this CIC filter at this point, but we'll probably need two more.

The direction of blurring can be chosen as either (-3, 4) or (3, -4); I chose the first here in order to use previous scan lines rather than previous columns, in the interest of making pipelining possible (see below.)

So implementing the transverse window requires another six stages, each consisting of an addition or subtraction.

Summary

The overall Gabor filter, then, requires a cascade of around $8 + 5 + 8 + 6 = 26$ stages, each performing a single integer addition or subtraction, followed by some final scaling by a constant. This is quite small compared to the millions of pixels in the support of the approximate Gabor wavelet that is the system's finite impulse response, although downsampling the image on its way into the pipeline would reduce this disparity somewhat.

Python smoke test of the algorithm

I did try it in IPython tonight (notebook viewer), and got a pretty round-looking kernel with $n = 5$, $m = 3$, $p = 30$, and the (4, 3) stride suggested above. In plots it looks just fine, but of course that's not

strong evidence. I haven't calculated its error (that would require figuring out the diameters of the Gaussians), but I estimate that in essentially this form it should be able to deliver worst-case errors of less than 1% (-40 dB) and average-case errors that are smaller still.

The code in the notebook boils down to something like this very crude code, with all but the last plot stripped out:

```
from numpy import zeros, dtype
from matplotlib import imshow, colorbar

n = 5
m = 3
p = 30

impulse = zeros((700, 700), dtype=dtype('float16'))
impulse[300, 100] = 1.0
ir1 = impulse.copy()
for i in range(4*n, len(ir1)):
    ir1[i, 3*n:] -= ir1[i-4*n, :-3*n]
fr1 = ir1.copy()
for i in range(8*n*m, len(fr1)):
    fr1[i, 6*n*m:] -= ir1[i-8*n*m, :-6*n*m]
ir2 = fr1.copy()
for i in range(4*n, len(ir2)):
    ir2[i, 3*n:] -= ir2[i-4*n, :-3*n]
fr2 = ir2.copy()
for i in range(8*n*m, len(fr2)):
    fr2[i, 6*n*m:] -= ir2[i-8*n*m, :-6*n*m]
ir3 = fr2.copy()
for i in range(4*n, len(ir3)):
    ir3[i, 3*n:] -= ir3[i-4*n, :-3*n]
fr3 = ir3.copy()
for i in range(8*n*m, len(fr3)):
    fr3[i, 6*n*m:] -= ir3[i-8*n*m, :-6*n*m]
hs = fr3.copy()
for i in range(int(4*n/3), len(hs)):
    hs[i, 3*n/3:] += fr3[i-int(4*n/3), :-3*n/3]
ai1 = hs.copy()
for i in range(4, len(ai1)):
    ai1[i, 3:] += ai1[i-4, :-3]
ff1 = ai1.copy()
ff1[8:, 6:] -= ai1[:-8, :-6]
ai2 = ff1.copy()
for i in range(4, len(ai2)):
    ai2[i, 3:] += ai2[i-4, :-3]
ff2 = ai2.copy()
ff2[8:, 6:] -= ai2[:-8, :-6]
bf1i = ff2.copy()
for j in range(1, len(bf1i[0])):
    bf1i[:, j] += bf1i[:, j-1]
bf1c = bf1i.copy()
bf1c[:, 8:] -= bf1i[:, :-8]
bf2i = bf1c.copy()
for j in range(1, len(bf2i[0])):
    bf2i[:, j] += bf2i[:, j-1]
```

```
bf2c = bf2i.copy()
bf2c[:, 8:] -= bf2i[:, :-8]
bf3i = bf2c.copy()
for i in range(1, len(bf3i)):
    bf3i[i] += bf3i[i-1]
bf3c = bf3i.copy()
bf3c[6:] -= bf3i[:-6]
bf4i = bf3c.copy()
for i in range(1, len(bf4i)):
    bf4i[i] += bf4i[i-1]
bf4c = bf4i.copy()
bf4c[6:] -= bf4i[:-6]
tvi1 = bf4c.copy()
for j in range(4, len(tvi1[0])):
    tvi1[:-3, j] += tvi1[3:, j-4]
tvc1 = tvi1.copy()
tvc1[:-3*p, 4*p:] -= tvi1[3*p:, :-4*p]
tvi2 = tvc1 / 256
for j in range(4, len(tvi2[0])):
    tvi2[:-3, j] += tvi2[3:, j-4]
tvc2 = tvi2.copy()
tvc2[:-3*p, 4*p:] -= tvi2[3*p:, :-4*p]
tvi3 = tvc2 / 256
for j in range(4, len(tvi3[0])):
    tvi3[:-3, j] += tvi3[3:, j-4]
tvc3 = tvi3.copy()
tvc3[:-3*p, 4*p:] -= tvi3[3*p:, :-4*p]
imshow(tvc3[200:500, 300:600], origin='lower'); colorbar()
```

Reducing memory usage

As is standard practice, you can pipeline these stages (see Evaluating DSP operations in minimal buffer space by pipelining (p. 321)) so that you don't need 26 modified copies of the entire image floating around in memory (some at increased precision). But doing this in the straightforward way, scan line by scan line, you still need a pretty big buffer to do this in, because some of the 26 stages need to look pretty far back into the past. If we suppose $n = 5$, $m = 3$, and $p = 60$, for example, the number of scan lines of memory needed is as follows:

stage	scan lines	$y[i, j] = x[i, j] +$
resonator 1	15	$-y[i - 4n, j - 3n]$
oscillation window 1	90	$-x[i - 8nm, j - 6nm]$
resonator 2	15	
oscillation window 2	90	
resonator 3	15	
oscillation window 3	90	
resonator 4	15	
oscillation window 4	90	
heart-stabbing filter	5	$x[i - \frac{1}{34}n, j - \frac{1}{33}n]$
axis LPF integrator 1	3	$y[i - 4, j - 3]$
axis LPF comb 1	6	$-x[i - 8, j - 6]$
axis LPF integrator 2	3	

axis LPF comb 2 6
antialias filter 1 7 (4 kernels) $y[i-1,j]$; $-x[i-8,j]$; $y[i,j-1]$; $-x[i,j-6]$
antialias filter 2 7 (4 kernels)
transverse integrator 1 4 $y[i-3,j+4]$
transverse comb 1 240 $-x[i-3p,j+4p]$
transverse integrator 2 4
transverse comb 2 240
transverse integrator 3 4
transverse comb 3 240

Whew. That's 1189 scan lines of memory in total, plus some fractional scan lines I'm not considering. Is there any way to reduce this?

(Well, of course there's decimation. But I mean aside from decimation.)

I thought about tiling. It doesn't help, because you just switch from having to buffer previous scan lines to having to buffer previous rows of tiles. In fact it hurts a little because you can't discard fractional tiles. Maybe there's still a way for it to work but I don't see it.

I thought about maintaining resonator state in a different way, using per-pixel Minsky or Goertzel resonators, spatially shifted by varying integer amounts per scan line, rather than a feedback comb (which is basically a Karplus-Strong oscillator). This might help but it only saves you the memory needed by the resonator, which is relatively small compared to that needed to window the oscillations. And it makes the assertion about the precise cancellation of windowing more dubious, since I don't think there's a precise way to calculate Minsky or Goertzel resonation. (See Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679).)

I thought about using more stages for the transverse window. That will give you a more precise Gaussian but uses more memory, not less.

If you try to use exponential blur for the transverse blur instead of an honest Gaussian blur, you completely lose not only finite impulse response but also zero-phase behavior. To get zero-phase behavior back you need to do a second pass backwards, which requires keeping the whole result image in memory instead of just part of it.

Multiple Gabor sharing

So, if you're going to do a bunch of Gabor convolutions on the same image, what parts of the pipeline should you share between them?

The most computationally intensive (heh) parts of the pipeline are those dependent on the frequency and the angle: the resonating feedback comb, the oscillation-axis low-pass filter, and (more loosely) the antialiasing filter. These depend on λ and θ, but not σ or γ. It might make sense to move these stages, as well as the transverse blur's integrators, to the beginning of the pipeline so that they can be shared between different window sizes and shapes, running the oscillation window and the transverse blur window later, which are only six of the 26 stages. This probably is not practical to do in floating-point because of the large magnitudes needed to feed the Hogenauer-style

cascade of six windowing combs.

The antialiasing filter is applicable over an octave or so of frequencies at any angle (although in the above example it's a bit longer in one direction than the other), and the transverse-blur integrators are applicable for any frequency at a given angle. They can't both go first, though; putting the antialiasing filter first is probably better because it allows you to decimate the image immediately.

Dealing with invalid pixels ("NA")

Sometimes we can identify certain pixels as containing invalid data — for example, they don't work on the sensor, or they're suffering salt-and-pepper noise in this frame, or they're saturated (perhaps due to a lens flare). Some numerical-computation environments have special facilities for such situations; Octave and R have "NA", while Numpy has "masked arrays" which support most of the same operations as ordinary arrays.

Dealing with such invalid pixels is a particularly tricky problem for FFT-based convolution algorithms, since the structure of the FFT doesn't have a reasonable way to incorporate validity information. R's `fft` function, for example, will simply return an array of NA values if asked to transform an array with a single NA value in it. Octave, by contrast, returns an array whose values are NA only in the phase or phases affected by the NA value — so they may have a NA real magnitude, a NA imaginary magnitude, or both. In either case, you can't use FFT convolution on a signal containing even a single invalid pixel; the signal comes back entirely NA.

By contrast, a direct implementation of convolution has three straightforward ways to handle NA values: it can propagate them to the affected neighborhood, turning each single NA pixel into a giant NA hole in the result; it can omit the NA pixels from the weighted sum, increasing the weights on the other pixels to compensate, unless all pixels with nonzero weight are NA; or it can switch between these two strategies at some threshold of invalidity, such as 50%. (Both Octave's `conv` and R's `filter(method='convolution')` take the first approach.)

This kind of sparse filtering using two-input kernels could take any of these three approaches, but in the recursive case (integrators) any of them would lead rapidly to disaster. Probably in that case the least bad approach is to treat NA pixels as 0 in recursive filters.

Conclusions

Well, really, beginnings... but it seems like this approach to approximating convolution with a Gabor wavelet probably works and probably is efficient (although my IPython notebook prototype certainly is not). It provides obvious ways to set five of the six parameters, but not ψ, the phase offset. This is a little alarming because without the phase offset there's no way to get the *complex* Gabor wavelet. It would be surprising if there turned out to be no way to do this, but none is obvious to me at the moment.

(The issue is that you want to phase-shift the oscillation by a quarter cycle, but without moving the Gaussian window along with it. Maybe the solution could be something as simple as a differentiator in the direction of oscillation, perhaps by a half cycle,

and a compensating constant factor.)

Among the possible applications are approximating an arbitrary convolution kernel as a sum of Gabor wavelets. One possible approach to this is analyzing it in different directions and at different scales using the Gabor transform or the one-dimensional wavelet transform with the Morlet wavelet, choosing a sparse subset of these basis functions that could possibly approximate the target kernel well, then optimizing the weights of that sparse subset further, under a loss function that drives small weights toward zero.

Moreover, there's no particular reason to limit yourself entirely to Gabor wavelets when optimizing that approximation; you can include other kernels that can be computed efficiently using such sparse cascades, such as separable kernels (see The miraculous low-rank SVD approximate convolution algorithm (p. 747)), flat and especially polygonal kernels (see Real-time bokeh algorithms, and other convolution tricks (p. 2661)), and, using the techniques used above to propagate the Gabor's oscillation along a line, line segments.

Another possible application is the design of transmission-line RF filters, including stripline/microstrip transmission lines built into printed circuit boards, and waveguide filters — though I'm not sure how much that will buy you, given the need for matching networks to interconnect the signal paths. An open-ended transmission line spur is in some sense a unity-positive-weight feedforward comb filter whose lag is twice the length of the line, while a closed-ended one amounts to a unity-negative-weight feedforward comb filter, similarly.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Graphics (p. 3483) (91 notes)
- Math (p. 3564) (78 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Prefix sums (p. 3645) (18 notes)
- Convolution (p. 3391) (15 notes)
- Sparse filters (p. 3725) (11 notes)
- Approximation (p. 3321) (2 notes)
- Gabor

Replicating education

Kragen Javier Sitaker, 2017-07-19 (7 minutes)

Education is broken; people don't learn much, it takes them a very long time to learn it, they forget most of it, and then they don't generalize. It's tempting to attribute this to people being stupid, and clearly that's a condition of the problem space, but it's no excuse for not solving the problem.

Lectures are terrible

The medieval lecture model we use is partly to blame. It originated as a kind of mass production for books: the reader reads from one copy of the book while the audience notes down verbatim everything they say. As manual methods of making many copies of a text go, this is quite efficient, but once the number of copies goes past 500 or so, the printing press is far more efficient.

Sadly, this model survived the introduction of the printing press and even the xerox, continuing to this day in many classes, in which the pupils waste precious attention they could spend assimilating the material on copying it instead, as if they didn't have smartphones in their pockets capable of copying documents over Bluetooth at hundreds of pages per second.

Nowadays the goal of lectures is not really making copies of written material, but transferring knowledge from the mind of the lecturer into the minds of the students, at its best leavened with some entertainment or practical demonstrations to clarify the abstract concepts. But it is a terribly inefficient way of achieving this kind of knowledge transfer; it's roughly as effective as reading a textbook [citation needed] but consumes the time of the lecturer as well as imposing expensive coordination costs on the lecturer and students, who must not only arrive at the lecture hall together but also pause to think together if they are to have a chance of understanding the material.

(In British universities, the rank below Professor, equivalent to US "professor", is in fact "Reader", followed by "Lecturer", which is a mangled French word that also means "reader".)

As a means of inspiring enthusiasm, however, lectures are almost unmatched; only music, movies, sports, and blood rituals seem to be able to compete, and of these all except music are much more costly. So lecturers often opt for lectures in order to manipulate their audiences emotionally, usually to further the interests of the lecturer (for example, persuade colleagues that their research interests are worthwhile, or to whip up political sentiment) but occasionally in the interests of the audience.

Sadly, nowadays our lectures are quite often afflicted with slide decks full of text. While these are of some communicational benefit when a language barrier separates the lecturer from their audience, far more often, they further reduce the bandwidth of information transfer while vitiating the lecture's most signal virtue, that of inflaming the passions.

Knowledge transfer

Learning depends, of course, on many other factors beyond knowledge transfer. For example, you need problem sets to guide students to construct their knowledge, you need some kind of feedback loop to make sure that the students aren't learning errors as fact, you need spaced practice to prevent the loss of knowledge to disuse, and you need motivation to interest the students in the knowledge. But in what follows I'm focusing on purely the knowledge-transfer component of the education process.

Bligh's book *What's the Use of Lectures?* and Armstrong's Natural Learning in Higher Education go into more detail on what lectures are good for and how to promote better lectures. But I come not to praise lectures but to damn them, to damn them to Hell forever.

This is because reading a textbook and attending a lecture are far from the fastest or most thorough means of learning. In numerous controlled studies students progress about two standard deviations faster with a dedicated one-on-one tutor than with traditional classes. (I think this means about 30% faster.)

You could argue, though, that one-on-one tutoring is too expensive, and can only be justified when the goal of knowledge distribution is light (i.e. it is desired for a small number of students to learn the material) or the benefits of transferring the knowledge are so great that it justifies occupying the time of a large number of skilled tutors who could be practicing the knowledge rather than transferring it. Bloom does in fact argue this: "...more practical and realistic conditions than the one-to-one tutoring, which is too costly for most societies to bear on a large scale."

However, this is not in fact the case. Most lecture classes only lecture to about 24 listeners, the majority of whom (let's say ⅔) do not learn the material well enough to teach it. Of those who do, almost none are allowed to teach the material for some three or four years.

If we track the number of people to whom the knowledge has been transferred over time, it can grow exponentially in this model, but it works out to a doubling time of a year or a bit more.

By contrast, in the "one-room schoolhouse" model, the teacher mostly teaches the most advanced students, who teach the less advanced students, who teach those who are less advanced still. The process of teaching what you have recently learned serves both as a check to unmask the pernicious fluency illusion and as additional practice time. Typically in the schoolhouse, according to rumor, this also achieved a doubling time of around a year.

But we could do much better; the crucial question is how short a time you have between mastering a lesson and having to teach it to someone else. If, for example, you study a lesson one week, balanced half-and-half between working with a tutor and doing exercises on your own, and act as a tutor in that lesson to two other people in the next week, then the doubling time could be under a week. (During the first week you were presumably teaching the previous lesson to two other people during the hours that you weren't studying the new lesson.)

Even if one-on-one tutoring were only as effective as attending lectures, this model would still defeat it; only four weeks after tutoring the two initial students, you'd have $2+4+8+16 = 30$ students already up to speed on it.

Other Benefits of Tutoring

Bloom points out (in his paper on the "2 Sigma Problem" I mentioned above) that tutoring also helps with focus (students spent 90% of time on task instead of 65%) and with positive attitude.

Topics

What's wrong with CoAP

Kragen Javier Sitaker, 2017-06-15 (3 minutes)

I've just read through a bunch of the CoAP RFCs (RFC 7252, say), and although it sounds like a workable protocol (and I know people are using it), it seems pretty suboptimal.

I may be coming at this from a kind of bad position given that I've written an HTTP server that's under 2K of code (albeit on top of Linux's TCP/IP stack), so I'm maybe a bit unsympathetic to arguments claiming that a more constrained protocol is needed for more constrained devices.

You're going to use X.509 certificates on devices with 10 kilobytes of RAM? You're mandating the secp256r1 curve? You're mandating AES? That's feasible, but AES is bigger than my entire web server, and let's hope none of your devices are unconstrained enough to have cache timing attacks on the S-boxes.

You have ETags, great — but If-None-Match can't accept one, so there's no way to revalidate your ETag-indexed cache with a conditional GET? No, wait, you just send an ETag option in your request, and it works like If-None-Match with an ETag does in HTTP.

The minimal CoAP packet size is pretty great, though.

Every path segment of up to 13 bytes needs only a single option-header byte with a 0 option-delta nybble, except for the first one, which will have an option-delta nybble of 11. That's pretty reasonable. And they're guaranteed to occur together in sequence. So actually you can get things pretty compact; here's a 16-byte request:

```
>>> m = aiocoap.Message(code=aiocoap.GET, mtype=aiocoap.CON, mid=1234)
>>> m.opt.uri_path = ('temperature',)
>>> m.encode()
b'@\x01\x04\xd2\xbbtemperature'
>>> len(_)
16
```

`\xbb` there is the pair of "11" nybbles, one of which is the length of "temperature". And the rest of the packet is just the four-byte header, whose first byte encodes version 1, type 0 (CONfirmable), and 0 bytes of token length TKL, and whose second byte is GET (0.01); the request for CONfirmation, and the message id (0x04d2). This packet (with a different message ID) is the first example in Appendix A.

Here's an encoding of `GET /~kragen/sw/dev3/zmqhello.c`:

```
>>> m.opt.uri_path = ('~kragen', 'sw', 'dev3', 'zmqhello.c')
>>> m.encode()
b'@\x01}4\xb7~kragen\x02sw\x04dev3\nzmqhello.c'
>>> len(_)
31
```

I mean, that's not too bad, is it? 31 bytes.

There are no byte-ranges. So you can't use it for partial retrieval. But that's probably just as well.

All the multicast stuff seems kind of poorly thought out, as well as all the security stuff. The "resource discovery" stuff makes it sound like they didn't really understand REST, but its application/link-format content-type is the only format that has a content-format number assigned that can reasonably contain links (assuming we discount XML XLink).

Allowing content-type sniffing even in the presence of an explicit content-format is a terrible idea.

I suspect that the duplicate-transaction problems that RFC 1644 T/TCP had due to lacking the three-way handshake may also afflict CoAP, although of course REST cuts down tremendously on such problems.

Topics

- Performance (p. 3621) (149 notes)
- Facepalm (p. 3450) (24 notes)
- Protocols (p. 3668) (21 notes)
- Security (p. 3701) (9 notes)
- Cryptography (p. 3397) (9 notes)
- CoAP (p. 3380) (4 notes)

HTML is terser and more robust than S-expressions

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

HTML is more succinct for things in its intended domain than S-expressions, but still has better error-detection and correction capabilities.

S-expression fans like to say that HTML, SGML, and XML are just bastardized S-expression languages. SGML partisans often respond that matching end-tags allow for better error-reporting and correction. But for typical HTML content --- mostly running text with a little bit of interspersed markup --- S-expressions are not only harder to correct, but also more verbose.

Consider this partial paragraph from the Ur-Scheme web page http://pobox.com/~kragen/sw/urscheme:

```
<li><b>Reasonably fast.</b> It <b>generates reasonably fast
code</b> &mdash; when compiled with itself, it runs 2½ times
faster (in user CPU time) than when it's compiled with <a
href="http://www.call-with-current-continuation.org/"
>Chicken</a>, 1½ times faster than when it's compiled with...</li>
```

Now, in traditional HTML, I could have left out the quotes around the URL and the ending tag. Consider this S-expression version:

```
(li (b "Reasonably fast.") " It " (b "generates reasonably fast
code") " " mdash " when compiled with itself, it runs 2½ times
faster (in user CPU time) than when it's compiled with "
(a :href "http://www.call-with-current-continuation.org/"
"Chicken") ", 1½ times faster than when it's compiled with...")
```

Most of the markup constructs take up more characters here:

```
LI: '<li></li>'    (end tag could be omitted in traditional HTML)
    '(li "")'
B:  '<b></b>'
    '(b "") '
B:  '<b></b>'      (the second one)
    '" (b "") "'
--- '&mdash;'
    '" mdash "'
A:  '<a href=""></a>'  (quotes could traditionally be omitted)
    '" (a :href "" "") "'
```

If you look at this in a fixed-width font, you'll see that the number of markup characters is detectably smaller in the S-expression serialization of the structure, with the exception of the first two. I maintain that this is typical of the bulk of HTML, especially if you weight it by how often people write it instead of how often it gets sent to browsers. You can come up with examples where that is not the case:

```
<html><head> <title>...</title>
             <link rel="stylesheet" href="../../style.css" />
             <meta http-equiv="Content-Type" content="..." />
             <style type="text/css">...</style></head>...</html>
```

vs.

```
(html (head (title "...") (link :rel "stylesheet" :href "../../style.css")
            (meta :http-equiv "Content-Type" :content "...")
            (style :type "text/css" "...")))
```

but those structure-heavy, text-light examples with long-winded tag names are relatively rare for people to read and write.

Of course, the cost of terser syntax is often that errors are hard to diagnose. Ada's `end loop`, `end if`, `end record`, and so on mean that if you leave out an `end` delimiter, the compiler will usually be able to tell you which one you left out. At the opposite end of the spectrum, S-expression languages in which all the various kinds of `end` are spelled as `)` can only tell you when they get to the end of the program or to something that doesn't make sense in the current context.

This is not a phenomenon limited to end-delimiters. In programming languages, there are many other examples of verbosity that helps to diagnose errors; for example, explicit type declarations, mandatory delimiter characters (in cases where the syntax would be no more ambiguous if they were removed from the grammar), sequences of single-line comments, and the conventional parenthesization of the arguments of fixed-arity functions ("ratio square sin x square sin y" is perfectly unambiguous, after all, and Forth, PostScript, Logo, and REBOL use more or less that syntax.).

However, in the case of HTML, the terser syntax does not make errors harder to diagnose; in fact, the HTML syntax permits better error-detection and even error-correction, because all of the end-tags are explicitly labeled. (It differs from SGML in this regard; in SGML, you can write `<li><b/Reasonably fast./ It ...</>` and eliminate the redundant end-tags altogether.)

Topics

- Syntax (p. 3738) (28 notes)
- Serialization (p. 3707) (6 notes)
- HTML (p. 3508) (6 notes)

Entry-C: a Simula-like backwards-compatible object-oriented C

Kragen Javier Sitaker, 2015-04-05 (updated 2017-04-03) (24 minutes)

Entry-C is a Golang- and SIMULA-67-inspired design for adding object-orientation to C in a backwards-compatible fashion, unlike C++, while adding only a small amount of extra syntax. As with C++, it doesn't require garbage collection, it can be used to write allocation-free programs to ensure constant space usage, and only the language facilities you use will cost you anything at run-time. However, it is more flexible and potentially more efficient than C++.

Semantic overloading of existing language features can make it harder to report compiler errors usefully, because it increases the space of valid programs.

The mysterious `entry` keyword explained

The C language has a set of reserved words that cannot be used for identifiers (such as names for variables, fields, or types), such as `struct` and `if`, which are generally used as syntactic elements of the language. Among these, we find the curiosity `entry`, which is not used for anything; it's simply illegal in valid C programs, like a stray `@` or `` ` ``.

Presumably this was intended for defining functions with multiple entry points, which is an easy enough thing to do in assembly. For example, maybe you'd like to make an argument optional:

```
printf:  push $stdin
fprintf: ...
```

In general, multiple entry points give you a way to make a tail call to another function at a cost of zero instructions in code space and zero execution time. Here's an example tail call from http://canonical.org/~kragen/sw/netbook-misc-devel/nand.c:

```
int inputs_for_pattern(char *pattern) {
  char *s;

  for (s = pattern; *s; s++)
    if ((*s != '0') && (*s != '1') && (*s != 'x')) return 0;
  return log2_int(s - pattern);
}

int log2_int(int n) {
  int rv = 0;
  while (n > 1) {
    if (n & 1) return 0;        /* lousy error reporting, but whatever */
    rv++;
    n >>= 1;
  }
```

```
  return rv;
}
```

Presumably what the `entry` keyword was contemplated for was being able to write the above code more like this:

```
int inputs_for_pattern(pattern)
char *pattern;
{
  char *s;
  int n;

  for (s = pattern; *s; s++)
    if ((*s != '0') && (*s != '1') && (*s != 'x')) return 0;
  n = s - pattern;

  entry int log2_int(n);

  int rv = 0;
  while (n > 1) {
    if (n & 1) return 0;        /* lousy error reporting, but whatever */
    rv++;
    n >>= 1;
  }
  return rv;
}
```

If you remove the `entry` statement in the middle, this is valid C already; but presumably the idea of the `entry` statement is that you could, in a sense, enter in the middle of the function; the tail end of the function is a function in its own right.

This is a relatively minor micro-optimization in most contexts, and error-prone; and making the compiler smart enough that it's actually a win is kind of hard.

Activation-record objects: unifying structs with functions

SIMULA 67's big insight was that, in a way, a function call has a lot in common with an object. It has a set of variables and some code to run with access to those variables. The difference is that a function call has very strict sequencing constraints: first the space for the variables is allocated (and some of the variables are initialized), then the code is executed, and when it returns, the space is deallocated. A C++ object is, in some sense, the union of the features of a struct and a function call.

Nesting constructor code

Suppose we simply relax this constraint and blur the distinction between structs and functions. Then, instead of writing this code (adapted from http://canonical.org/~kragen/sw/inexorable-misc/simpleproxy.c):

```
struct pipe {
    struct pipe *next;
```

```
    struct pipe *partner;
    struct bufferevent *buf;
    int marked_for_deletion;
    int fd;
    char *error;
};

static struct pipe *
new_pipe(int fd)
{
    struct pipe *p = malloc(sizeof(*p));
    if (!p) {
        log_msg("malloc failed (%s)", strerror(errno));
        return NULL;
    }

    p->next = NULL;
    p->marked_for_deletion = 0;
    p->fd = fd;
    p->error = NULL;

    p->buf = bufferevent_new(fd, read_callback, NULL, error_callback, p);
    if (!p->buf) {
       p->error = "bufferevent_new failed";
       return NULL;
    }

    if (-1 == bufferevent_enable(p->buf, EV_READ | EV_WRITE)) {
        p->err = "bufferevent_enable failed";
        bufferevent_free(p->buf);
        return NULL;
    }

    return p;
}

static void
error_callback(struct bufferevent *bufev, short what, void *arg)
{
    struct pipe *p = arg;
    log_msg("error %d on conn %d", what, p->fd);
    mark_for_deletion(p);
}
```

we could write

```
struct pipe(int fd)
{
    struct pipe *next = NULL;
    struct pipe *partner;
    char *error = NULL;
    int marked_for_deletion = 0;
    struct bufferevent *buf = bufferevent_new(fd, read_callback, NULL,
                                              error_callback, &fd);
    if (!buf) {
```

```
        error = "bufferevent_new failed";
    } else if (-1 == bufferevent_enable(buf, EV_READ | EV_WRITE)) {
        error = "bufferevent_enable failed";
        bufferevent_free(buf);
        buf = NULL;
    }
};

static void
error_callback(struct bufferevent *bufev, short what, void *arg)
{
    struct pipe *p = arg;
    log_msg("error %d on conn %d", what, p->fd);
    mark_for_deletion(p);
}

static void
mark_for_deletion(struct pipe *p)
{
    /* ... */
}
```

This puts the initialization code for the struct inside the struct itself, leaving the allocation or deallocation up to the caller. It's clearly a win in terms of avoiding code duplication, but it has a few drawbacks.

It doesn't have a convenient way to report initialization errors, since it doesn't have a return value, and it also doesn't have convenient syntax like C++'s `this` to take the address of the object it's initializing, for use in registering the callback `error_callback` with libevent. Instead, it type-puns a pointer to the first parameter, which is implicitly stored in the struct, into a pointer to the struct as a whole — which will break with no compiler warnings if someone inserts a new parameter before `fd`.

Pointers to nested functions with trampolines

GCC implements Pascal-like nested functions, with access to the outer function's stack frame, using a small amount of runtime code generation, putting a small trampoline that pushes a pointer to the stack frame and then jumps to the function, into the stack frame itself. If we use that feature, we could simplify the above to the following:

```
struct pipe(int fd) {
    struct pipe *next = NULL;
    struct pipe *partner;
    char *error = NULL;
    int marked_for_deletion = 0;

    void error_callback(struct bufferevent *bufev, short what, void *arg)
    {
        log_msg("error %d on conn %d", what, fd);
        mark_for_deletion();
    }

    void mark_for_deletion()
```

```
    {
        /* ... */
    }

    struct bufferevent *buf = bufferevent_new(fd, read_callback, NULL,
                                              error_callback, NULL);
    if (!buf) {
        error = "bufferevent_new failed";
    } else if (-1 == bufferevent_enable(buf, EV_READ | EV_WRITE)) {
        error = "bufferevent_enable failed";
        bufferevent_free(buf);
        buf = NULL;
    }
};
```

Now, `error_callback` no longer needs an explicit pointer passed in, and indeed if libevent were written to require you to use runtime code generation to use basic functionality, `bufferevent_new` could take three parameters instead of five. This expands `struct pipe`'s space by the space needed to hold the trampolines.

GCC's regular nested function support generates functions that are only safely callable until the outer function returns, because both the outer function variables and the trampoline itself are stored on the stack. This design avoids that restriction, because the trampoline can be stored within the struct itself.

Invoking nested functions without trampolines

The only reason for the trampolines, though, is that we're converting nested-function pointers into regular function pointers, and then, instead of calling them immediately, we are passing them to a function that takes regular function pointers as parameters. The call to `mark_for_deletion` in the above code needs no such trampoline; neither does a call such as the following:

```
void invoke_error(struct pipe *p) {
    p->error_callback(NULL, 37, NULL);
}
```

The underlying `error_callback` function takes an extra hidden parameter that is a `struct pipe *`, and in this case we're passing that parameter explicitly.

If we adopt a rule that only code lexically within the struct declaration is permitted to convert nested-function pointers to regular C function pointers, then we can statically compute the set of nested functions that need trampolines generated for them, which will usually be empty.

Initializing objects

Normally you can initialize a statically-allocated or stack-allocated struct with an initializer list (from http://canonical.org/~kragen/sw/netbook-misc-devel/hanoitri.c:

```
typedef struct { int num, denom; } ratio;
static const ratio sin_60 = { 56755, 65535 };
```

It makes sense that you'd repurpose that same syntax to supply initializer arguments to struct types like `pipe` above that have declared constructor arguments. Also, as in C++, it seems convenient to allow you to just say `= x;` rather than `= { x };` in the case where there's just one argument.

But what about dynamically-allocated objects on the heap, or in an array or whatever?

I think the most harmonious approach is to define an additional coercion rule from lvalues that are structs with constructors, or pointers to structs with constructors, to function pointers. If you invoke such an lvalue as if it were a function, then it decays to a pointer to the constructor code, which has a void return value.

Thus, this will initialize element 5 of the array of pipes with the `fd` argument `fd0`, and element 6 with `fd1`, and give a compile error about a missing constructor argument on element 7:

```
struct pipe pipes[10];
pipes[5](fd0);
struct pipe *p = &pipes[6];
p(fd1);
pipes[7]();
```

The following will also give a compile error, because you're trying to invoke a constructor on a `struct pipe` rvalue, which is generally a useless thing to do:

```
struct pipe pipes[10];
struct pipe f() { return pipes[5]; }
f()(fd0);
```

This seems compatible with considering structs and functions merely slightly different versions of the same thing.

Temporary objects

In C++, if you have a `Rational` class, you can say

```
Rational seven_sixths = Rational(2, 3) + Rational(1, 2);
```

which (given the existence of the relevant constructors and overloads) constructs two or three temporary `Rational` objects and then destroys them (you might say "on the stack", although of course that's just one possible implementation). This is pretty handy, but the rules about the lifetimes of those objects are very tricky, and the syntax creates a conflict between the namespace of classes and the namespace of functions and variables. The interaction of this feature in C++ with the absence of sequence points between the constructors created a bug in most C++ programs where one constructor may be partly complete when the other throws an exception.

So, for the moment, I'm not adding temporary objects to Entry-C.

Ad-hoc polymorphism with dynamic dispatch with Golang-like interfaces

I've been claiming Entry-C was a "backwards-compatible object-oriented C", but so far all I've given is a way to avoid writing

a separate initialization function for your structs and a little syntactic sugar to define functions that operates on them more conveniently. Object-orientation is usually described as consisting of encapsulation, (ad hoc) polymorphism, and inheritance, [and something else?] none of which have made an appearance so far. Also, I haven't used the `entry` keyboard I spent so many bytes bloviating about at the beginning.

So let's steal Golang's approach to polymorphism wholesale. Golang uses these things called interfaces to get polymorphism. Golang interfaces at runtime are basically three-tuples: a pointer to an object, a pointer to a vtable, and a pointer to the underlying type of the object for the sake of further interface conversions. Any method can be invoked polymorphically, via an interface, or monomorphically, via some concrete type; and you can attempt to convert anything to any interface type. Conversions from a concrete type to an interface type use a vtable for that interface type computed at compile time, and give a compile-time error if that conversion is impossible. Conversions from one interface type to another use a vtable generated at runtime, and if the new interface type is not simply a subset of the old one, they can yield a runtime error.

We can add the same mechanism to C by using the `entry` keyword for a totally unintended purpose — declaring an interface type, with a syntax exactly analogous to `struct` or `union`:

```
entry callback_handler {
    void error_callback(struct bufferevent *, short, void *);
    void read_callback(struct bufferevent *, void *);
};
```

Now we can cast a `struct pipe` to an `entry callback_handler`, which will use a pointer to a compile-time-generated vtable, as with Golang; and we can also attempt to cast an `entry {}` to an `entry callback_handler`, which may succeed (constructing a new vtable at runtime, or finding a cached one) or may fail.

And that's polymorphism in Entry-C: both more efficient and more flexible than in C++. It's more efficient because you only use polymorphism where it's needed; it's more flexible because the caller, not the callee, decides when polymorphism is needed.

Encapsulation

Encapsulation might be a good idea, but Entry-C doesn't have it, or even a good way to approximate it.

You can make some of your methods "private" by declaring them `static`, which will also keep them from being called polymorphically via an `entry`. But usually the things you would most like to make private are variables, and you can't make variables private by declaring them `static`, because static variables are something else.

The best you can do is only a little better than in standard C: you can put an `entry` type in your .h file and keep the actual implementation type inside your .c file. Then, because the callers can't allocate space for your implementation type, you probably need to provide a function to allocate new instances of it and return `entry`s to them, and a function (perhaps in the `entry`) to deallocate instances.

This is better than the "pimpl" idiom in standard C or C++

because it doesn't have that disgusting name.

Inheritance

Traditional inheritance is a terrible idea:

• Subtyping gives rise to questions of covariance and contravariance, which make your programming language an unnecessarily confusing user interface.
• The possibility of inheritance typically makes every variable run-time polymorphic, which makes reading the code more challenging. As a consequence, things that ought to be compile-time errors (e.g. casting from a type that doesn't define `foo()` to an interface that requires it) become runtime errors (because some subclass might define `foo()`). Also, things that ought to be trivially easy for the compiler to optimize become difficult. You can't inline a callee if you don't know which callee is going to be called.
• Classes that are not designed to be subclassed often can't be subclassed to do what you want, although the language won't stop you from trying.
• Classes that *are* designed to be subclassed have no way in the language to define the interface between themselves and their subclasses, and it's very common for new versions of library classes to break existing subclasses; for example, it's trivially easy to accidentally get recursion this way, creating the danger of not just infinite recursive loops but also problems with re-entrancy.
• Everything that you can do with inheritance, you can do better with delegation.

So Entry-C doesn't have traditional inheritance.

However, as in Plan 9 C and consequently as in Golang, you can import the names defined in a nested struct into your own namespace by simply not naming the struct:

```
struct pipe;
```

The names in the inner anonymous struct (both inner functions and data fields) will be visible wherever your own names would be visible. If you define conflicting names, that's a compile error. This enables you to share implementations of certain methods across multiple classes in a convenient way, without the problems introduced by standard inheritance, but it's purely syntactic sugar for a bunch of one-line delegation methods. The "inherited" methods can't implicitly call methods you define.

At file scope, this allows you to incrementally refactor from global data to local data; you can start by collecting the global data into an anonymous struct, then move functions into it one or a few at a time.

Error reporting

There's a big problem with error reporting in C, and adding constructors to structs just makes it worse, because you might want to do things in a constructor that could fail, and as in C++, the constructor doesn't really have a way to return that failure value. This is, I think, why Golang doesn't have constructors.

The standard C++ answer is to use exceptions, but exceptions have their own problems, especially in the absence of garbage collection

and in the presence of the kind of things that make code non-reentrant.

The Golang answer is to use multiple return values for everything that might fail. Entry-C doesn't have multiple return values, although clearly C should have had them.

The standard C answer is for your initializer function to either take the address of the struct as an argument and return an int, or to return the address of a newly allocated struct or NULL. Entry-C constructors don't have return values.

You can do what I did in the example above: define an "error" field in the object, indicating whether construction failed. Alternatively, you can move things that could fail out of the constructor, and return a success or failure code:

```
struct pipe() {
    struct pipe *next = NULL;
    struct pipe *partner;
    struct bufferevent *buf = NULL;
    int fd;
    int marked_for_deletion = 0;

    void error_callback(struct bufferevent *bufev, short what, void *arg) {
        log_msg("error %d on conn %d", what, fd);
        mark_for_deletion();
    }

    void mark_for_deletion() {
        /* ... */
    }

    char *open(int new_fd) {
        fd = new_fd;
        buf = bufferevent_new(fd, read_callback, NULL,
                              error_callback, NULL);
        if (!buf) {
            return "bufferevent_new failed";
        }

        if (-1 == bufferevent_enable(buf, EV_READ | EV_WRITE)) {
            bufferevent_free(buf);
            buf = NULL;
            return "bufferevent_enable failed";
        }
        return NULL;
    }
};
```

However, Entry-C also adds a new kind of error condition: casting one entry type to another that isn't a superset could produce a runtime error. XXX I need a design for how to handle that operation and the resulting error.

Function overloading

No. *Fuck* no.

Operator overloading

This might be a good idea, especially for arithmetic. I'm not sure how to do it, especially without constructing temporary objects or throwing exceptions.

Example: integer DSP pipelines set up at runtime

Here we have a bunch of DSP filters that can be connected to each other at runtime, including a software phase-locked loop, using no heap allocation and no casts. The final paragraph sets up an example.

This code is adapted from http://canonical.org/~kragen/sw/netbook-misc-devel/pll.c.

A "crad" is a made-up unit of angular measure designed to avoid multiplication and division in the main loop of the PLL algorithm. (You could argue that it's at least as bad to do polymorphic method dispatch, but probably only if you're on a computer so big that memory references are slower than division.)

```
enum {
    pi = 0x400,     /* Crads per pi radians. must be a power of 2 for & */
    iir_shift = 9,  /* >> for low-pass rolloff at 512 samples (16Hz). */
};

/* Every struct defined below can be cast to this type. */
entry sink {
    void consume(int sample);
};

/* om0 is const base angular frequency, omega, in crads/sample. */
struct pll(int om0) {
    int pha;         /* Current phase in crads. */
    int err;         /* Accumulated low-pass-filtered phase error. */
    int amp;         /* Accumulated low-pass-filtered amplitude. */

    /* Compute the actual current angular frequency of a PLL. The bit
       shift "10", which determines how much frequency shift we get, was
       determined by trial and error because I am ignorant of theory. */
    static int current_om() { return om0 + (err >> 10); }

    /* Update PLL state for a new input sample. */
    void consume(int cc) {
        pha += current_om();
        iir_low_pass(&err, (pha       ) & pi ? cc : -cc);
        iir_low_pass(&amp, (pha - pi/2) & pi ? cc : -cc);
    }
};

static void iir_low_pass(int *out, int in) {
    *out += in - (*out >> iir_shift);
}

struct pll_amp_filter(struct pll *p, entry sink s) {
    void consume(int cc) {
        p->consume(cc);
```

```
        s.consume(p.amp);
    }
};

struct pll_freq_filter(struct pll *p, entry sink s) {
    void consume(int cc) {
        p->consume(cc);
        s.consume(p.current_om());
    }
};

struct low_pass_filter(int current_value, entry sink s) {
    void consume(int sample) {
        iir_low_pass(&current_value, sample);
        s.consume(current_value);
    }
};

struct freq_to_squarewave(entry sink s) {
    int ph = 0;
    void consume(int omega) {
        ph += omega;
        s.consume(ph & pi ? -1 : 1);
    }
};

struct hold(int val) {
    void consume(int new_val) {
        val = new_val;
    }
};

struct multiplier(entry sink s, struct hold volume) {
    void consume(int sample) {
        s.consume(volume.val * sample);
    }
};

struct putchar_sink {
    void consume(int sample) {
        putchar(sample);
    }
};

struct example {
    struct pll p = 110 * pi * 2;
    void consume(int sample) {
        p.consume(sample);
    }
    struct putchar_sink s;
    struct hold volume = 255;
    struct multiplier m = { s, volume };
    struct pll_freq_filter f = { &p, m };
```

/// XXX missing square wave generator struct pll_amp_filter a =

```
{ &p, volume };  };
```

Example: some of Lisp

This example suffers from the lack of a way to write `this`, and it probably needs garbage collection, which will really benefit a great deal from being able to write `this`.

```
entry value {
    entry value car();
    entry value cdr();
    int is_null();
    struct symbol *as_symbol();
    void visit(entry collector gc);
};

entry collector {
    void recurse(entry value child);
    void found(void *data);
};

struct pair(entry value car_, entry value cdr_) {
    entry value car() { return car_; }
    entry value cdr() { return cdr_; }
    int is_null() { return 0; }
    struct symbol *as_symbol() { return NULL; }
    void visit(entry collector gc) {
        gc.found(&car_);  /* XXX this means "this" */
        gc.visit(car_);
        gc.visit(cdr_);
    }
};

struct symbol(const char *name, struct symbol *next) {
    entry value car() { return err("car of a symbol"); }
    entry value cdr() { return err("cdr of a symbol"); }
    int is_null() { return 0; }

    struct symbol *as_symbol() {
        return (symbol*)&name;
    }
    void visit(entry collector gc) { gc.found(&name); }
} *symbol_table[256] = {0};

void *allocate(size_t size) {
    void *rv = malloc(size);
    if (!rv) abort();
    return rv;
}

struct symbol *intern(const char *name) {
    /* This hash function is almost as silly as PHP using strlen */
    struct symbol *b = &symbol_table[(int)name[0] & 255];
    struct symbol *s = b;
    while (s) {
        if (0 == strcmp(name, s->name)) return s;
```

```
        s = s->next;
    }

    s = allocate(sizeof(*s));
    s(name, b);
    *b = s;
    return s;
}

struct _nil_type {
    entry value car() { return err("car of nil"); }
    entry value cdr() { return err("cdr of nil"); }
    int is_null() { return 1; }
    struct symbol *as_symbol() { return NULL; }
    void visit(entry collector gc) { }
} nil;

struct pair *cons(entry value car, entry value cdr) {
    struct pair *rv = allocate(sizeof(*rv));
    rv(car, cdr);
    return rv;
}

int length(entry value v) {
    return v.is_null() ? 0 : 1 + length(v.cdr());
}

entry value assq(entry value alist, symbol *key) {
    if (alist.is_null()) return nil;
    entry value item = alist.car();
    if (item.car().as_symbol() == key) return item;
    return assq(alist.cdr(), key);
}

entry value reverse(entry value list) {
    static entry value reverse_with(entry value list, entry value tail) {
        if (list.is_null()) return tail;
        return reverse_with(list.cdr(), cons(list.car(), tail));
    }
    return reverse_with(list, nil);
}
```

Nothing surprising so far, except maybe that I didn't make a `typedef` for `entry value`. Now let's amp up the weird with a lazy mapcar:

```
struct mapcar *mapcar(entry value *f(entry value), entry value items) {
    struct mapcar *rv = allocate(sizeof(*rv));
    rv(f, items);
    return rv;
}

struct mapcar(entry value *f(entry value), entry value items) {
    entry value car() { return f(items.car()); }
    entry value cdr() { return mapcar(f, items.cdr()); }
    int is_null() { return items.is_null(); }
```

```
    struct symbol *as_symbol() { return items.as_symbol(); }
    void visit(entry collector gc) { gc.found(&f); gc.visit(items); }
};
```

Heap-allocation here is necessary because the caller of `cdr()` doesn't know they're receiving a `struct mapcar`; the declared return type of `cdr()` is just `entry value`. Consequently, they can't allocate space for a `struct mapcar` return value; so, that needs to be allocated on the heap. That's one reason this needs garbage collection, but of course Lisp in general kind of needs it.

This is the kind of situation where "virtual destructors", which is to say a run-time polymorphic method to recursively deallocate child resources, are helpful, although in this particular case they aren't enough, because some objects in the object graph may be referenced more than once.

Topics

- Programming (p. 3658) (286 notes)
- History (p. 3500) (71 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Programming languages (p. 3656) (47 notes)
- Syntax (p. 3738) (28 notes)
- C (p. 3359) (28 notes)
- Object-oriented programming (p. 3606) (10 notes)

Oscilloscope screens

Kragen Javier Sitaker, 2018-06-05 (2 minutes)

I was thinking it would be pretty nifty to rig up something like the displays in Brazil using a big Fresnel lens and an old analog oscilloscope tube. The benchmark here is maybe the TX-2 display used to develop Sketchpad and GENESYS in the 1960s: 1024×1024 with a single brightness, a limited range of color (blue initially, yellow afterglow), typically less than 32768 points lit, and I think 100 000 points per second (?).

If we were to drive a 20MHz oscilloscope with the analog VGA signal from a modern video card, we could probably actually do better than that. The video card can produce about 72 *million* points per second, about 720 times greater than the TX-2, and has memory for a bit over a million of them, about 32 times greater. The tube becomes the limiting factor — if we figure that it can be persuaded to go to an arbitrary spot on the display 40 million times a second, perhaps with a bit of filtering to compensate for the 3dB attenuation and phase shift at that point, that's one limitation. But that still means we could paint every point on the display 40 times a second — the slow phosphor becomes an even bigger limitation. And it probably doesn't have 1024×1024 faceplate resolution, maybe more like 512×512 if we're lucky.

A possibly more entertaining approach is to allocate, say, 100k points to each of many oscilloscopes, and refresh them each at, say, 10 Hz — with a fast enough switch you could multiplex one pixel to each display and thus drive 72 oscilloscopes off a single video card, but it's probably easier to switch a 20MHz signal to each display for the requisite 2.5 milliseconds out of every 100, limiting you to 40 oscilloscopes but allowing you to use a 4kHz multiplexing switch.

If you wanted to drive them all from a single frame — so you didn't have to synchronize your page flips with the switch — you could still drive about 10 oscilloscopes.

Topics

• Retrocomputing (p. 3685) (13 notes)
• Displays (p. 3414) (13 notes)
• Oscilloscopes (p. 3614) (12 notes)

India rubber memory

Kragen Javier Sitaker, 2019-03-19 (4 minutes)

I watched Tony Hoare's 2009 talk about how null pointers were a billion-dollar mistake today. One of the things he mentioned was that, often, pointers are used because computer memory doesn't stretch — you want to logically include some child object into a parent object, but the child object has a potentially variable size, so you're stuck using a pointer to it (a Box, in Rust parlance). If you had "India-rubber memory" that could expand without invalidating later pointers, you could just include the child object inside the parent, as we traditionally represent syntax trees in text using nested brackets.

This is an appealing idea, and I wonder if you actually could make it work in practice, using some kind of flexible buffer scheme to hold all your data — like Emacs buffers with markers pointing into them, except that the buffers can contain not only characters but also references to markers. To the extent that you can make your data immutable, you can avoid the need to update the positions of markers from inserting and deleting data in the middle of the buffer. You need the markers in order to be able to rapidly jump over variable-length child objects and reach later children.

For example, suppose you have these definitions (from the FlatBuffers tutorial):

```
struct Vec3 {
  x:float;
  y:float;
  z:float;
}

table Monster {
  pos:Vec3; // Struct.
  mana:short = 150;
  hp:short = 100;
  name:string;
  friendly:bool = false;
  inventory:[ubyte];  // Vector of scalars.
}
```

Now, if you know the starting byte position of a `Monster`, you can easily find the starting byte position of its `pos`, `mana`, `hp`, or `name`, since those are all at fixed offsets. But `friendly` comes after `name`, and `name` is variable-length, so somewhere we must store a marker to find `friendly` with — a marker which will continue to point to `friendly` even if some characters are inserted or deleted in `name`. From that marker we can find the byte position of `inventory`, since `friendly` itself is fixed-size. We don't need a second marker to find the end of the object, but since the variable-length `name` and `inventory` fields make `Monster` as a whole variable-length, we would need an array of markers to organize an array of `Monsters`, and if `Monster` were embedded in some other object, we'd need a marker to find the field following the `Monster`.

At least, that's one possible set of implementation decisions. You

could also imagine storing the markers at the end of the `Monster`, and using the end position rather than the start position to identify the `Monster`, which would have the advantage that you could write out the representation in a purely sequential fashion with no backpatching, which in turn would allow you to use variable-length representations for the markers, such as variable-length byte counts. In that case, you'd need a marker to find the beginning of `inventory` (and the end of `friendly`) and a marker to find the beginning of `name` (the end of `hp`). In this case, this "trailers" representation requires more markers than a "headers" representation would, but that's only because the last field is variable-length; more usually both would require the same number of markers.

You could also imagine representing the markers in such a way that an update to them was needed under different circumstances. For example, you could represent a marker as a sequence of indices into B-tree blocks walking down from some root, and only an insertion into a B-tree block before the marker would require an update to that marker; or you could represent it as something like a BASIC line number, in the sense that the marker destinations would be physically present in the buffer along with characters, in a lexicographically ordered fashion, and might occasionally need a "renum" operation to redistribute them over the index space, which could be carried out somewhat lazily.

Topics

• Programming (p. 3658) (286 notes)
• Memory models (p. 3572) (13 notes)
• FlatBuffers

Hot lye granite cutting

Kragen Javier Sitaker, 2019-11-01 (2 minutes)

Granite and similar materials are commonly cut to shape with diamond saws. This has a few disadvantages: even abrasive-grade diamonds are still somewhat expensive, the process produces carcinogenic dust of crystalline silica, and, with the usual circular-saw blades, the cuts are necessarily fairly straight and not very precise.

Granite, as I understand it, gets most of its strength from the interlocking quartz crystals that make up a significant part of its bulk. Other major constituents such as feldspar and mica are mostly much softer and less chemically inert, so they are easier to cut. (Zircon is an exception, but it typically constitutes only a small part of granite.)

Quartz can be converted to water-soluble sodium silicate waterglass with an aqueous solution of sodium hydroxide heated a bit above room temperature, ideally to 100 degrees. So a plausible approach to these problems is to cut preheated granite with a wire saw flood-lubricated with hot lye. For the wire, you'd need a metal that was adequately stable in the strongly reducing, warm, wet, and alkaline conditions we're talking about; I think ordinary steel wire would work fine, but if not, surely some kind of brass or bronze would be adequate, in an argon chamber if all else fails. Using a wire made of a hardened tool steel would increase the risk of wire cracking but reduce the abrasive wear on the wire. Some saw-tooth-like surface treatment (perhaps a helical ridge) might be effective at increasing swarf clearance and concentrating abrasive wear on a non-structural part of the wire.

The side forces on the "sawing" wire would be much smaller than the forces on a diamond saw or even an abrasive-sawing wire, and it could cut paths through the stone with tight curves and corners, enabling the fabrication of a much wider array of shapes than a circular saw can, perhaps at higher precision, and certainly with much smaller side forces on the machine guiding it.

This approach could also work for cutting other troublesome materials that are vulnerable to lye, in particular including glasses such as soda-lime glass and borosilicate, but also some other silicate minerals and metal sulfides.

Topics

- Materials (p. 3560) (112 notes)
- Manufacturing (p. 3558) (50 notes)
- Digital fabrication (p. 3411) (42 notes)

Bicicleta maps

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

It should be relatively straightforward to implement Bicicleta with Self-style "maps" --- where each "map" contains a bunch of constant things inherited from somewhere (in Self, cloned from somewhere) and a few variable things.

Many objects in Bicicleta seem to inherit directly from some widely-used global constant (prog.if, prog.sys.nil, and so on) and then override a small number of things. Objects that inherit directly from a particular other object and override the same set of things can share the same "map", just as if they were members of a Self clone family.

If you take outer-self references into account as if they were overridden slots, you can cover a larger number of cases. For example, prog.sys.rational.coerce:

```
coerce = {op: arg1 = 2
    '()' = prog.if(
        op.arg1.denom.is_ok -> op.arg1,
        op.arg1.as_integer.is_ok -> self.new(op.arg1.as_integer, 1),
        else = prog.error(\"could not coerce {arg1} to rational\" % op)
    )
}
```

The 'coerce' methods in different 'rational' objects differ only in their binding of 'self', and their children differ only in that and in their binding of 'arg1'. Ideally you could put most of this stuff into a prog.sys.rational.coerce "map" object.

This brings up a bit of a problem --- ideally you'd have one map object for all the (ordinary) coerce activation records, and those activation records would just have slots for self and arg1. As proposed above, each rational object's coerce method would have only a slot for self; but the activation records of a particular rational object's coerce method would have a map of their own, corresponding to that rational object.

Topics

Gradient rendering

Kragen Javier Sitaker, 2016-09-24 (11 minutes)

From https://news.ycombinator.com/item?id=12223616

Raph wrote:

Another serious potential win in SVG is that you might be able to interleave the area integration with alpha compositing / masking. Could end up pretty darned fast.

I responded:

I was thinking about using your prefix-sum approach for exactly that almost all of last night. I don't have an attack on the problem that I'm satisfied with yet:

• Render each path to a fresh pixel buffer, alpha-compositing it down onto the final canvas. Advantage: straightforward, works for sure. Disadvantage: you need a lot of multiplies per pixel.
• Partition paths into "layers" of nonoverlapping paths, render each layer as a unit, and composite each new layer down onto the final canvas. Overlapping opaque paths can be incorporated into the same layer as whatever is below them by cutting an overlap-shaped hole in what's below before adding in the new path; although that involves some intersection tests to know where to stop, my intuition is that it will be a big win. Advantages: straightforward, probably faster than the previous one. Disadvantages: the partitioning is a potentially costly extra step (one of those things that makes me wonder if it's NP-complete to do it optimally), and there are still potentially many multiplies per pixel.
• Separately accumulate a numerator (total premultiplied color) and denominator (total alpha) for each pixel, then divide in the end. Advantages: You avoid doing lots of work per pixel. Disadvantages: This is a weighted sum, not alpha blending. Alpha blending is a different thing. So the result is wrong. Also, an honest division per pixel is more expensive than quite a number of multiplications, although maybe you could cheat on the final division with a table of approximate multiplicative inverses or something. So this would probably be super slow.
• Find a different group other than $\mathbb{Z}/256\mathbb{Z}$ in which to do prefix-sum that somehow gives you the right results. Then you can just render all the edges into the same buffer and do a single vectorizable prefix-sum operation over it.

Advantages: This sounds super fast.

Disadvantages: It seems clear that this group is going to have to be able to represent the entire Z-ordered stack of colors at every pixel, because if I'm looking at some translucent green on top of translucent red on top of opaque black on top of pale blue, and I reach the right (negative) edge of the opaque black path, somehow I have to have remembered the blue thing underneath in order for it to peek through, which suggests to me that I need an unbounded number of bits per pixel to implement this scheme, which probably is not going to admit an actually fast implementation. In effect it has to reduce to the second approach, except that the software has to deal with the stack of layers once for every pixel. Or is there some magical way around this, at least for a fast-path case?

This part is probably obvious to you, Raph, but you can do SVG linear gradients with two prefix-sum passes instead of one, where the first pass just runs over signed gradient stops and gradient clipping boundaries, and then you draw the signed path boundaries into the buffer before the second prefix-sum pass. (Is that clear? I suspect it may be too abbreviated.)

I suspect that with three prefix-sum passes you could do a decent quadratic-spline† approximation of arbitrary gradients, including the weird skew cone gradients SVG calls "radial gradients". But I haven't worked out the details.

I know you don't have a lot of time to hack on this stuff right now, but would you have time to provide feedback if I were to hack on it a bit? I imagine that I'd run into any number of places where talking to you about it for half an hour could save me days of wasted effort.

† here I'm talking about what Carl de Boor calls "splines", which I know disagrees with your usage of "splines". I think you called them "B-splines" in your dissertation.

Then I wrote this:

You might have seen Raph's recent article on font-rs, where he observes that the color of a pixel in an antialiased painted polygon is the prefix sum of the signed edge coverages, and prefix sum is a thing you can do super fast with SSE.

Dan Amelang's work with Gezira and Nile is pretty inspiring to me, and the idea that you could get enough performance out of modern hardware to not have to worry about precaching font glyphs or really anything else is pretty neat. And hey, AAA games rerender each frame more or less from scratch, right? And in 3D at that.

I've had pretty good experiences hacking together stuff in SVG, especially with the little bit of data binding provided by D3 — you end up with a buttload of code, but you have a lot of control over how things look, and you seem to have pretty good composability, even if there's a large constant factor. It's just kind of like writing stuff in assembly, but less error-prone. (So maybe the verbosity problem of D3 is a shallow problem that could be solved with a better surface notation, like BLISS or Thompson's B instead of assembly?)

Amelang's Gezira, as I understand it, uses a single graphical primitive: a path made of quadratic Bézier curves, filled with a single solid alpha-blended color. SVG primitively supports filling with multi-stop linear RGBA gradients as well as solid RGBA colors, and if you were going to pick a single kind of fill, gradients include solid colors as a special case. And linear gradients are powerful enough that you can Gouraud-shade projected triangles with them, although I never took advantage of that when I did that long-ago 3-D torus.

It may have occurred to you by now that linear gradients can also be rendered by the prefix-sum method; it just requires two passes instead of one. First you fill the polygon with the ∂color/∂x of the fill (which is constant for a linear gradient, regardless of its orientation, until you hit a stop) using a first pass of the prefix sum; then, to that, you add the signed edge coverages at the borders of the polygon before doing a second pass of prefix sum.

I haven't yet figured out how to do overlapping alpha-blended filled paths with this method. Nonoverlapping paths can be done simply by rendering all their signed edges into a single buffer before the prefix-sum step or steps, and overlapping opaque paths can be done by cutting holes in the paths on the bottom, removing the hidden parts of their original edges (which nearly requires the property Levien calls *extensionality* in his dissertation) and adding extra edges to route them around the intruding space. But overlapping alpha-blended paths seem like they could be arbitrarily expensive.

So, I was thinking that it might be reasonable to make a UI out of nothing but a collection of closed Bézier polylines filled with linear RGBA gradients. Maybe? Texture maps would be hard to fit into that model, and textures are pretty important, I don't know.

Dave Long pointed out that two overlapped alpha-blended linear gradient regions still add up to a linear gradient:

> Is that really true? After reading, I'd guess you'd calculate stops at every path vertex *and* path intersection[0]. At each stop, the color layers are known and because gradients are linear, the combination of gradients from all layers ought to also be linear.
>
> [0] these could be expensive, up to n**2

More recently, it occurs to me that there is a potential problem with precision here, but it can be overcome. If we want to be able to make a linear gradient from any given 24-bit color to any other given 24-bit color across some horizontal distance, then we need to support a change of a single count across that distance. If our intermediate results are only 48 bits (16 bits per color plane), then the furthest we can reach without missing is 256 pixels horizontally. (In a sense, the other 8 bits per pixel hold a fractional color or error accumulator.)

This suggests that the optimal way to do this thing is to render things in 256×1 chunks, with the buffer occupying 1536 bytes. Alternatively, you could do it a scan line at a time with 32-bit ints, which would work properly as long as the scan lines were shorter than 2^{24} pixels, but this requires more buffer space and also runs half as fast with optimal SIMD instructions. For example, a 2048-pixel scan line would require 12288 bytes of buffer space, big enough to put serious

pressure on the L1D cache. (I'm writing this on an Atom with a 24 KB 6-way set associative write-back L1D cache, for example.)

It might turn out to be work more effectively with current SIMD instruction sets to render a band of several scan lines at a time, because the vertical SIMD instructions have better coverage of the available space. For example, using AVX-512 instructions, you could maybe handle 32 scan lines in each instruction, or maybe one color plane for each of 10 scan lines, thus rendering a 256×10 or 256×32 band. I think that with AVX-512 and four-way loop unrolling, the inner loop of the critical prefix sum there will take probably 320 cycles; if it's 256×10, that would be 8 pixels per cycle, or 4 pixels per cycle if you do it twice. Repainting a 2048×1536 screen that way will take about 786432 cycles, plus whatever overhead is needed to set up the gradient stops and copy the finished pixels into the framebuffer and whatnot.

(AVX2 and AVX-512 support gather load with the VGATHERDPS, VGATHERDPD, etc., instructions, as well as permuting stuff within a register with VPERMPS, VPERMD, PSHUFB, etc., so copying the finished pixels into the framebuffer might not be the horrible performance nightmare you might expect. AVX-512F also has scatter store.)

Topics

Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity

Kragen Javier Sitaker, 2016-09-17 (13 minutes)

I've been thinking about a new computational model to eliminate the runtime costs of dynamic dispatch and take better advantage of modern heterogeneous hardware.

Background

The Tera MTA

The Tera MTA was a computer that achieved consistently high performance despite a high-latency main memory and no cache hierarchy by hiding the latency with multithreading — its heavily-multibanked RAM system could hold a lot of requests in flight, each one being routed to an autonomous memory unit, and the CPU's hardware context switching with multiple hardware register sets meant that whenever some data arrived from a memory unit, it would probably wake up a thread that could do work on it. In this way, the MTA was able to achieve very high performance not only on cache-friendly tasks like dense linear algebra but also on cache-unfriendly tasks like traversing linked lists.

The Actor model

The Actor model unifies objects, threads, and activation records into a single "actor" abstraction, which reacts to an incoming message by sending some set of other messages and changing its internal state. Erlang is probably the language most directly based on this model, but its evolution was intimately intertwined with that of Smalltalk, in which an object that "receives" a "message" will "answer" it.

You could imagine an implementation of the Actor model in which each actor is a physically separate piece of hardware, communicating with other actors through some kind of bus or packet-switching network that carries the messages; but it's much more typical for a single CPU to sockpuppet the actors one by one, first executing the code of one actor, then the code of another. Although the Actor model easily accommodates an actor sending several messages without waiting for responses, or receiving a message and not responding to it, the traditional approach is for the CPU to implement nearly all message sends by subroutine calls or returns.

Dynamic dispatch is slow

One of the key difficulties in efficiently implementing Smalltalk and similar object-oriented languages has been the overhead of dynamic dispatch. In theory, in Smalltalk, not just every function call, but every arithmetic operation and every conditional, is performed by sending a message to an object. Conditionals are performed by sending messages like #ifTrue:ifFalse: to a boolean

object; arithmetic is performed by sending messages like #+ to a number object. The lack of static type information in Smalltalk and descendant languages like JS, Python, Objective-C, and Ruby means that in general the dispatch of these messages requires some pointer indirections and a hash-table search with the method selector. This has led to performance in optimized versions of these languages typically about an order of magnitude worse than optimized C; Objective-C, like C++, avoids this by reserving such dynamic method calls for special occasions. (C++ additionally uses extra static type information to reduce the cost of method lookup.)

There are two efficiency difficulties with this. One is that the dynamic dispatch itself requires a certain amount of computational work — comparing a class identifier to the class identifier in an inline cache, for example, or comparing method selectors found in a hash table to the selector for the method being called. The other is that it necessarily involves a certain amount of pointer-chasing, which implies random memory access patterns. On a large memory, this inevitably results in higher latency. The single-threaded programming model we use turns that high latency into low throughput.

Actor message queues

Some versions of the Actor model, like the one implemented in Erlang, associate a message queue with each actor — rather than running the actor's code immediately when a message is sent, the message is added to a queue, and the actor's code can run later to process the queue. In Erlang, the process's code can actually select particular messages from the queue to process out of order, which is used to support a call-response pattern in which a client sends a message to a server process and then blocks until the reply is available.

Such queuing is a general-purpose system design pattern for maintaining throughput in the face of large or highly variable latency.

Fast dynamic allocation with good locality

Generational copying garbage collectors typically require only about two or three machine instructions to allocate memory in the nursery: one to save the old value of a register, another to add an immediate constant to it, and a third to compare its value against the end of the nursery to see if the allocation succeeded. In theory, if the nursery is large enough, the work to iterate over the root set and fish the few surviving objects out of the nursery will be amortized over enough objects that the garbage collection work per object is also very small.

This is still a great deal more work than is needed to allocate and deallocate an object on the stack in C or C++, which is typically zero, because the compiler can determine upon entry to the function how much space its local variables will consume and subtract the total from the stack pointer in a single operation.

However, maybe you can do the same thing for generational heap allocation — check upon entry to a function to see whether the nursery is big enough for everything you're going to allocate, do a minor GC if necessary, then preallocate all the objects you're possibly going to allocate in the function. If the collector happens to run

while you're still on the stack, it can relocate your entire "heap frame" out of the nursery, consulting a liveness map to see which objects in the heap frame have been initialized thus far in the function.

This should have the benefit that the objects you allocate in your function remain close together in memory until after the function returns.

Mergesort converts random memory-access patterns to near-sequential

This is how people did compilers and databases in the days when they only had a few kilobytes of RAM: they used multiple tape drives to store data for the different passes of the compiler and to sort the database into an order that allowed a single sequential pass over it to produce the answers for all the queries (called "reports", more or less).

Consider the basic problem of a linker: after concatenating its input files, it needs to combine a potentially large amount of object code full of relocations with symbols providing definitions for those relocations. One way to do this is to take the list of relocations, sort it in symbol order, and then merge it with the sorted list of symbols to produce a list of instructions roughly of the form "add 382 to location 10078". Then, sort this list of instructions by location, and then merge it with the object code.

If you have 100 megabytes of object code, 3 million relocations (totaling 36 megabytes), 300,000 symbols (totaling 4 megabytes), 64 kilobytes of RAM, and eight "tapes", then sorting the symbols requires three passes totaling 24 megabytes of sequential I/O (one to break them into 128-kilobyte sorted chunks, a second to merge those into 896-kilobyte sorted chunks, and a third to merge the five chunks); sorting the relocations requires five passes, merging them with the symbols requires one pass, and sorting them again requires five more passes, for a total of 796 megabytes of sequential I/O; and then you need a final pass to generate the linked object code.

Using actor queues to hide dynamic dispatch latency and increase locality

What if we use message queues instead of a call stack?

When you start running a function, it allocates a heap frame and starts processing its incoming message queue. While it's running, it can send many, many messages to other actors, but it can't get responses from them; it can allocate new objects in its heap frame that will later handle responses, if need be. It doesn't need to actually be able to access any information of those other actors; all of its outgoing messages could even be accumulated in an outgoing-queue buffer, which is perhaps part of that heap frame.

This requires that all of its loops be statically bounded from the time you enter the function. It's okay if they iterate some dynamically-determined number of times, but that number has to be determined up front, because it affects how much to allocate. This means that the function is guaranteed to terminate. This unforgivable incursion upon Turing-completeness is not really a problem, since the function can send messages to itself if need be. In the worst case, this

reduces to continuation-passing style.

Once it exits, perhaps it is time to sort the outgoing messages by destination. Then we can select a next function to run, ideally one with at least one message in its message queue. Maybe we select the function with the most messages in its message queue.

If the function's input data and computational structure are sufficiently regular, we can vectorize it, so that in fact it is processing many input messages in a single instruction. Perhaps eventually those many input messages will result in response messages going to many different destinations.

If the messages are like Smalltalk messages, consisting of a selector, a "receiver" (in Smalltalk terms) on whose class the selector is dispatched, and arguments, there are a few different ways the dispatching could be amortized. If the routing system can identify the method, it can have a single queue for that method, with messages for many different receivers mixed in it.

Note that in this model, unlike in Erlang, we can guarantee in-order message delivery between a given pair of actors, even though there's a great deal of nondeterminism about sequencing otherwise. But taking advantage of that requires state to exist in the actor, unlike the actor-per-method approach suggested in the previous paragraph, where the actor is a stateless method. (And it seems like with this amount of nondeterminism in ordering, you really want to keep local state to a minimum; maybe only a kind of barrier-synchronization primitive, which aggregates two or more expected messages from different sources, should be allowed to have any mutable state.)

Alternatively, you could have queues per selector, per object, or per class. The function handling the per-selector queue would forward the message, somehow, to the appropriate method; the per-object or per-class function could perhaps contain the code for all the methods.

This approach, despite its obvious drawbacks in nondeterministic ordering of operations and potential to be slower rather than faster, has some potential advantages:

- You can, obviously, replicate and distribute the actors across a network for (ha) performance.
- You can take advantage of heterogeneous hardware by running different actors on different hardware — some might run much faster, for example, on a GPU or an FPGA.
- You can distribute actors across *time* rather than *space* in cases where memory is limited — you need only have one actor in RAM at a time, as each message contains everything that is needed to process it. This could allow you to run large late-bound systems on very minimal hardware. (This is really just another way of looking at the benefits of running in the CPU cache on a modern big CPU.)
- Even very large setup and teardown costs could potentially be amortized over a large number of messages. For example, reprogramming an FPGA with a bitstream that implements a particular actor might take milliseconds — maybe seconds if you have to run project IceStorm to generate the bitstream — but if it's processing a million queued messages, that's likely okay.
- The ordering nondeterminism I moaned about over and over again amounts to separating the scheduling concerns about latency vs. throughput vs. memory usage from the algorithmic concerns about

what to compute. Perhaps being able to change your scheduling policy separately from the code will empower you to choose a better one.

Topics

Arduino curve tracer

Kragen Javier Sitaker, 2018-06-17 (10 minutes)

I was thinking about diode logic (see Diode logic (p. 272)) and I realized that I really need a curve tracer in order to theorize sanely about diodes and whatnot. But an off-the-shelf curve tracer is pricey.

You could generate carefully controlled ramps using PWM and filtering, but you don't need to; if you're willing to take the voltages that show up, you can just measure voltages at different points in a test circuit and see what happens. A simple test circuit would have a capacitor, an inductor, a resistor, and the device under test, in that order, and you'd apply voltages (5V or 0V) at the ends of the circuit and measure the voltages at the two ends of the resistor.

The idea is that the capacitor and inductor and so on turn your step-function voltage changes into a smoothly changing voltage, so you can see what the DUT does at a bunch of different voltages. The inductor also serves to limit current, as well as permitting variation of dV/dt at a given voltage, and the capacitor serves to limit the total charge you can put through the device, thus the energy dissipated, thus the damage done. The resistor also limits the current and additionally provides a way to measure currents.

An Arduino's built-in ADC can sample at 200 ksps, and you probably only really need 1000 samples to draw an adequate curve, which ends up being about 10ms if you're sampling two separate points. If you'd like this to be a single ramp, you probably want the resonant frequency $1/(2\pi\sqrt{(LC)})$ to have less than a single cycle in 10ms, i.e. to be less than 250Hz, but it's probably okay for it to be three or four or even ten cycles. And the resistor will probably slow it down further.

The ADC is most accurate when measuring against the 1.1 V bandgap reference, but of course we will potentially have voltages to the 5V rail and beyond — if the DUT is a piece of wire, then the voltage at that end of the resistor will be 5V all the time. So we should probably use the 5V reference and merely calibrate it against the bandgap reference periodically.

If we figure on currents up to about 2 mA, then a 2.2 kΩ resistor would scale those currents to be within the same 0–5V range. If we use a much smaller resistor, we start losing precision on the current measurements. Then if we want our RC time constant to be around, say, 2ms, then we need something like a 1 μF capacitor. Then, if we want $2\pi\sqrt{(LC)}$ to be something like 1ms, we need a honking 22 millihenry inductor. Maybe a better balance is to use a 1 kΩ resistor, a 3.3 μF capacitor, and a 4.7 mH inductor.

ω_0 = 8000 radians/sec, $Q = \omega_0 L/R \approx 0.04$. This suggests that with such low inductance this is going to behave basically like an RC circuit. You don't even get ringing until Q = 0.5, I think. So to get the potential benefits of inductance here, you really would need a bigger inductor, which is surprising to me, since I think of millihenries as largish.

The maximum energy stored in the capacitor at 5 V is 41 μJ, which seems comfortingly small. The 100-pF 1.5 kΩ Human Body Model

reaches that energy level at only 900 volts, so hopefully any device that is ESD-rated should be able to handle this without breaking down. Maybe even back-biased tantalum capacitors. This should be gentler than the HBM, because its initial current spike into a 1.5 kΩ load is 300 mA and 135 watts, while this circuit should be closer to 5 mA and 25 mW, or maybe twice that.

I wish I could use a smaller resistor, too, because I'd like to trace curves well beyond 5 mA.

So, next iteration of design: 220 Ω resistor, 1 μF capacitor, 47 mH inductor, resonant frequency 734 Hz, ω_0 = 4600 radians/sec, Q = 1.0, maximum current should be in the ballpark of 20 mA (the maximum on the datasheet for an AVR pin), 12 μJ in the capacitor when charged to 5 V, but we should be able to get a bit of overshoot from Q = 1.0; RC = 220 μs, which is about 44 samples from the ADC, so we probably have to do a few cycles to get good coverage of the curve.

If we were naughty we could perhaps get rather higher voltages out of the inductor by switching the pin on the other end of the DUT from its low-impedance "output" state to a high-impedance "input, no pullup" state.

With a low-impedance DUT, when voltage is initially applied, all of it will be across the inductor. This will result in a dI/dt of 5 V / 47 mH = 106 A/s or 106 μA/μs. Every 10 microseconds we will take a sample of voltage and a sample of current, with the current ramping up about 1.06 mA per sample at first, which works out to 230 mV across the resistor (about 48 counts out of 1024). This seems like it will give poor coverage of the originally planned 0–2 mA range, but we could use a short pulse to start a current through the inductor, then turn the output back off to allow the inductor current to decay under the influence of the DUT, the resistor, and the capacitor. If the current is at 5 mA, the capacitor is empty, and the DUT is low impedance, then the sense resistor will oppose with 1.1 V, which will cause the inductor current to decay more slowly at 23 A/s (23 μA/μs, 0.23 mA per sampling interval).

Also, though, for many kinds of DUTs, we should have plenty of time at low currents when the capacitor is almost charged.

I hacked together a shitty simulation (using difference equations to approximate the differential equations, with a timestep of 200 ns) and observe the following:

• The capacitor voltage overshoots the 5V rail by almost exactly a diode drop, 5.79 volts. In a sense that's about the best we can hope for — it'll start to hit the AVR's input-clamping action there.
• The simulation sees the current continuing to ring down for 9 full oscillations before it dies from roundoff error, but only the first two full oscillations are above 5 mV, which is one count on the AVR's ADC when it's using 5V as the reference. Also they are negative half the time, although they never reach a full diode drop below ground.
• Those 9 full oscillations reach about 15 ms, which would be about 600 Hz, suggesting that either my simulation is crappy or my calculation of the resonant frequency is not a good way to calculate the intervals between zero-crossings during ringdown. Lowering the resistance to 47 Ω shortens that to about 12 ms, which would be about 820 Hz, which seems right.

• The peak current is a bit over 12 mA, which is better than 5 mA, but still leaves out a lot of interesting territory for many devices.

Slow devices — over a millisecond or two — may be hard to test in this circuit.

(Discarded idea: how about if we use 220 Ω, 47 mH, but 4.7 μF? This eliminates the overshoot entirely (and thus presumably the possibility of doing much interesting with the inductor) but does give us slower ramps — we have like 6 ms of V_R being over 5 mV. I didn't expect that adding more capacitance would make it stop ringing entirely, but I guess it lowers the resonant frequency, and consequently Q if you don't change R and L.)

I tried driving this (simulated) circuit with a squarewave chirp and got peaks of current up to 15 mA where the period was about 3.6 ms. However, this also resulted in inductor voltages down to -6 volts. Better make sure that inductor-capacitor point isn't connected to an AVR pin! Worse, though, even the resistor voltage got down to about -3 volts, and that point *does* have to be connected to a pin. We can perhaps raise the whole RLC circuit bodily to Vcc by raising the pins at each end to 5V, thus allowing us to measure these reverse currents and avoiding clamping.

(You do want to make sure you have pins connected separately on both sides of the DUT and both sides of the capacitor, which amounts to three of the four pins you need to make this work at all. Otherwise you have no way to discharge the capacitor in a reasonable amount of time if the DUT is diodish.)

Topics

Recuperator heat storage

Kragen Javier Sitaker, 2016-11-01 (updated 2019-08-21) (4 minutes)

Stirling engines and adiabatic compressed-air energy storage both depend for their efficiency on a regenerator, which can be something as simple as a pebble bed or packed column, or as elaborate as a ceramic honeycomb.

I think regenerators have some significant tradeoffs, though. In particular they tend to impose a fairly large pressure loss on the air (or other fluid) passing through them, and they cool off significantly if left to sit, just through the diffusion of heat through the regenerator itself. And I think — though I'm not sure — that the thermal mass of the regenerator itself slows down the ramp time of Stirling engines.

You can cut a regenerator's head loss by making it shorter, so the air travels less distance while constricted by it, but this puts a low ceiling on the total amount of thermal energy that can be stored, and also worsens the diffusion problem.

A countercurrent heat exchanger with a different, probably liquid, coolant — a recuperator — could solve these problems. This allows you to keep separate superinsulated hot and cold reservoirs, connected to the heat exchanger with long, thin pipes, and pump the coolant either direction through the pipes to keep the heat exchanger at a constant temperature.

Countercurrent heat exchangers with very low pressure drops and very high heat fluxes are feasible — in biology they are called "retia mirabilia" and were discovered 1800 years ago. They intertwine two fractal branching structures in such a way that they come in contact throughout a surface with a large fractal dimension vaguely resembling the surface of a piece of broccoli. As far as I know, nobody has ever built an artificial rete mirabile, although there have been a number of papers and books on process intensification that come close. It will be best to build them from a solid material with low thermal conductivity such as a glass or ceramic, since if you can get the fluid passages below 100µm in diameter, the distance between them is very small and the surface area between them is very large, so it's probably more important to slow lengthwise heat diffusion than to promote transverse heat diffusion.

(See Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) for more about such heat exchangers.)

If the secondary coolant is a liquid, it may be necessary to use more than one liquid, because most liquids have a narrow usable temperature range. For example, ethanol spans -120° to about +100°, propylene glycol spans -59° to +188°, and glycerol spans about 0° to +290°, but organic liquids in general start not merely to boil but to break down chemically somewhere between 200° and 300°. Molten nitrate salts span a somewhat larger temperature range but are solid anywhere near room temperature. Liquid metals cover the 200° to 1000° range reasonably well, but most of them are also solid at room temperature, potentially posing difficulties for a cold start.

I think this is important because Carnot efficiency is $1 - T_k/T_h$, where T_k is the temperature of the kold reservoir and T_h the temperature of the hot one, so a wide temperature swing is crucial for

heat engine efficiency; this clearly applies to Stirling engines, but I'm not yet clear on whether it's necessary for adiabatic compressed air energy storage.

Topics

Applying FM synthesis to natural sounds such as voices

Kragen Javier Sitaker, 2019-11-12 (2 minutes)

A common technique in modern music (going back to the Beatles) is to sweep a comb filter over the frequencies. One variant of this is for no particularly good reason called "flanging", but there are others.

Another common technique is to use FM synthesis, in which the time axis is distorted according to a modulator waveform. Typically this differs from the kind of FM used in FM radio in that, in FM radio, the modulator is many octaves lower than the carrier, while in FM synthesis, typically they are the same frequency or an octave or two apart, so the waveform repeats with the same period as the carrier — so the FM distortion merely produces harmonic distortion, rather than inharmonic sidebands. A typical technique is to gradually reduce the "depth of modulation" down to zero, thus causing the harmonics thus produced to die away, just as the overtones of a string or a bell do.

An interesting thing about FM synthesis is that the carrier wave being distorted is almost invariably a simple sinewave. It would be interesting to use some kind of frequency and phase tracking to apply the same kind of "FM" periodic time-domain distortion to a wave from some other source, such as a singer's voice.

Most children in the rich world have done something similar by talking or singing through the blades of a spinning box fan to make a "robot voice", which provides a sawtooth "FM" distortion of time at frequencies somewhat lower than voice frequencies, perhaps 50–100 Hz, and without phase tracking of them.

Topics

Heliogen

Kragen Javier Sitaker, 2019-11-19 (6 minutes)

From a comment on the orange website.

Solar furnaces have been useful for reaching especially high temperatures since at least Trombe's 1949 furnace, which can hit 3500°, without computer vision or even closed-loop control. So why is a 1500° solar furnace being touted as a groundbreaking innovation and a new high-temperature landmark?

A friend asked me what I thought about this earlier, based on a somewhat better reprinting of their press release.

Scroll back to 2010, when Bill Gross started working on this. That's when he got funded by that dude whose futurism book about the Information Superhighway, *The Road Ahead*, didn't mention *the internet*. In 1995. In 2010, photovoltaic modules cost €1.62 per watt. Concentrating solar power was a promising alternative; it uses the same steam engines used by coal and nuclear power plants, so at scale it should be just as cheap as they are, as long as you can get the cost of the heliostats under control somehow and scale up. It also didn't have that pesky intermittency problem PV modules have: you can store the heat overnight.

Since then, though, heat engines have become economically uncompetitive relative to PV, because PV modules now cost €0.19 per watt, where they've been stuck all year. And steam turbines, almost a century and a half old, aren't improving or getting cheaper rapidly the way PV has been. Being just as cheap as coal isn't a blessing anymore; it's a handicap.

So, if you've been working on CSP and filing patents for a decade before getting your pilot plant up and running, a decade during which the PV market has left your product's price in the dust, what do you do? You look for a possible use where CSP is still viable, such as process heat; you hire a good PR firm; you announce that you won't be building any plants, but you're "willing to partner with" companies that want to build your design; and you hope to God nobody says "Solyndra".

But what's the actual invention? It seems like the actual news is that Bill Gross has patented some aspect of his closed-loop control system using webcams and GPU-accelerated CV to figure out where the mirrors are pointing to improve your concentration factor. The key improvement that made it possible was better GPUs, according to the press release, anyway.

So what happened, from the point of view of anyone outside Idealab, is that now Idealab and Intellectual Ventures will sue you if you do this fairly obvious thing of using high-resolution webcams for precise heliostat control.

So, when would this be a sensible thing to do?

Trombe's solar furnace and similar devices are able to compete quite effectively in the "market" for process heat at the 2500–3500° level, since, as I said, 1949. (I guess Bill's PR firm didn't know this, or hopes you don't.) That's a level almost impossible to achieve using fire (oxy-acetylene burns at 3500° under ideal conditions), and difficult even with arc furnaces. But Bill's thing is designed for a more prosaic

1000–1500° level, where it's competing not only with fire but also Kanthal or SiC fed from PV, wind, hydro, and nukes, as well as induction, dielectric heating, and microwave heating.

The potential advantage of CSP for process heat at these lower temperatures is that it's cheap and abundant. If you fill a field with mirrors, they can harvest 6× as much power than PV modules covering the same field can. But if land area is your limiting factor, your cement plant or steel mill or whatever probably isn't in the middle of a big field; it's using a lot more energy than your land receives in sunlight. In that case, you probably want to pull your power from someplace further off, whether in the form of coal, oil, gas, biomass, or electricity. Probably electricity from PV panels if we're talking about anything post-2030.

But suppose you *can* put your factory in the field where the mirrors are, and the limiting resource isn't land but money. In that case, it might be a reasonable approach. PV modules cost €30 a square meter now. That's probably more expensive than mirrors, if you take into account that mirrors give you 6x as much energy: €180 per square meter is the price mirrors have to beat, and that seems doable.

But now you are on notice: if you do that, make sure it's in a country where Intellectual Ventures's shell companies haven't gotten a patent on it, or you have to deal with patent trolls. The press release reprinted above is clear: as with IV's laser mosquito swatter, they aren't going to make it happen themselves, but they'll definitely "partner with" you if you try.

I think we're about to see a giant boom in shitty "do well-known thing X, but with computer vision" patents similar to the shitty "do well-known thing X, but on a computer/on the internet" patents that plagued us in the early 2000s. The availability of massive GPU power means that many things that used to be impractical to do with video data have become possible.

Topics

• Energy (p. 3438) (63 notes)
• Solar (p. 3717) (30 notes)

Offline datasets

Kragen Javier Sitaker, 2014-04-24 (15 minutes)

What currently existing datasets would most effectively use the capacity of a modern laptop disk, if it was going to be disconnected from the internet? What would most powerfully augment its possessor?

8.4 GiB: Project Gutenberg 2010 DVD, 29,500 ebooks, most pre-1921 English lit

12.5 GiB: Debian 7.0.0 amd64 DVD images, all free software in Debian, compiled

34.1 GiB: Debian 7.0.0 source DVD images, the source for the same software

9.7 GiB: the articles in the English Wikipedia, an overview of all knowledge

13.2 GiB: StackExchange dumps, all common technical questions with answers

19.9 GiB: Planet.osm, a map of most of the streets and rail lines in the world

6.8 GiB: Open Library latest dump, bibliographic data on all published books

15.0?GiB: Freebase in Turtle RDF, a unified database about everything known

20.0 GiB: wikileaks-files-20100612.tar, Wikileaks Full Archive, including cables

139.6GiB: total

http://aws.amazon.com/datasets lists a number of freely-available datasets on Amazon S3; not only can you download them directly from there, but you can also fire up an EC2 machine with gratis network access to them in order to analyze them, extract relevant parts, and summarize and compress. For example, the 2.2 terabyte Google N-Grams corpus is available, and you could quite reasonably fire up a machine to fetch all the 3-grams that occurred more than 5 times and return them in a compressed format. (A particularly interesting free dataset is the 81-terabyte Common Crawl Corpus, containing 5 billion crawled web pages.)

http://arxiv.org/help/oa/index describes the arXiv Open Archives Initiative interface, which is suitable for bulk-downloading of all abstracts of the 850 000 papers in the arXiv (essentially all significant current math and physics papers, and a smaller but still significant fraction of other academic papers.) In 2010, the papers totaled 200GiB , but there was no bulk download interface publicly available. Later, http://arxiv.org/help/bulk_data_s3 explains that they are now available in S3, including both the source LaTeX and the rendered PDF versions.

Citeseer has a bibliographic record for the majority of scientific publications in any field, including pointers to online versions of the articles where available. They, like the arXiv, support the OAI for bulk downloading of this data. There seems to be a dump from 2010 in http://www.cs.purdue.edu/commugrate/data/citeseer/, which looks like it might be around a hundred megs compressed. http://www.hrstc.org/node/33 explains how to do an OAI bulk download and reports that, as of 2009, it ended up as about 500MB of

uncompressed XML.

IMDB has bulk downloads available, and I guess movies are pretty popular: http://www.imdb.com/interfaces. The data isn't free, but it's available for some uses. It looks like it's a gigabyte or two.

Project Gutenberg has gone from 30 000 titles to 43 000 since 2010, and many of the new titles and versions include illustrations, but I don't know where to do a bulk download of all of this data. It has a substantial number of non-English books these days, too.

http://datahub.io/, organized by the Open Knowledge Foundation, has a collection of already-structured datasets, but most of them are small and they are somewhat spammy. http://www.infochimps.com/datasets is another similar, but apparently somewhat better, collection.

http://dbpedia.org/About is an effort to extract structured data from Wikipedia, which has substantial overlap with Freebase. Shallow discussion of the relationships between the two projects can be found at http://wiki.freebase.com/wiki/DBPedia and http://www.google.com/url?sa=t&rct=j&q=dbpedia%20freebase&so
ource=web&cd=3&ved=0CD0QFjAC&url=http%3A%2F%2Fblog.db
pedia.org%2Fcategory%2Finter-linkage%2F&ei=mU_aUZqYF6iXi
QKJwoCoDw&usg=AFQjCNHcf8Bti6uQcqv-aDUjwG4lg_YY7A
&bvm=bv.48705608,d.cGE&cad=rja. Among the interesting sub-datasets in DBpedia are "bijective inter-language links", "short abstracts", and "geographic coordinates". I can't tell how big the whole DBpedia dataset is, but it looks like it should be a few gigabytes. It also sort of looks like the project is faltering, since the latest DBpedia release is at http://downloads.dbpedia.org/3.8/en/?C=S;O=A, and it's a year old, despite their quarterly release schedule.

http://www.clearbits.net/torrents/680-california-learning-resource-
network-textbooks is among the interesting things on ClearBits other than StackExchange; it's 0.8 gigabytes of supposedly high-quality secondary-school textbooks in English, called the California Learning Resource Network Textbooks, from the California Free Digital Textbook Initiative. Another 0.3 gigabytes at http://www.clearbits.net/torrents/158-physics-textbooks covers secondary-school physics.

There's a couple of attempts to put the Khan Academy in an offline-accessible form. One is Khan Academy on a Stick, which is just a 16GiB selection of 2000 English video lectures, and a similar set of 800 in Spanish. A much more ambitious project is KA-Lite, which includes exercises, progress tracking, multilingual subtitles, and the ability to make your own selection of videos; I don't have a clue how much space this stuff takes up, aside from the videos, but I imagine not much on the scale we're talking about here.

http://www.clearbits.net/torrents/571-fsi-mandarin-chinese---comp
lete-course is a public-domain Mandarin Chinese course in 1.6 GiB.

GenBank is the NIH's annotated collection of all publicly available DNA sequences. A 200GiB 2009 snapshot of GenBank is on S3.

http://earthobservatory.nasa.gov/Features/BlueMarble/ has a 500-meter-resolution true-color map of the earth made from MODIS satellite data, with month-by-month composites. This could

be a nice supplement to OpenStreetMap data, but I'm having a bit of a hard time downloading it. Calculations suggest that in JPEG form it should be about 9 gibibytes, since each world coverage should be about 0.7 gibibytes. Now that Landsat data is open-access, and even available in pre-downsampled form, it should be straightforward to produce a higher-resolution version; if we budget 20 GiB, to be comparable in weight to the OpenStreetMap planet.osm data, we could manage about 100-meter resolution. If you were more judicious, you could skip the 100-meter resolution on open water and use data down to Landsat's 15-meter resolution limit in areas the OSM data shows to be dense.

http://geonames.org/ is a CC-BY collection of eight million place names, with coordinates, which is a lot more than you can get from Wikipedia. I think it's about a gigabyte.

Not all the software in Debian is adequately documented within Debian itself. In particular, Debian includes no tutorial for C, as far as I know, although I am pleasantly surprised to find that the `c++-annotations` package contains a tutorial for C++ for people who already know C; and the RFCs that document much of what you need to know about networking are relegated to the "non-free" Debian repository, which is not included on the DVD images. I have no idea how big non-free is.

I think the WikiLeaks snapshot (from 2010, when they were having a hard time keeping the site up in the face of censorship attempts from the USG) includes the full Cablegate file only in encrypted form. After David Leigh negligently published the encryption key in a book, WikiLeaks re-released the full, but censored, Cablegate archive (only 0.6 GiB) at . These leaked US diplomatidc cables have been a major primary source for journalists writing new articles over the last few years, as well as a major source of civil unrest in US "allies".

A similarly important set of primary sources may be WikiSource. The English Wikisource dump is currently 1.3GiB http://dumps.wikimedia.org/enwikisource/20130629/ but doesn't include images, because Wikimedia dumps never include images. But that's probably okay in this case, because most of Wikisource seems to be transcriptions rather than scans.

What about music?

• The Archive Team archive of all the MIDI files they found on GeoCities, which probably covers most 20th-century US popular music and a fair bit of other music (albeit in low quality), is fairly tiny and contains some fifty thousand songs. It's very likely illegal, though.
• http://www.infochimps.com/tags/music?page=4 has some other related datasets.
• http://cantorion.org/ has a large collection of scanned public-domain sheet music, mostly classical, and collections of it have been uploaded to the Internet Archive.
• http://discogs.com/ has a public-domain database of metadata on 35 million recorded tracks, with dumps at http://www.discogs.com/data/, which are some 0.2GiB.
• http://freesound.org/ has a mixed CC-sampling-plus and CC0 database of samples for making music with, which may or may not be

bulk-downloadable.
• However, there's an 8.5GB chunk of sound samples released under CC-BY for OLPC at http://wiki.laptop.org/go/Sound_samples.
• http://imslp.org/ has 220 000 musical scores, mostly of public-domain music, which may or may not be bulk-downloadable; there's a very old torrent backup from 2008.
• Musicbrainz I think is already incorporated in Freebase, but it has rich metadata on some 12 million recorded tracks.
• The MOD Archive had some 120 000 songs in MOD and related formats as of 2007, totaling 31.2 GiB; these are somewhere in between MP3 and MIDI in their level, since they include the samples. But the MOD Archive is also probably almost entirely illegal, because the samples come from all over the place and were frequently copied from one MOD musician to another without consent.
• Mutopia has some 1600 public-domain musical scores which can be played with MIDI or typeset with LilyPond. I assume these can be bulk-downloaded, but they're tiny.
• Musopen has a collection of public-domain music recordings, but no bulk download as far as I can tell.

Some very valuable kinds of information that may not be present in any of the above:

• Transport schedules, e.g. bus and train schedules, not to mention route information (OSM has some route information);
• Contact information for people and businesses (i.e. phone books; available from, say, Yelp or Foursquare).
• Reputation information for people and businesses (i.e. Yelp), or even for pages (e.g. the freely-available page hits data for Wikipedia articles, for use in ranking search results.)
• In-depth engineering data, e.g. Machinery's Handbook, the CRC Handbook, Organic Syntheses, electronics datasheets, and so on. US military MILSPECs may provide some useful data here.
• Pricing information about pretty much anything: minerals, chemicals, electronic parts, commodities, stocks, forex, household appliances, foods.
• Social graph information. Presumably any number of spy agencies have a full crawl of the social graph from Facebook, including everybody's profile photos, but Facebook's terms of service forbid doing this, so nobody admits to it. But this is in the terabytes, anyway, since you have at least thousands of bytes for the profile picture times about a billion accounts. If you just wanted uncompressed names and graph links, without geographic or photo info, it's probably on the order of 64 links (of 4 bytes each) per person, total 256 bytes, plus their name, so about 200 to 300 GB.
• Travel information, like WikiTravel: what to see when you get to Córdoba, how to get there, what neighborhoods to avoid, how the local taxi system works, where to stay. WikiTravel is licensed under a CC-BY-SA license that would permit this in theory, but I'm not sure how to download a snapshot dump of it in practice; they seem to be trying to make it difficult. Hitchwiki might be an alternative, but it seems nascent — its dump is only 15MB.
• Cooking recipes, other than the very old ones from Project Gutenberg.
• Guides to identifying natural things, such as plants, animals, rocks,

and diseases.
• Much in the way of modern how-tos. en.wikibooks has almost 2700 books at the moment, and might help some; but if you want to know how to fix your 1997 Ford Taurus, how to incorporate a company in Argentina, how to upgrade the hard disk in your Dell Mini 9, etc., you're out of luck. A fair bit of this kind of information may be available in the public domain from the US Military, but finding and organizing it will be difficult.
• Nutritional information.
• Public records information, such as land titles, lawsuit records, invention patents, EDGAR filings, trademark filings, marriage records, statute law, caselaw, or international treaties.

More possibly relevant URLs:

Datasets for data mining http://www.kdnuggets.com/datasets/

A billion-web-page snapshot amounting to 5 terabytes http://lemurproject.org/clueweb09/

http://stackoverflow.com/questions/2674421/free-large-datasets-to-experiment-with-hadoop

The Berkeley Earth Temperature Study dataset http://berkeleyearth.org/dataset/

UN treaties data https://github.com/zmjones/untreaties

70 years of historical stock price data on Quandl http://www.reddit.com/r/datasets/comments/1egihx/15000_stocks_x_70_indicators_x_10_years/

IRS nonprofit filing data http://projects.propublica.org/nonprofits/

USDA nutrient data https://github.com/thebishop/usda_national_nutrients

Social networks http://arcane-coast-3553.herokuapp.com/sna/visual

Movie subtitles http://www.reddit.com/r/datasets/comments/1efi20/looking_for_movie_subtitles/

http://www.quora.com/Data/Where-can-I-find-large-datasets-open-to-the-public?share=1

Stanford Large Network Dataset Collection http://snap.stanford.edu/data/

http://lemire.me/blog/archives/2012/03/27/publicly-available-large-data-sets-for-database-research/

3D models of furniture (2.97 GB) http://kickass.to/avshare-furniture-3d-models-t7291976.html

1.44 GB of 3D models http://kickass.to/large-collection-of-3d-models-t1709247.html

A much smaller-scale version of this problem is: what should I put in my next paper notebook? I'd like to print some things out: obviously my friends' phone numbers and other contact information, a map of the city and surrounding areas, and so on. I can print them in reduced size, as long as I can still read them; I think I can distinguish 600 pixels per inch with a magnifying glass, and ordinary laser printers can print that. My current notebook is 288 pages, which is to say 144 leaves, or 72 sheets of paper, which are roughly A5-size,

one thirty-second of a square meter, so 2¼ square meters of paper in total, or 4½ square meters of paper surface. The smallest reasonable ASCII font is about 6×4 pixels, which works out to 23.25 million characters per square meter, or about 105 megabytes of text. If we use the traditional 80×66 unit for a "page", that's 19 815 pages.

Now, I only want to fill a fraction of the notebook with text. So what are the most useful, say, ten to thirty megabytes I could print out and bind into my new notebook? The Wikipedia Vital 100 articles at http://en.wikipedia.org/wiki/Wikipedia:Vital_100 might be a reasonable thing to include, for example. Ten articles chosen from there by random clicking are Fire (12 pages), Crime (16 pages), Biology (18 pages), Human sexuality (38 pages), Earth (32 pages), History of the World (27 pages), History of art (16 pages), Philosophy (33 pages), Mathematics (12 pages), and Energy (19 pages). These sum up to 223 pages, suggesting that the Vital 100 in total will be about 2230 pages. This would be a fairly straightforward thing to include in the notebook in its reduced form; if we figure 56 reduced pages per notebook page, it would occupy about 40 pages.

You might be able to do something similar with

Topics

Constant current switching capacitor charging

Kragen Javier Sitaker, 2017-07-19 (1 minute)

Lots of energy-harvesting devices use capacitors for energy storage. The simplest way to do this, given a somewhat unpredictable AC voltage source, is with a diode or bridge rectifier. This has a couple of big problems, though: the diodes dissipate more than half of the energy you're trying to harvest, and without any significant linear resistance, the capacitor gets fully charged during the first quarter-cycle, so if the capacitor is large, they're dissipating it very rapidly and will explode.

(A reason you might not see this happen in practice is that your energy source isn't actually a voltage source, i.e. negligible impedance; more on that later.)

If you have a diode or resistor or some other passive element in series with the capacitor, it's going to take up whatever the voltage difference is between the capacitor's current state of charge and the voltage source. Ideally you would like that to be just enough to push the right amount of current through the passives to charge the capacitor, so that almost all the energy gets harvested instead of dissipated.

You could maybe do it literally like that using a buck regulator: instead of regulating the duty cycle of the PWM signal in the buck regulator to seek a fixed goal voltage, regulate it to seek a fixed goal current.

Topics

Practically decodable random error correction codes with popcount

Kragen Javier Sitaker, 2015-07-01 (updated 2015-09-03) (6 minutes)

I tried designing a super-simple holographic ECC scheme, and although it does work, it bloats the message by an order of magnitude before it stops being an error-introduction code instead of an error-correction code. The scheme is described below, in case someone wants to try to rescue it.

RESCUE: MAKE THE MATRIX SPARSE AND COMPARE POPCOUNTS OF ZERO-DERIVED AND ONE-DERIVED CODEWORDS! popcount(num & row) > popcount(~num & row).

To encode a block of N (preferably N is odd) bits into a block of M bits, where normally M>N, use a M×N matrix of random bits defining the code. Invert (NOT) each of the N columns that corresponds to a 1 bit, which is to say, XOR each of the M rows with the N bits you are attempting to encode. The M-bit codeword is then the majority-rule of each of the M rows of the output matrix: if it contains more 1 bits than 0 bits, then the output is 1, and otherwise 0. To decode, you apply the same process, but with the original matrix transposed.

Why should we expect this to work? The output bits are a holographic representation of the input bits. Each bit in the matrix represents a coupling between the probability that a given output bit is 1 and the input. If a column happened to be all 1s, then it would slightly increase the probability for each output bit to be 0 when the corresponding input bit was 1, or 1 when the corresponding input bit was zero. If there are enough output bits, and the other columns are uncorrelated, then this will probably flip a few of them — enough that the majority of output bits will correctly reconstruct the original input bit.

This can be implemented with somewhat reasonable efficiency (a few machine instructions per bit) on normal CPUs now that the NSA has finally pushed a POPCOUNT instruction into them; or at extremely high speed in hardware.

This Python code, encoding and decoding a message, shows that this works in practice if M is large enough. However, it seems that M needs to be almost always 8× and often 16× larger than N for it to work, so in practice this code is dramatically worse not only than Reed-Solomon codes, but in fact worse even than just repeating the message several times.

```
#!/usr/bin/python
from __future__ import division

import random

def main(N, M, message='This is a test message'):
    print 'N =', N, 'M =', M
    r = random.SystemRandom()
```

```
    key = [r.randrange(2**N) for ii in range(M)]
    print 'key', key
    unkey = transpose(key, N)
    print 'unkey', unkey
    # If this fails, it means transposing has a bug, so nothing can
    # work.
    assert key == transpose(unkey, len(key))
    print 'message', `message`
    message_digits = to_base(2**N, bytestring_to_int(message))
    print 'digits in base', 2**N, message_digits
    encoded = [encode(digit, key, len(unkey)) for digit in message_digits]
    print 'encoded', `encoded`
    decoded = [encode(item, unkey, len(key)) for item in encoded]
    print 'decoded', `decoded`
    decoded_bytes = int_to_bytestring(from_base(2**N, decoded))
    print 'decoded bytes', `decoded_bytes`
    print 'matched' if decoded_bytes == message else 'mismatch', (
        'N =', N, 'M =', M, '+%.2f%%' % (100*(M/N-1))
    corrupted = [item ^ (1 << r.randrange(128)) for item in encoded]
    print 'corrupted', `corrupted`
    decodedc = from_base(2**N, [encode(item, unkey, len(key))
                                for item in corrupted])
    print 'corrected bytes', `int_to_bytestring(decodedc)`

def bytestring_to_int(s):
    return from_base(256, (ord(b) for b in s))

def from_base(base, digits):
    "Expects digits in big-endian order."
    i = 0
    for digit in digits:
        i = i * base + digit
    return i

def int_to_bytestring(i):
    "Loses trailing NULs."
    return ''.join(chr(b) for b in to_base(256, i))

def to_base(base, i):
    "Returns digits in big-endian order."
    digits = []
    assert i >= 0
    while i:
        i, digit = divmod(i, base)
        digits.append(digit)
    digits.reverse()
    return digits

assert int_to_bytestring(bytestring_to_int('hello')) == 'hello'

def popcount32(num):
    assert num < 2**32
    num = (num & 0x55555555) + ((num & 0xAaaaAaaa) >> 1)
    num = (num & 0x33333333) + ((num & 0xCcccCccc) >> 2)
    num = (num & 0x0f0f0f0f) + ((num & 0xf0f0f0f0) >> 4)
```

```
    num = (num & 0x00ff00ff) + ((num & 0xff00ff00) >> 8)
    num = (num & 0x0000ffff) + ((num & 0xffff0000) >> 16)
    return num

def popcount(num):
    n = 0
    while num:
        n += popcount32(num & 0xFFFFFFFF)
        num >>= 32
    return n

def encode(num, matrix, bitwidth):
    threshold = bitwidth / 2
    bits = [popcount(num ^ row) > threshold for row in matrix]
    return int(''.join('1' if bit else '0' for bit in bits), 2)

def transpose(matrix, bitwidth):
    return [int(''.join('1' if 2**ii & row else '0' for row in matrix), 2)
            for ii in range(bitwidth-1, -1, -1)]

if __name__ == "__main__":
    for N in [3, #5, 7, 11, 17, 31,
              63, 127, 255]:
        for M in [7, # 15, 31, 63, 95, 127, 191, 255,
                  1023, 2047]:
            main(N=N, M=M)
```

I learned about codes like these from reading a paper of Pentti Kanerva's on "Fully Distributed Representation" around 2000.

Possible ways to rescue this scheme might include:

• A smaller number of points in the output vector, but more than one bit of information for these points?
• Deploying the single bit of information per point more judiciously? For example, maybe only set 1 when the population count exceeds some threshold? (This seems doomed to worsen the code rather than improving it!)
• Matrix columns that are exactly uncorrelated instead of approximately uncorrelated? For example, Hadamard words.
• Actually working out the probability distributions for output bits and thinking about that?

Topics

Developing Win32 programs on Debian

Kragen Javier Sitaker, 2007 to 2009 (12 minutes)

So MinGW originally was part of the GNU-Win32 project, aka Cygwin. Cygwin provides a Unix environment inside of Microsoft Windows, on top of Win32; I used to use it a lot when I used Windows NT at work. Cygwin includes a full GCC compiler suite and everything, but the executables it produces depend on the Cygwin DLL to provide the POSIX API.

MinGW, the "Minimalist GNU-Win32", instead let you use GCC to build applications that are written to use the Win32 API instead of the POSIX API.

This was all very well, but not of much use to those of us who used free operating systems like Linux and therefore didn't have an implementation of the Win32 API at hand.

But gradually, over the years, WINE got better and better, and now it implements the great majority of the Win32 API (although not enough to run the great majority of Win32 programs). And Linux's flexible binary format support makes it relatively transparent.

So now, on my Debian Stable system (that is, "etch", which is mostly from 2006), I can do this:

```
kragen@thrifty:~/notes$ cd ~/devel
kragen@thrifty:~/devel$ cat > hello.c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("hello, %s\n", argc > 1 ? argv[1] : "world");
    return 0;
}
kragen@thrifty:~/devel$ i586-mingw32msvc-gcc hello.c -o hello.exe
kragen@thrifty:~/devel$ ./hello.exe
Invoking /usr/lib/wine/wine.bin ./hello.exe ...
hello, world
Wine exited with a successful status
kragen@thrifty:~/devel$ ./hello.exe Win32
Invoking /usr/lib/wine/wine.bin ./hello.exe Win32 ...
hello, Win32
Wine exited with a successful status
kragen@thrifty:~/devel$
```

(I have both mingw32 and wine installed.)

Some GDI Hello Worlds

I googled up some Win32 tutorials, got some code that almost ran, and hammered it into shape and got this, which I can compile and run:

```
// Win32 GDI hello world program, slightly modified to compile in C
// (C9x), without MSVC, not be formatted like total ass, and add some
// obscenities, by Kragen Javier Sitaker
```

```
// But the original is written by RoD@cprogramming.com, to whom I am
// greatly indebted for sharing his knowledge of Win32 (despite my
// complaints about the formatting); it is available at
// <http://www.cprogramming.com/tutorial/opengl_first_windows_app.html>

// I compile and run it on my Linux box with MinGW and WINE as follows:
// kragen@thrifty:~/devel$ i586-mingw32msvc-gcc -Wall hi.c -lgdi32 -o hi.exe
// kragen@thrifty:~/devel$ ./hi.exe

#include <windows.h>

// event handler
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
  PAINTSTRUCT paintStruct;
  HDC hDC;                      // device context
  char *string = "Hello, World!";

  switch(message) {
  case WM_CREATE:               // window is being created
    return 0;
  case WM_CLOSE:                // window is closing
    PostQuitMessage(0);
    return 0;
  case WM_PAINT:                // window needs update
    hDC = BeginPaint(hWnd, &paintStruct);
    SetTextColor(hDC, (COLORREF)0x00FF0000);  // blue
    // (150, 150) is more or less the middle of the 400x400 window
    TextOut(hDC, 150, 150, string, strlen(string));
    EndPaint(hWnd, &paintStruct);
    return 0;
  default:
    break;
  }

  return DefWindowProc(hWnd, message, wParam, lParam);
}

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow)
{
  HWND hWnd;
  MSG msg;
  WNDCLASSEX windowClass = {    // What a fucking pain in the ass.
    .cbSize = sizeof(WNDCLASSEX),
    .style = 0,
    .lpfnWndProc = WndProc,
    .cbClsExtra = 0,
    .cbWndExtra = 0,
    .hInstance = hInstance,     // handle to the application itself
```

```
    .hIcon = LoadIcon(NULL, IDI_APPLICATION),
    .hCursor = LoadCursor(NULL, IDC_ARROW),
    .hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH),
    .lpszMenuName = NULL,
    .lpszClassName = "MyClass",
    .hIconSm = LoadIcon(NULL, IDI_WINLOGO),
  };

  if (!RegisterClassEx(&windowClass)) return 0;

  // What shithead thought it was a good idea to write a function with
  // ten positional parameters?
  hWnd = CreateWindowEx(0,                    // extended style
                        "MyClass",            // class name
                        "A Real Win App",     // app name
                        WS_OVERLAPPEDWINDOW | // window style
                        WS_VISIBLE |
                        WS_SYSMENU,
                        100,100,              // x/y coords
                        400,400,              // width,height
                        NULL,                 // handle to parent
                        NULL,                 // handle to menu
                        hInstance,            // application instance
                        NULL);                // no extra parameters
  // XXX probably should use ShowWindow, not WS_VISIBLE

  if (!hWnd) return 0;

  // main message loop
  for (;;) {
    // XXX probably should call GetMessage, not PeekMessage
    // XXX and look for <= 0 return value to exit loop
    PeekMessage(&msg, hWnd, 0, 0, PM_REMOVE);
    if (msg.message == WM_QUIT) return msg.wParam;
    // Translate and dispatch to event queue
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

That builds and runs. A little later I found this much smaller Win32 "hello, world" in what looks like a much more competent tutorial:

```
// http://www.winprog.org/tutorial/start.html

#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Goodbye, cruel world!", "Note", MB_OK);
    return 0;
}
```

That builds and runs too.

For some reason, `winedump` reports that they belong to "subsystem 0x3 (Windows CUI)" rather than "subsystem 0x2 (Windows GUI)". I found that I could fix this by using the `-Wl,--subsystem,0x2` option on the GCC command line. This also keeps Windows XP from opening a console window for it.

Upsides

Most obviously, you can write, test, debug, and compile programs for Win32 so that you can run them on Win32-only machines.

Also, you get fantastic stack dumps when you crash. Here I've deliberately introduced a null pointer write to demonstrate:

```
wine: Unhandled page fault on write access to 0x00000000 at address 0x4012df (thr
ead 0009), starting debugger...

Unhandled exception: page fault on write access to 0x00000000 in 32-bit code (0x0
04012df).
Register dump:
 CS:0073 SS:007b DS:007b ES:007b FS:0033 GS:003b
 EIP:004012df ESP:0069fc30 EBP:0069fc88 EFLAGS:00010246(   - 00      -RIZP1)
 EAX:00000000 EBX:7ebbe83c ECX:00000000 EDX:00401280
 ESI:00000000 EDI:00010024
Stack dump:
0x0069fc30:  0000000f 00000000 00403000 00000000
0x0069fc40:  7b8a7fe0 0069fcec 7b884bfb 7ebbe83c
0x0069fc50:  00160690 00000006 00010024 0000000f
0x0069fc60:  00000000 00000000 00000860 00000000
0x0069fc70:  00000000 00000008 000001ba 7b88481f
0x0069fc80:  00010024 00000001 0069fcb8 7eb9a6aa
fixme:ntdll:RtlNtStatusToDosErrorNoTeb no mapping for c0000119
Backtrace:

=>1 0x004012df WndProc+0x5f(hWnd=0x10024, message=0xf, wParam=0x0, lParam=0x0) [/
home/kragen/devel/hi.c:34] in hi (0x0069fc88)
  2 0x7eb9a6aa WINPROC_wrapper+0x1a in user32 (0x0069fcb8)
  3 0x7eb9ae0b in user32 (+0x9ae0b) (0x0069fcf8)
  4 0x7eb9eb5a CallWindowProcA+0x5a in user32 (0x0069fd38)
  5 0x7eb69218 DispatchMessageA+0x148 in user32 (0x0069fd78)

  6 0x004014c6 WinMain+0x15f(hInstance=0x400000, hPrevInstance=0x0, lpCmdLine=0x1
0157b8, nCmdShow=0xa) [/home/kragen/devel/hi.c:95] in hi (0x0069fe38)
  7 0x004015f7 in hi (+0x15f7) (0x0069feb8)

  8 0x004011d9 __mingw_CRTStartup+0xc9 [/home/ron/devel/debian/mingw32-runtime/mi
ngw32-runtime-3.9/build_dir/src/mingw-runtime-3.9/crt1.c:226] in hi (0x0069fee8)
  9 0x00401223 in hi (+0x1223) (0x0069ff08)
  10 0x7b86eeab in kernel32 (+0x4eeab) (0x0069ffe8)
  11 0xb7dfa7a7 wine_switch_to_stack+0x17 in libwine.so.1 (0x00000000)

0x004012df WndProc+0x5f [/home/kragen/devel/hi.c:34] in hi: movb    $0x78,0x0(%ea
x)
34      *nullptr = 'x';
Modules:
```

```
Module  Address         Debug info  Name (48 modules)
PE  400000-48e000   Stabs           hi
ELF 7b800000-7b919000   Export          kernel32<elf>
  \-PE  7b820000-7b919000   \               kernel32
ELF 7bc00000-7bc83000   Deferred        ntdll<elf>
  \-PE  7bc10000-7bc83000   \               ntdll
ELF 7bf00000-7bf03000   Deferred        <wine-loader>
ELF 7ccdf000-7cce4000   Deferred        libxfixes.so.3
ELF 7cce4000-7cced000   Deferred        libxcursor.so.1
ELF 7cced000-7cd09000   Deferred        imm32<elf>
  \-PE  7ccf0000-7cd09000   \               imm32
ELF 7cd09000-7cd0c000   Deferred        libxrandr.so.2
ELF 7cd0c000-7cd14000   Deferred        libxrender.so.1
ELF 7cd14000-7cd17000   Deferred        libxinerama.so.1
ELF 7e517000-7e73a000   Deferred        savage_dri.so
ELF 7e73a000-7e741000   Deferred        libdrm.so.2
ELF 7e741000-7e7ab000   Deferred        libgl.so.1
ELF 7e7ab000-7e7b0000   Deferred        libxdmcp.so.6
ELF 7e7b0000-7e89c000   Deferred        libx11.so.6
ELF 7e89c000-7e8aa000   Deferred        libxext.so.6
ELF 7e8aa000-7e8c2000   Deferred        libice.so.6
ELF 7e8c2000-7e8cb000   Deferred        libsm.so.6
ELF 7e8cb000-7e958000   Deferred        winex11<elf>
  \-PE  7e8e0000-7e958000   \               winex11
ELF 7ea27000-7ea47000   Deferred        libexpat.so.1
ELF 7ea47000-7ea72000   Deferred        libfontconfig.so.1
ELF 7ea72000-7eadc000   Deferred        libfreetype.so.6
ELF 7eadc000-7ec13000   Export          user32<elf>
  \-PE  7eb00000-7ec13000   \               user32
ELF 7ec13000-7ec77000   Deferred        msvcrt<elf>
  \-PE  7ec20000-7ec77000   \               msvcrt
ELF 7ec77000-7ecbd000   Deferred        advapi32<elf>
  \-PE  7ec80000-7ecbd000   \               advapi32
ELF 7ecbd000-7ecc8000   Deferred        libgcc_s.so.1
ELF 7edad000-7ee66000   Deferred        gdi32<elf>
  \-PE  7edc0000-7ee66000   \               gdi32
ELF 7ef8e000-7ef99000   Deferred        libnss_files.so.2
ELF 7ef99000-7efa3000   Deferred        libnss_nis.so.2
ELF 7efa3000-7efb9000   Deferred        libnsl.so.1
ELF 7efb9000-7efc2000   Deferred        libnss_compat.so.2
ELF 7efc2000-7efe7000   Deferred        libm.so.6
ELF 7efe9000-7efec000   Deferred        libxau.so.6
ELF 7efec000-7f000000   Deferred        libz.so.1
ELF b7c93000-b7c97000   Deferred        libdl.so.2
ELF b7c97000-b7dc8000   Deferred        libc.so.6
ELF b7dc8000-b7dda000   Deferred        libpthread.so.0
ELF b7ddb000-b7de0000   Deferred        libxxf86vm.so.1
ELF b7df3000-b7f04000   Export          libwine.so.1
ELF b7f06000-b7f1d000   Deferred        ld-linux.so.2
Threads:
process  tid      prio (all id:s are in hex)
0000000a
        0000000c    0
        0000000b    0
00000008 (D) Z:\home\kragen\devel\hi.exe
```

```
        00000009    0 <==
Wine exited with a successful status
```

It even disassembled of the instruction that it crashed at.

Downsides

There are also some downsides.

For example, I have a little SDL display hack, and since SDL runs on Win32, in theory I ought to be able to compile it for Win32 with MinGW and run it in WINE. But I don't have a Win32 version of SDL, any idea of how to compile one, or any idea of where to install it so that MinGW can find it; these problems would be easier if I were actually using Visual Studio on Microsoft Windows XP or something, since lots of other people would have solved them already. They're really, really easy for the platforms Debian focuses on:

```
kragen@thrifty:~/devel$ apt-file search SDL.h
gambas-doc: usr/share/gambas/help/ArticleSDL.html
iceape-dev: usr/include/iceape/websrvcs/nsIWSDL.h
icedove-dev: usr/include/icedove/websrvcs/nsIWSDL.h
libsdl1.2-dev: usr/include/SDL/SDL.h
libxul-dev: usr/include/xulrunner/websrvcs/nsIWSDL.h

pike7.6-reference: usr/share/doc/pike7.6-doc/html/reference/ex/predef_3A_3A/SDL.ho
otml
plib1.8.4-dev: usr/include/plib/puSDL.h
plib1.8.4-pic: usr/include/plib/puSDL.h
kragen@thrifty:~/devel$ apt-get install libsdl1.2-dev
```

And after doing that, you can compile your SDL program.

And, of course, most of the conveniences we glibc users have grown accustomed to just aren't there in Win32; it's really an environment designed for C++, not C.

The first "hello, world" above produces an executable of over 200kB, or almost 600kB with `-g`, but that's mostly debugging symbols; `strip` reduces that to just over 7kB, and `-Os` plus `strip`.

And, although GDB knows how to read `hello.exe`, it doesn't really know how to debug it:

```
kragen@thrifty:~/devel$ gdb hello.exe
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.

This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/lo
oib/tls/i686/cmov/libthread_db.so.1".

(gdb) b main
Cannot access memory at address 0x280
(gdb) r
Starting program: /home/kragen/devel/hello.exe
add-symbol-file-from-memory not supported for this target
```

```
Warning:
Cannot insert breakpoint -2.
Error accessing memory address 0x280: Input/output error.

(gdb) delete 1
No breakpoint number 1.
(gdb) delete -2
negative value
(gdb) c
Continuing.
Warning:
Cannot insert breakpoint -2.
Error accessing memory address 0x280: Input/output error.

(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kragen/devel/hello.exe
add-symbol-file-from-memory not supported for this target
Warning:
Cannot insert breakpoint -3.
Error accessing memory address 0x280: Input/output error.

(gdb) q
The program is running.  Exit anyway? (y or n) y
```

(In theory there's also `winedbg --gdb` which works OK when I `winedbg --gdb winemine.exe` but not when I try to run it on my `hello.exe`.)

Finally, although Win32 has some good points, mostly it is a tacky piece of crap, and not that much fun to program against. It's roughly as bad as Xlib.

Other Win32 Tutorials

theForger's/Brook Miles's/forgey's win32 tutorial is fantastic: http://www.winprog.org/tutorial/start.html

Another copy of it: http://www.vczx.com/tutorial/win32-tutorial/message_loop.html

RoD's fairly incompetent and sloppy tutorial, which additionally is full of patronizing content-free text: http://www.cprogramming.com/tutorial/opengl_first_windows_app00.html

Another one, on "Application Creation", verbose and full of errors (I count nine errors in the 12 paragraphs before the second heading, and it seems to continue at almost one error per paragraph thereafter; my god, it's comedically awful --- you might actually know less about Win32 after reading this than before): http://www.functionx.com/win32/Lesson01b.htm

This one isn't too terrible, but it still contains a few errors and some voodoo code; I haven't finished it: http://www.mdstud.chalmers.se/~md7amag/code/wintut/wtpart1.html

Others I haven't had the chance to read: http://www.cprogramming.com/snippets/show.php?tip=6&count=300&page=0

http://source.winehq.org/WineAPI/LoadLibraryExA.html
http://www.reactos.org/generated/doxygen/d9/d22/ldr_8c-source.html http://www.mpcforum.com/archive/index.php/t-133031.html
http://www.piotrbania.com/all/protty/p63-0x0f_NT_Shellcode_Prevention_Demystified.txt
http://members.fortunecity.com/blackfenix/dongles.html
http://www.codeproject.com/KB/winsdk/cwin32error.aspx
http://msdn.microsoft.com/en-us/library/994a1482.aspx
http://www.definitivesolutions.com/sourcecode/GenericAtl.cpp
http://www.codeguru.com/forum/archive/index.php/t-362322.html http://www.simplesamples.info/MFC/UDPSendReceive.php
http://www.winehq.org/pipermail/wine-patches/2002-May/002490.html
http://www.google.com/search?hl=en&safe=off&client=iceweasel-a&rls=org.debian:en-US:unofficial&q=+site:www.winehq.org+GetLastError+message
http://mail.python.org/pipermail/python-list/2000-June/039309.html

Conclusions

So, although we've come a long way, Debian isn't yet really the environment of choice for developing Win32 programs; but it's at least barely possible to do so. I think we're actually a lot further along for .NET stuff than for Win32 stuff.

Topics

- Programming (p. 3658) (286 notes)
- C (p. 3359) (28 notes)
- Win32 (p. 3777) (2 notes)
- Cross compiling (p. 3396) (2 notes)

Pixel stream

Kragen Javier Sitaker, 2017-06-15 (updated 2018-10-26) (4 minutes)

Suppose we wanted to design a simple unidirectional byte-stream protocol for efficiently drawing pixel graphics on a relatively unadorned framebuffer — maybe in an alternate history world where textual communications protocols evolved from gradual digitalization of TV rather than gradual multimediation of text.

For additional enjoyment, we could imagine that this happened in a world where Japan wasn't devastated by WWII, and so the raster lines run from top to bottom, then right to left, with both coordinates running in the opposite direction from the Cartesian convention.

The GIF89a extensions for animation are a somewhat promising direction: they allow you to update part of the display after a time delay. Because GIF is LZW-compressed, you get a fair bit of compression and a little bit of abstraction for a tiny amount of hardware; it takes very few bytes to redraw something you already drew.

Suppose we go a bit further in that direction. We have, say, a 16-bit color framebuffer, and normally we just stream 16-bit pixels into it in raster order. If we have a 128×128 framebuffer, which is an acceptable size, we have 16384 pixels, 32768 bytes of data; if we want to be able to repaint that completely 5 times per second, we need 163 840 bytes per second, which is 655 360 bits per second. This is a fairly easy data rate at the electronic level.

For faster updates, we can add escape sequences! In particular, let's consider four additions:

window(x, y, width, height): subsequent data updates the specified area of the framebuffer. This allows you to update part of the display more frequently, while updating other parts less frequently.

scroll(x, y): the next screen repaint starts from position (x, y) in the framebuffer, with wraparound for subsequent rasters if it runs off the edge of the framebuffer.

font(x, y, tilewidth, tileheight): subsequent data bytes are interpreted as byte indices into a 16×16 "font" located at (x, y) in the framebuffer, with tiles of tilewidth×tileheight. So, for each data byte, the current position is incremented by tileheight or, if it has exceeded the current window, set back to the top of the window and incremented by tilewidth.

nofont(): subsequent data bytes are again interpreted as raw pixel data.

You could imagine a couple of other escape sequences to support sprite compositing.

Now let's suppose that the actual framebuffer is not 128×128 but 256×256, but only 128×128 of it is normally visible; we can assign some fraction of it to ROM fonts, such as half. We can still encode all our pixel coordinates as bytes, but now we can use the scroll() sequence both for smooth pixel scrolling and for double-buffering.

This requires 128×256×2 = 65536 bytes of framebuffer. In our universe, the CGA shipped in 1981 with 640×200 bits = 128000 bits = 16000 bytes of framebuffer. So building a "terminal" to interpret such a command set only became practical two Moore's Law

iterations later, in 1984, about eight years after the US$1195 ADM-3A started supporting cursor control in 1976, and 20 years after ASCII printing terminals were already in widespread use as computer output devices. So, as an alternate history encoding, this fails.

128×128 is enough space to display 32 columns of 21 characters each in 6×4 size, but I'm not sure you can get readable katakana at that size. (Roman letters are already a bit of a stretch.) 6×6 squares, 21 columns of 21 characters, should be easy.

See also Window systems (p. 1335).

Topics

Tabulating your top event of the month efficiently using RMQ algorithms

Kragen Javier Sitaker, 2019-03-19 (8 minutes)

There are a couple of algorithms for computing a linear-time sliding RMQ ("range minimum query"): the ascending minima algorithm and the van Herk/Gil–Werman algorithm. The ascending-minima algorithm is interesting in that all of its comparisons of data (as opposed to indices) are comparisons against the most recent datum: you run all your input data through a deque, which you maintain in ascending order by popping items off its back when they slide out of the window and off its front when they are higher than the new datum.

This occurs to me as an interesting way to compute a "top event of the month" kind of list: add new events as they come in, removing any older events that are less significant and any events that, though more significant, are older than a month. At the end of the month, you simply write down the oldest event in the list, which is more significant than anything that came before it and at least as significant as anything that followed. This could work for personal achievements as well as news events; it has the characteristic that one of the events you're comparing is always the current event, which you presumably have uppermost in your consciousness. Unfortunately, both news events and personal achievements share the characteristic that it's often hard to determine their significance until after the fact.

There are some interesting tweaks to be made on the ascending-minima algorithm.

If you aren't space-limited, you could put the items on a stack rather than a deque; rather than shifting items off the left end of a deque, you can just increment an "oldest" pointer up the stack. The ultimate contents of the stack are the global minimum, the minimum of all the items that followed it, the minimum of all the items that followed *it*, and so on. These are the range minima of all possible ranges that end at the end of the entire event sequence.

Suppose we push in the normal way, but "pop" from the stack not by physically removing events but merely by updating a predecessor pointer on the newly added item. The physical sequence of the stack, then, will be the entire event sequence, augmented with predecessor pointers that enable rapid traversal of all the possible ranges ending at the end of the entire event sequence. These predecessor pointers convert the stack into a concise tree representation of the state of the stack at every point in time. This enables us to answer any range minimum query in expected logarithmic time: we start with the event at the end of the desired range, then follow its predecessor pointers until they lead us outside the desired range. The item whose predecessor pointer led us outside the desired range is, then, the range minimum.

If we furthermore update each "popped" item with the time when it was popped, then we can find in constant time the largest interval it

was the minimum of: it was the minimum from the moment following its predecessor until it was popped.

To be concrete, to compute the predecessors array, we can do the following, in Python notation:

```
js = [None] * len(xs)

for i in range(1, len(xs)):
    js[i] = i-1
    while js[i] is not None and xs[js[i]] >= xs[i]:
        js[i] = js[js[i]]
```

And to use it to find the index k of the minimal element in some nonempty [i, j):

```
k = j-1
while js[k] is not None and js[k] >= i:
    k = js[k]
```

The van Herk/Gil–Werman algorithm computes the sliding RMQ (for a single window width) of the pixels in O(N) linear time, while this takes O(N log M) time, where M is the window size. If you have a fixed number of window sizes before you start the algorithm, you could compute them in linear time (each) by walking their respective pointers up the stack as you pass over the input pixels, thus avoiding the logarithmic-time slowdown from computing them after the fact.

I'm not sure how the performance of this approach compares to Urbach and Wilkinson's 2008 chord-table algorithm (doi 10.1.1.442.4549, "Efficient 2-D Grayscale Morphological Transformations With Arbitrary Flat Structuring Elements".) Their objective is to compute sliding RMQ for a set of "chord lengths" or window sizes for each scan line; they do this by augmenting the set of chord lengths with enough powers of 2 to reach the longest chord length; they start with trivial case of window size 1, and then, to compute sliding RMQ for each larger window size R(i) as T[i, ...] from already-computed results for window size R(i–1) in T[i–1, ...] — R(i–1) is guaranteed to be at least half of R(i) due to the augmentation with the powers of 2 — they compute d = R(i) - R(i–1) and then compute each result pixel T[i, x] = T[i–1, x] ∧ T[i–1, x+d], where ∧ is the pairwise-minimum operation.

So, for example, the chord-table algorithm will compute a sliding RMQ result for the window starting at position 71 with a window size of 18 (T[R^{-1}(18), 71]) from two previously computed results with a window size of 16, we can take T[R^{-1}(16), 71] and T[R^{-1}(16), 73]. These two 16-pixel windows overlap by all but two pixels, which is harmless. In many cases the chord-table algorithm will compute more window sizes than necessary, but the computation for each window size is very regular, while the computation of the predecessor array described above is very irregular, even if a known set of window sizes is being pursued. In particular, it should be trivially possible to vectorize the chord-table algorithm, computing results for 16 or 32 scan lines in parallel.

(Urbach and Williamson's paper actually writes T_y(i, x, r), but the extra parameters r and y are, as far as I can tell, not actually useful;

the chord table for each scan line is computed entirely independently.)

Returning to the problem of computing a backward-looking greatest achievement of the month, we can of course compute the backward-looking greatest achievement of the past 1, 2, 4, 8, and 16 days, each by comparing the greatest achievement from the smaller number of days to the achievement in the previous window — for instance, comparing the greatest achievement of the last 8 days with the previously-computed greatest achievement of the previous 8 days in order to compute the greatest achievement of the last 16 days. Then for a given month we simply use two overlapping 16-day windows. This is clearly more work than the ascending-minima algorithm, requiring as it does 5 comparisons per day rather than 2. However, a sliding-window algorithm is unnecessary for this non-sliding-window application, and a simple binary-tree algorithm would require only 1 comparison per day on average.

Wikipedia has an RMQ solution using Cartesian trees achieving constant-time queries with linear space, which I think is a result due to Harel and Tarjan, which I don't understand yet. Cartesian trees are binary trees obeying the min-heap property whose inorder traversal returns the original sequence of elements. Interestingly, constructing the Cartesian tree uses almost precisely the algorithm given above! The stack is used to maintain the "rightmost spine" of the Cartesian tree under construction.

Topics

Selfish conformity

Kragen Javier Sitaker, 2016-11-15 (5 minutes)

One problem that society has always faced is that its members selfishly choose to conform to social norms instead of altruistically doing unique things. Individuals usually benefit from merely following the herd — the herd is usually headed in a safe direction, and often the nail that sticks up gets hammered down. But the herd's choice of direction is critically dependent on its members steering it away from danger, which they cannot do if they are merely following it.

This is true across many fields — mainstream investments are safe because some investors use their investment sense some of the time; mainstream morals are not completely depraved because some moral actors use their moral sense some of the time; mainstream science is mostly correct because some scientists use their critical thinking abilities some of the time; and so on.

But that's not the only benefit of nonconformity. There are many fields of human action where the value of an effort is precisely in its uniqueness. Copying an existing painting is of relatively little value; painting something new is valuable, especially if the product is successful, as few are. (The majority of novel paintings are in themselves worthless, but their very worthlessness is educational.) Repeating an already-proposed hypothesis, reimplementing an already-written program, or replicating an already-performed experiment is of some value, but not nearly as much value as proposing a new hypothesis, writing a novel program, or performing a new experiment, at least if they are successful. Advances are made by people doing things that nobody else has done yet, by originality.

Even aside from enterprises where originality is not a *sine qua non*, specialization is enormously economically beneficial, and that's true even if you leave money out of it. People are simply more productive when they work in a narrow area where they have developed very deep skills, cultivated deep relationships, and acquired or gained access to specialized capital goods that improve the productivity of that area. But such specialization unavoidably requires a more diverse set of occupations.

In the limit of job specialization, every person is doing a job that nobody else in the world is doing. Jason Evans, for example, is the maintainer of the jemalloc memory allocator, one of the highest-performance general-purpose allocators, used in many pieces of software by a few billion people. Nobody else knows as much about jemalloc as he does, despite his efforts to explain it to people. Anyone can use and modify it (it's free software) but if you need to debug a performance problem with it or want to add features to it, he can very likely achieve this with less effort than anyone else can.

Finally, there are rival resources that can serve many more people if their tastes differ. If you prefer cafés with loud music, and I prefer cafés with no music, we will tend to go to different cafés. That sort of thing prevents everyone from just going to the single best café, which would then be either overcrowded or (to my taste anyway) oversized.

Unfortunately, in every society, many people are too selfish to

manifest their uniqueness. Because they merely imitate others, the society is robbed of the originality they could have produced; its choices are poorer, sometimes fatally so, because it is guided by fewer minds; its economic productivity is vastly lower than it could be, because of underspecialization; and its rival resources are oversubscribed.

There are always social rewards for conformity and punishments for nonconformity, but these undermine the very fabric of society. Ironically, in many societies, nonconformity is confused with selfishness (perhaps because it seems arrogant) and conformity with altruism, but in fact it is impossible to be simultaneously altruistic and conform.

Uniqueness involves risk. Most new things fail. Most new ideas are wrong. Most dissenters from the scientific consensus are mistaken. For every abolitionist or pacifist, who dissent from society's moral consensus, there are five rapists of children. If you order a dish your friend knows is tasty, you won't get an inedible dinner. If you drive a truck, like millions of others, instead of writing your own memory allocator, you have a pretty good idea what the job market is like. Jason Evans worked on a bunch of unique software projects before he hit on one that got adopted by billions of people and created the unique occupation of Jemalloc Maintainer for him.

Traditional societies, since the advent of agriculture, have lived close to the Malthusian population limit. They could not tolerate much risk. Poor people even today can't tolerate much risk, so the burden of uniqueness falls on the middle and upper classes. But as a civilization we are blessed with so much abundance that we are collectively capable of absorbing much more risk than before by allowing people to be unique — and when we do, everyone benefits.

(Inspired by http://www.ribbonfarm.com/2015/02/18/a-dent-in-the-universe/.)

Topics

- Politics (p. 3639) (39 notes)
- Psychology (p. 3669) (18 notes)

A hand-powered computer?

Kragen Javier Sitaker, 2015-09-03 (updated 2017-07-19) (11 minutes)

I've written a little bit about my desire for a portable computing device powered by keystrokes on its keyboard, in particular for note-taking on journeys where I'm not driving a car. (I haven't driven a car since the last time I was outside Argentina, in 2008.) I've even done some calculations about the energy available from normal keystrokes on a keyboard, and it turns out that it's more than an order of magnitude greater than what's needed to run an e-Ink word processor. But I haven't written much about why I think this kind of machine is interesting.

Some decades back, my photographer stepfather Karl boated down the Colorado River with a friend of his. Karl took photographs; his friend wrote about the trip on a portable typewriter he carried with him. For a while I owned the typewriter. It weighed about two or three kilograms, a hefty weight for most such wilderness trips, but bearable in a boat. The typewriter didn't need batteries, since its mechanics were powered by the action of the fingers pressing its keys, and still worked when damp.

A netbook might weigh less, but lacks the other desiderata here.

Consider the case of a week-long hiking trip, with the objective of spending two or three hours each day writing, a total of 20 hours. It's straightforward to pack enough food for such a trip: if we're only worried about calories, each kilogram of oil is three days' worth of calories, even without spending down our body's fat reserves; so one or two kilograms of dry food is plenty, if you have a way to purify water. And the cost is reasonable, if not trivial. (Two kilograms of Clif bars can set you back a fair bit, and you will definitely get sick of them before the week is up. I speak from experience.)

I'm typing this on a city bus on a netbook that weighs about 1kg, about half of which is its battery. This battery might last for 4 hours of active typing use. A kilogram of batteries, then, will last you about 8 hours, far short of the 20 hours we're hoping for. 20 hours of batteries, plus the netbook would weigh 3 kilograms, as much or more than the typewriter, and cost about US$600.

Imagine, by contrast, an electronic device actually intended for use in such situations. With an e-ink screen, it's clearly visible in direct sunlight, unlike the screen of this netbook, where I struggle to read this text through the reflection of my sunlit T-shirt. It can maintain a display of a static image indefinitely without power, like a book; but it can also easily contain tens of gibibytes of maps, photographs, wildflower and edible-plant identification guides, knot-tying instructions, and so on, not to mention the entire human literary canon, from Mozi to Mark Twain, from the Bhagavad Gita to Plato. It could have a weight similar to an Amazon Swindle, perhaps 200 grams, though taking a weight hit to make the e-ink screen as robust against impacts and pressure as a paper book. It can be more waterproof than a paper book. And perhaps it can include long-distance low-bandwidth radio communications without being dependent on cellphone networks or satellites. It can include voice recording and even voice recognition.

Such a device could run from the energy of typing on it, just like the old typewriter. Or it could include a solar panel. You couldn't repair its electronics without the resources of an industrial civilization, even if it included its own blueprints, but as long as it remained working, it would be independent of that civilization. And it could remain working for a very long time indeed; you could build it without non-solid-state components except for the e-ink screen and, perhaps, parts of the keyboard.

Could you use it, additionally, for some kind of mechanical measurement and control? Historically speaking, tooling precision has been a major limitation on manufacturing; we spend a lot of effort, even today, on meeting manufacturing tolerances. A good measurement tool can dramatically speed up any number of processes. Typically mechanically exercising this kind of control requires relatively large amounts of both energy and power, but consider:

- An inkjet print head that you manually roll across a surface --- the output equivalent of the old hand scanner. If you print barcodes or OCRable text, this can serve as a means of permanent data storage and interchange.
- As you pull a wire through the device, it measures the wire and applies a brake at certain points in order to kink the wire at precise positions.
- Similarly, an electronic planing device could minutely adjust the angle of a blade as it's being rolled across a wood surface in order to minutely contour the wood surface.
- On a larger scale, if the device has some means of measuring its position (lasers? computer vision coregistration, like a flying optical mouse?) it could activate shoe-mounted effectors so that you imprint an image on the ground as you walk around.
- DLP projection or lasers, which can be pretty low-energy if it's dark, can project precise reference points onto the environment --- useful both for construction and for carpentry.
- For communication over tens or hundreds of kilometers, a heliograph can be controlled with almost arbitrarily small energy input, and can be received by a human being without needing any kind of computer.
- With some focusing optics, perhaps the heliograph approach can also be used to make controllable shapes. A heavier-weight version of this approach has been used to spectacular effect to make a solar-powered 3-D printer that works by melting desert sand into quartz glass, using a giant Fresnel lens. Perhaps an array of lightweight digitally-controlled shutters in front of the Fresnel could produce the Fourier transform of an image to project around the focal point of the lens, thus burning it into whatever material is present. (What are the best photosensitive materials to use for this kind of thing? Natural materials tend not to be photosensitive at low energies, and such materials need careful handling and darkroom processing. All materials are heat-sensitive at some point --- quartz sand is sort of a worst case there --- but maybe materials that are sensitive to very small amounts of heat are better. For example, sugar melts at a relatively low temperature, and is very good at sticking to other things; finely powdered sugar mixed with some kind of "susceptor" to absorb light could work.)

Solar

My netbook battery holds a few hundred kJ (now down to 135 kJ, over 200 kJ when it was newer). My cellphone battery claims to be 4.8 Wh, which would be 17 kJ. A 10kJ battery powering 200-pJ instructions can power 50 trillion instructions. There's a 250-watt-peak solar panel for sale right now on MercadoLibre of size 1640 mm × 992 mm, which is 154 μW/mm²; a 100 mm × 150 mm solar panel of that efficiency would be 2.3 watts peak, and take 72 minutes to charge such a 10kJ battery, assuming 100% charging efficiency. Looking at it a different way, that solar panel has a power of 12 billion 200pJ instructions per second.

One such 200-or-so-pJ-per-instruction machine is the Silicon Labs EFM8SB20F16G-A-QFN24, a 25MHz 8051, with a bit over 4 kiB of RAM, costing US$1.11. The EFM8 Sleepy Bee line is designed for special power efficiency especially in sleep mode, using 300 nA with the RTC running, and working down to 1.8 volts. 10 kJ / (300 nA × 1.8 V) ≈ 587 years. It uses 170 μA / MHz, so at the full 24.5 MHz speed of its internal oscillator, it uses about 5 mA or 9 mW, so it could run 13 days at full speed on that 10kJ battery. Reaching the 2.3 W that the above solar panel can supply for opportunistic computation would require 256 of these chips, costing US$284, and you could run them in sleep mode most or all of the time; if one of them is running at full speed, the power usage of the other 255 in sleep mode would be insignificant by comparison.

Capacitors

At this point it starts to look like a capacitor-powered machine would be a good idea, eliminating the short lifetime of batteries, which is only a few years. Individual 7.3 mm × 6.1 mm tantalum capacitors like the AVX TAJV157M025#NJ can hold 47 mJ, which would be 230 million 200pJ instructions, nearly 10 seconds at full speed, or a day in sleep. However, this capacitor is bigger and costlier than the microcontroller package itself, and it's apparently hard to get. The apparently equivalent Vishay-Sprague 597D157X9025F2T is US$8.91 in quantity 1, at 7.6 mm × 6.0 mm; the 597D227X0025M2T, with 220μF, is US$9.55. At 2.3W, charging these 69 millijoules would take 30ms.

The Kemet T491D107K016AT is a 100μF 16V MnO_2 tantalum capacitor which costs only US$1.49. That's 13 millijoules, 8.6mJ/$, which is slightly more cost-effective than the larger Vishay part, which is only 7.2mJ/$. Also, the Kemet part is enormously more popular. You could quite reasonably put six of them in parallel to slightly exceed the Vishay part's energy capacity, without resorting to the higher voltage.

What does the circuitry to charge and discharge these big capacitor arrays look like? I'm not sure.

Discharging it continuously, you might use something like the AOZ1280CI buck regulator, which PWMs anywhere from 3 to 26 volts down to whatever voltage you like, regulated by a feedback voltage divider, at 80% to 90% efficiency. This particular part is overkill — it can handle six watts of output power continuously.

Topics

Adding GPIO lines over USB with a Saleae logic analyzer

Kragen Javier Sitaker, 2017-05-10 (1 minute)

https://www.youtube.com/watch?v=dobU-bo_L1I explains that the Saleae 24MHz logic analyzer and its clones are basically a 48MHz Cypress EZ-USB chip hooked up to the USB bus and some I/O pins, but *with the program loaded each time you plug it in*. The 8051 doesn't have a Flash or MRAM for its program. So you can use it, for example, in the way they have designed it, to stream parallel data over USB at its maximum sampling rate.

Sigrok comes with an open-source firmware for the Cypress FX2 chips called fx2lafw: FX2 logic analyzer firmware.

But you could also download a different program into the peripheral when you plug it in, a program to make it do anything else at all, as long as it's plugged in. You could use it to read sensors, control GPIO pins, whatever. And these chips have a *lot* of pins.

(He does in fact suggest using the Cypress breakout board for exactly this.)

Topics

• Electronics (p. 3430) (138 notes)

Chintzy depth of field

Kragen Javier Sitaker, 2016-10-27 (1 minute)

How do you render images with realistic depth-of-field defocus?

You can do a kind of reasonable out-of-focus blur with a Gaussian blur, although the actual OTF of an out-of-focus camera is more like a circular box filter. And you can do a pretty good approximation of a Gaussian blur with three (ordinary, non-circular) box filters. But the diameter of the blur changes according to how far things are from the focal plane.

It occurs to me that maybe you can adjust the width of those box filters dynamically as you move over the image, and that maybe this will give you a reasonable-looking approximation of depth-of-field blur for many images, though not all. I'm thinking that you could have a "weight" factor for each pixel that is highest at the focal point and decreases as you move further from the focal point, and you maintain a constant "weight" within the moving-average sliding window as you slide it, sliding one or the other edge faster as necessary to keep the weight inside the window constant. This way, the window is very narrow when it's at the focal plane, and very wide when it's far away.

(This still results in computing far too many pixels for the out-of-focus stuff, and doesn't help with blurring of reflected images.)

Topics

Instant hypertext

Kragen Javier Sitaker, 2013-05-17 (updated 2013-05-20) (14 minutes)

(I think I published this on kragen-tol at some point.)

I'm frustrated with the slowness of web browsers at navigating Wikipedia. Navigating Wikipedia is one of the most important tasks you can do on a computer; it provides access to a reasonable summary of all human knowledge.

Ideally, there should be *no perceptible delay* between selecting a link and seeing the resulting page. The `links` browser can more or less realize this for pages it has in its cache, although only up to a certain size and without displaying images. More full-featured browsers, unfortunately, are not capable of such a feat. To some extent this is a consequence of the very powerful features of CSS and JavaScript which do not admit general efficient implementations, but in the case of Wikipedia, the use of CSS and JS is quite stereotyped and not at all essential to the main content. It's entirely feasible to provide no perceptible delay for Wikipedia pages.

My current setup

I'm running `kiwix-serve` on an English Wikipedia snapshot and displaying it using links:

```
./kiwix/bin/kiwix-serve --port=8000 ~/Downloads/wikipedia_en_all_nopic_01_2012.zi
om &
links http://localhost:8000/wikipedia_en_all_nopic_01_2012/
```

This snapshot, which can be torrented, occupies 9.7 gibibytes of disk space and includes all the text and equations but none of the pictures from the English Wikipedia, not even the most popular pictures without copyright encumberment. (I think that an additional gibibyte or two of images could be extremely valuable.) It's deeply unfortunate that the version of `kiwix-serve` I'm using is not fully functional without JS in the browser, since the main reason for using `kiwix-serve` rather than `kiwix` is to be able to browse using an unusual browser.

Back to the ideal: "no perceptible delay" could be operationalized as 200 milliseconds (a traditional number from my memory of HCI research) or 70 milliseconds (the length of a typical keypress: you want to display the result page before the user finishes pressing the key) or 17 milliseconds (the screen refresh time of nearly all current computer monitors).

70 milliseconds should be achievable

Meeting any of these goals (17ms, 70ms, or 200ms) would necessarily require a local replica, if not a replica necessarily on my own machine; `ping en.wikipedia.org` from my location in Argentina reports 300ms average latency, so even a single round trip would blow it. In fact, I haven't found anyplace in Argentina with less than 250ms latency to anyplace in the US.

I argue that a 70-millisecond response time is achievable, with no

network server, using a reasonably powerful machine (a netbook from a couple of years back, or an Android phone with a large SD card). The minimal amount of work involved is displaying a screenful of data (around 0.6 megapixels, or 1.8 megs uncompressed) following a single random hard disk seek (a 10ms cost which goes away if you're using an SD card) after determining which of perhaps a few dozen links on the screen the user clicked on. That is, transferring three megabytes of data from a random location on the disk to the framebuffer. This is almost the same task that any movie player software performs in 17 milliseconds, except that it decompresses the frame from data in memory instead of loading it from a random location on the disk.

A typical modern laptop hard disk transfers some 20 megabytes per second, or 20 kilobytes per millisecond. That means that in the 70 – 10 – 17 = 43 remaining milliseconds, it can transfer 860 kilobytes. So we could meet our 70-millisecond performance goal if we could compress the screen image to display by only a factor of two. However, we wouldn't be able to fit much of Wikipedia onto a typical netbook hard disk in that representation; so compression is necessary for reasonable coverage, as well as saving us much of those milliseconds to use for other things!

A lot of work is needed.

According to Chromium, `kiwix-serve` can serve up a typical page in under 400ms on my netbook, although it occasionally takes almost 500ms. I don't know how much of that is due to the LZMA2 compression used by the `.zim` file format, and how much is due to `kiwix` being suboptimal.

Lossless image compression

I made a 1024×600 screenshot of `links` displaying a screenful of a Wikipedia page at 113×33 characters (3729 characters). This screenshot compressed as PNG takes 80 kilobytes, or 72 kilobytes after `optipng`; `pngtopnm` decompressing it to raw data takes 72 user milliseconds, although given that the clock time is around 250ms, I don't really trust that. (The gzipped PNM decompresses in 20ms.) A GIF version produced by ImageMagick is 63 kilobytes. This is to say, PNG or GIF compression reduces it to such a size that it needs about 3ms of disk transfer time to load.

Here is the screenful, stripped of its highlighting and hypertext links, which should not comprise a significant fraction of its size in bytes:

```
                                                                    Ba
nking in Switzerland (p2 of 17)

     * 8 International competition

     * 9 See also

     * 10 References
```

Overview

Switzerland is a prosperous nation with a per capita gross domestic product higher than that of most western European nations. In addition, the value of the Swiss franc (CHF) has been relatively stable compared with that of other currencies.[3] In 2009, the financial sector comprised 11.6% of Switzerland's GDP and employed approximately 195,000 people (136,000 of whom work in the banking sector); this represents about 5.6% of the total Swiss workforce. Furthermore, Swiss banks employ an estimated 103,000 people abroad.[4]

Swiss neutrality and national sovereignty, long recognized by foreign nations, have fostered a stable environment in which the banking sector was able to develop and thrive. Switzerland has maintained neutrality through both World Wars, is not a member of the European Union, and was not even a member of the United Nations until 2002.[5][6]

Currently an estimated one-third of all funds held outside the country of origin (sometimes called "offshore" funds) are kept in Switzerland. In 2001 Swiss banks managed US$ 2.6 trillion. The following year they handled $400 billion USD less which has been attributed to both a bear market and stricter regulations on Swiss banking.[7] By 2007 this figure has risen to roughly 6.7 trillion Swiss francs (US$6.4 trillion).

The Bank of International Settlements, an organization that facilitates cooperation among the world's central banks, is headquartered in the city of Basel. Founded in 1930, the BIS chose to locate in Switzerland because of the country's neutrality, which was important to an organization founded by countries that had been on both sides of World War I.[8]

```
Foreign banks operating in Switzerland manage 870 billion Swiss francs worth o
of assets (as of May 2006).[9]


http://localhost:8000/wikipedia_en_all_nopic_01_2012/A/Banking in Switzerland.htm
l#International_competition
```

This character data compresses to 1158 bytes with `gzip -9`, 3.22 times smaller, 1202 bytes with `bzip2 -9`, and 1446 bytes with `compress`.

None of these programs take a significant amount of time to decompress the data, but my measurement noise is around 4ms, so they might be taking as much as 4ms.

Conclusion: compressed images are very heavy, and to be avoided if possible.

Wikipedia size

As I mentioned, the English Wikipedia snapshot I'm using is 9.7 gibibytes. `kiwix-serve` reports that it contains "1,766,695MB (3,874,564 articles, 262,623 medias)", which is to say, 1.8 terabytes in 3.9 million articles, or 460 kilobytes per article. I seem to recall that the text of a typical article is some tens of kilobytes, and almost none are over 100 kilobytes, so I assume that the bulk of that is the "medias".

Five random Wikipedia pages (not in the snapshot) are 4, 6, 7, 7, and 10 such screenfuls in links, so I'm going to assume that 7 is a reasonable median and 20 is a reasonable guess at mean number of screenfuls. That means we have about 80 million screenfuls of text, or 300 gigabytes of text, uncompressed; perhaps it would be 93 gigabytes compressed. This seems surprisingly large, since the original Wikipedia source data is only about 9 gigabytes in bz2-compressed XML. Nevertheless, it's small enough to be feasible to store textual snapshots of this entire Wikipedia on a normal netbook, and you could retrieve any one of them with a single seek if you identified them by number.

(If you printed out these 80 million 113×33 screenfuls in a square, they would be about 800 thousand characters wide and 400 thousand tall; in a normal ten-point font, this would be about 500 feet square.)

Since lossless image compression produced files that were some 40 times larger than text compression, we probably don't have room on the disk for precomputed images of the entire Wikipedia, pre-rendered. However, it's probably perfectly adequate to layout and render the text on the fly, within limits.

Image selection

That doesn't help with things that aren't text, and in particular, math articles suffer badly from lack of diagrams. http://en.wikipedia.org/wiki/Cantor_function is rather hard to understand without the graphs, the first of which is an 8-kilobyte SVG drawn with Adobe Illustrator which gzips down to 988 bytes; the third is a 9-kilobyte PNG. If you included a single gigabyte of images, you could include a million images of the complexity of this SVG; you could include one such image per article in four gigabytes.

Presumably you want to optimize some kind of cost-benefit ratio,

perhaps using page view counts or lists of featured articles or vital articles to estimate the benefit, and the file size (perhaps with an estimate of benefit due to compression) to estimate the cost. For example, Georg Cantor is a "vital article" that is read some 700 times a day, and it uses a diagram of bijection which is 1.8 kilobytes gzipped and is also used on four other pages; perhaps the 27-kilobyte portrait of young Cantor used on no other pages does not contribute 15 times as much to your replica of Wikipedia. If a picture is one of the 3196 featured pictures it's probably better quality. http://commons.wikimedia.org/wiki/Commons:Picture_of_the_Ye
oar http://commons.wikimedia.org/wiki/Commons:Valued_images http://commons.wikimedia.org/wiki/Commons:Quality_images http://commons.wikimedia.org/wiki/Commons:Featured_pictures

Wikimedia Commons says it has 15 million files, which add up to 22.5 terabytes. This is substantially larger than any of the textual Wikipedias, by about two thousand times. The great bulk of this, about 19.6 terabytes, is JPEGs, followed by 880 gigabytes of TIFFs, 600 gigabytes of PNGs, 520 gigabytes of Ogg video, 300 gigabytes of DjVu, and so on.

These files range over a very wide range of sizes; even consulting statistics by MIME type shows that the 2597 MIDI files average 5.2 kilobytes while WebM video files average 27 megabytes, four orders of magnitude greater. That is, all the MIDI files put together are smaller than a single average WebM video, and of course within each category, there are orders of magnitude of variation.

The Commons metadata is available for bulk download, currently totaling about 2.4 gigabytes, and could be used to measure the file size distribution more precisely.

Most of the JPEGs are very large indeed, and a much smaller version would be more than adequate for initial page presentation; some random images I selected using http://commons.wikimedia.org/wiki/Special:Random/File were 0.3 megapixels, 0.3 megapixels, 3.1 megapixels, 5.9 megapixels, and a 22-kilobyte SVG. Most of the time, however, images are presented in the page at a much smaller size. For example, I clicked the "random" link above until I got a big image: http://commons.wikimedia.org/wiki/File:Ebensee1.jpg, which is 7.7 megapixels, but where it's used on http://de.wikipedia.org/wiki/Bezirk_Gmunden, it's scaled down to 150×99, or 0.015 megapixels, 500 times smaller, making it 7.5 kilobytes instead of 4.9 megabytes.

(en.wikipedia.org has its own set of uploaded files, totaling 800 thousand to Commons's 15 million. I'm not clear if Commons images used on en are included in that total.)

A quick sample of five random images finds one used on the Hebrew Wikipedia and none used in English Wikipedia. Four more finds one on nl.wikipedia; eight more finally finds one image used on en.wikipedia as well as others, one used on es.wikipedia, one used on de.wikipedia, and one more used on en.wikipedia. Naïvely, two out of 17 in the "sample" should mean that on the order of 15% of the files in Commons are used on en.wikipedia. Both of these are used at a reduced size in the articles, and have an en.wikipedia File: namespace page.

Topics

Performance properties of sets of bitwise operations

Kragen Javier Sitaker, 2018-11-06 (updated 2018-11-07) (16 minutes)

For Bootstrapping instruction set (p. 459), I did some studies of different possible sets of bitwise operations using http://canonical.org/~kragen/sw/dev3/abjsearch.py. Of the Boolean-complete sets, abjunction alone and NAND alone had nearly the worst worst case; computing `x ^ y` with them requires 5 operations, not counting the constant all-ones -1 (which it needs because it's falsehood-preserving): `-1 &^ ((-1 &^ (x &^ y)) &^ (y &^ x))` for abjunction, `(y &̄ (x &̄ x)) &̄ (x &̄ (x &̄ y))` for NAND.

Considering the case with only abjunction, which is to say, set subtraction or ANDNOT, which Golang spells `&^`.

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = 0 (truth table 0000, cost 1) = 0
r[ 3] = -1 (truth table 1111, cost 1) = -1
r[ 4] = r[0] &^ r[1] (truth table 0010, cost 1) = x &^ y
r[ 5] = r[1] &^ r[0] (truth table 0100, cost 1) = y &^ x
r[ 6] = r[3] &^ r[0] (truth table 1100, cost 2) = -1 &^ x
r[ 7] = r[3] &^ r[1] (truth table 1010, cost 2) = -1 &^ y
r[ 8] = r[0] &^ r[4] (truth table 0001, cost 2) = x &^ (x &^ y)
r[ 9] = r[3] &^ r[4] (truth table 1101, cost 3) = -1 &^ (x &^ y)
r[10] = r[3] &^ r[5] (truth table 1011, cost 3) = -1 &^ (y &^ x)
r[11] = r[6] &^ r[1] (truth table 1000, cost 3) = (-1 &^ x) &^ y
r[12] = r[3] &^ r[8] (truth table 1110, cost 4) = -1 &^ (x &^ (x &^ y))
r[13] = r[3] &^ r[11] (truth table 0111, cost 4) = -1 &^ ((-1 &^ x) &^ y)
r[14] = r[9] &^ r[5] (truth table 1001, cost 5) = (-1 &^ (x &^ y)) &^ (y &^ x)

r[15] = r[3] &^ r[14] (truth table 0110, cost 6) = -1 &^ ((-1 &^ (x &^ y)) &^ (y &^ x))
```

With only binary NAND, the situation is almost as bad, but without needing -1, and with more reuse:

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = r[0] & r[0] (truth table 1100, cost 1) = x & x
r[ 3] = r[0] & r[1] (truth table 1110, cost 1) = x & y
r[ 4] = r[1] & r[1] (truth table 1010, cost 1) = y & y
r[ 5] = r[0] & r[2] (truth table 1111, cost 2) = x & (x & x)
r[ 6] = r[0] & r[3] (truth table 1101, cost 2) = x & (x & y)
r[ 7] = r[1] & r[2] (truth table 1011, cost 2) = y & (x & x)
r[ 8] = r[2] & r[4] (truth table 0111, cost 3) = (x & x) & (y & y)
r[ 9] = r[3] & r[3] (truth table 0001, cost 2) = a & a where a = x & y
r[10] = r[3] & r[8] (truth table 1001, cost 5) = (x & y) & ((x & x) & (y & y))

r[11] = r[5] & r[5] (truth table 0000, cost 3) = a & a where a = x & b and b = x & x
```

```
r[12] = r[6] & r[6] (truth table 0010, cost 3) = a & a where a = x & b and b = x &
& y

r[13] = r[7] & r[7] (truth table 0100, cost 3) = a & a where a = y & b and b = x &
& x

r[14] = r[8] & r[8] (truth table 1000, cost 4) = a & a where a = c & b and b = y &
& y and c = x & x
r[15] = r[6] & r[7] (truth table 0110, cost 5) = (x & (x & y)) & (y & (x & x))
```

If we have 2-, 3-, and 4-input NANDs, the situation is pretty much exactly the same as with just 2-input NANDs.

Actually, though, separate `&` and `~` operations (or equivalently `|` and `~`) are even worse:

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = ~r[0] (truth table 1100, cost 1) = ~x
r[ 3] = ~r[1] (truth table 1010, cost 1) = ~y
r[ 4] = r[0] & r[1] (truth table 0001, cost 1) = x & y
r[ 5] = r[0] & r[2] (truth table 0000, cost 2) = x & (~x)
r[ 6] = r[0] & r[3] (truth table 0010, cost 2) = x & (~y)
r[ 7] = r[1] & r[2] (truth table 0100, cost 2) = y & (~x)
r[ 8] = r[2] & r[3] (truth table 1000, cost 3) = (~x) & (~y)
r[ 9] = ~r[4] (truth table 1110, cost 2) = ~(x & y)
r[10] = ~r[5] (truth table 1111, cost 3) = ~(x & (~x))
r[11] = ~r[6] (truth table 1101, cost 3) = ~(x & (~y))
r[12] = ~r[7] (truth table 1011, cost 3) = ~(y & (~x))
r[13] = ~r[8] (truth table 0111, cost 4) = ~((~x) & (~y))
r[14] = r[9] & r[13] (truth table 0110, cost 7) = (~(x & y)) & (~((~x) & (~y)))
r[15] = r[11] & r[12] (truth table 1001, cost 7) = (~(x & (~y))) & (~(y & (~x)))
```

Darius Bacon suggests considering the bitwise ternary operator `(x & y | ~x & z)` as a primitive operation.

The bitwise ternary operator is almost as universal as abjunction, which is to say that, since it's falsehood-preserving, it requires access to a constant -1 to be universal, and since it's also truth-preserving, it also requires a constant 0; but it's somewhat more efficient. It computes `x ^ y` in two operations plus a constant 0 as `x ? (y ? 0 : x) : y`, and reaching the negating operations NAND and NOR is barely more difficult. NAND is `x ? (y ? 0 : x) : -1` and NOR is `x ? 0 : y ? x : -1`.

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = 0 (truth table 0000, cost 1) = 0
r[ 3] = -1 (truth table 1111, cost 1) = -1
r[ 4] = r[0] ? r[0] : r[1] (truth table 0111, cost 1) = x ? x : y
r[ 5] = r[0] ? r[1] : r[0] (truth table 0001, cost 1) = x ? y : x
r[ 6] = r[0] ? r[1] : r[3] (truth table 1101, cost 2) = x ? y : -1
r[ 7] = r[0] ? r[2] : r[1] (truth table 0100, cost 2) = x ? 0 : y
r[ 8] = r[0] ? r[2] : r[3] (truth table 1100, cost 3) = x ? 0 : -1
r[ 9] = r[1] ? r[0] : r[3] (truth table 1011, cost 2) = y ? x : -1
r[10] = r[1] ? r[2] : r[0] (truth table 0010, cost 2) = y ? 0 : x
r[11] = r[1] ? r[2] : r[3] (truth table 1010, cost 3) = y ? 0 : -1
```

```
r[12] = r[0] ? r[1] : r[9] (truth table 1001, cost 3) = x ? y : (y ? x : -1)
r[13] = r[0] ? r[2] : r[9] (truth table 1000, cost 4) = x ? 0 : (y ? x : -1)
r[14] = r[0] ? r[10] : r[1] (truth table 0110, cost 3) = x ? (y ? 0 : x) : y
r[15] = r[0] ? r[10] : r[3] (truth table 1110, cost 4) = x ? (y ? 0 : x) : -1
```

The negated version of the bitwise ternary operator is even more expressive in this sense, as it has no need for constants; it reaches NAND, NOR, constant 0, constant -1, and the negation of either input in a single application, and needs only a single additional application to reach the rest of the binary boolean functions, including AND, OR, XOR, and XNOR. OR is `~(x ? a : a) where a = ~(x ? x : y)`, AND is `~(x ? a : a) where a = ~(x ? y : x)`, XOR is `~(x ? y : (~(x ? x : y)))`, and XNOR is `~(x ? (~(x ? y : x)) : y)`.

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = ~(r[0] ? r[0] : r[0]) (truth table 1100, cost 1) = ~(x ? x : x)
r[ 3] = ~(r[0] ? r[0] : r[1]) (truth table 1000, cost 1) = ~(x ? x : y)
r[ 4] = ~(r[0] ? r[1] : r[0]) (truth table 1110, cost 1) = ~(x ? y : x)
r[ 5] = ~(r[0] ? r[1] : r[1]) (truth table 1010, cost 1) = ~(x ? y : y)

r[ 6] = ~(r[0] ? r[0] : r[2]) (truth table 0000, cost 2) = ~(x ? x : (~(x ? x : xo
o)))

r[ 7] = ~(r[0] ? r[0] : r[3]) (truth table 0100, cost 2) = ~(x ? x : (~(x ? x : yo
o)))

r[ 8] = ~(r[0] ? r[1] : r[2]) (truth table 0010, cost 2) = ~(x ? y : (~(x ? x : xo
o)))

r[ 9] = ~(r[0] ? r[1] : r[3]) (truth table 0110, cost 2) = ~(x ? y : (~(x ? x : yo
o)))

r[10] = ~(r[0] ? r[2] : r[0]) (truth table 1111, cost 2) = ~(x ? (~(x ? x : x)) :o
o x)

r[11] = ~(r[0] ? r[2] : r[1]) (truth table 1011, cost 2) = ~(x ? (~(x ? x : x)) :o
o y)

r[12] = ~(r[0] ? r[3] : r[3]) (truth table 0111, cost 2) = ~(x ? a : a) where a =o
o ~(x ? x : y)

r[13] = ~(r[0] ? r[4] : r[0]) (truth table 1101, cost 2) = ~(x ? (~(x ? y : x)) :o
o x)

r[14] = ~(r[0] ? r[4] : r[1]) (truth table 1001, cost 2) = ~(x ? (~(x ? y : x)) :o
o y)

r[15] = ~(r[0] ? r[4] : r[4]) (truth table 0001, cost 2) = ~(x ? a : a) where a =o
o ~(x ? y : x)
```

Similarly, the closely-related AOI and-or-invert function of four bits, sometimes realized as a single gate, can also reach all of the binary boolean functions in only two applications.

```
 r[ 0] = x (truth table 0011, cost 0) = x
 r[ 1] = y (truth table 0101, cost 0) = y

 r[ 2] = ~(r[0] & r[0] | r[0] & r[0]) (truth table 1100, cost 1) = ~(x & x | x & x)

 r[ 3] = ~(r[0] & r[0] | r[1] & r[1]) (truth table 1000, cost 1) = ~(x & x | y & y)

 r[ 4] = ~(r[0] & r[1] | r[0] & r[1]) (truth table 1110, cost 1) = ~(x & y | x & y)

 r[ 5] = ~(r[0] & r[1] | r[1] & r[1]) (truth table 1010, cost 1) = ~(x & y | y & y)

 r[ 6] = ~(r[0] & r[0] | r[2] & r[2]) (truth table 0000, cost 2) = ~(x & x | a & a) where a = ~(x & x | x & x)

 r[ 7] = ~(r[0] & r[0] | r[3] & r[3]) (truth table 0100, cost 2) = ~(x & x | a & a) where a = ~(x & x | y & y)

 r[ 8] = ~(r[0] & r[1] | r[2] & r[2]) (truth table 0010, cost 2) = ~(x & y | a & a) where a = ~(x & x | x & x)

 r[ 9] = ~(r[0] & r[1] | r[3] & r[3]) (truth table 0110, cost 2) = ~(x & y | a & a) where a = ~(x & x | y & y)

 r[10] = ~(r[0] & r[2] | r[0] & r[2]) (truth table 1111, cost 2) = ~(x & a | x & a) where a = ~(x & x | x & x)

 r[11] = ~(r[0] & r[4] | r[0] & r[4]) (truth table 1101, cost 2) = ~(x & a | x & a) where a = ~(x & y | x & y)

 r[12] = ~(r[0] & r[2] | r[1] & r[2]) (truth table 1011, cost 2) = ~(x & a | y & a) where a = ~(x & x | x & x)

 r[13] = ~(r[0] & r[3] | r[3] & r[3]) (truth table 0111, cost 2) = ~(x & a | a & a) where a = ~(x & x | y & y)

 r[14] = ~(r[0] & r[4] | r[4] & r[4]) (truth table 0001, cost 2) = ~(x & a | a & a) where a = ~(x & y | x & y)

 r[15] = ~(r[0] & r[4] | r[1] & r[4]) (truth table 1001, cost 2) = ~(x & a | y & a) where a = ~(x & y | x & y)
```

These exotic three- and four-input Boolean functions, despite their attractive formal properties, are probably not ideal for a virtual machine design in practice; they are generally not provided by CPUs, so in a naïve implementation, they probably need to be emulated by a few instructions, possibly increasing register pressure.

If we limit ourselves to the full set provided by any CPU I'm familiar with — `&`, `|`, `~`, `^`, and `&^` — we never need more than two ops to reach any binary Boolean function:

```
r[ 0] = x (truth table 0011, cost 0) = x
```

```
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = r[0] & r[1] (truth table 0001, cost 1) = x & y
r[ 3] = r[0] | r[1] (truth table 0111, cost 1) = x | y
r[ 4] = ~r[0] (truth table 1100, cost 1) = ~x
r[ 5] = ~r[1] (truth table 1010, cost 1) = ~y
r[ 6] = ~r[2] (truth table 1110, cost 2) = ~(x & y)
r[ 7] = ~r[3] (truth table 1000, cost 2) = ~(x | y)
r[ 8] = r[0] ^ r[0] (truth table 0000, cost 1) = x ^ x
r[ 9] = r[0] ^ r[1] (truth table 0110, cost 1) = x ^ y
r[10] = r[0] &^ r[1] (truth table 0010, cost 1) = x &^ y
r[11] = r[1] &^ r[0] (truth table 0100, cost 1) = y &^ x
r[12] = r[0] ^ r[4] (truth table 1111, cost 2) = x ^ (~x)
r[13] = r[0] ^ r[5] (truth table 1001, cost 2) = x ^ (~y)
r[14] = r[1] | r[4] (truth table 1101, cost 2) = y | (~x)
r[15] = r[0] | r[5] (truth table 1011, cost 2) = x | (~y)
```

Removing `&^` does not worsen the situation in that sense, although `x &^ y` and `y &^ x` become two ops instead of one:

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = r[0] & r[1] (truth table 0001, cost 1) = x & y
r[ 3] = r[0] | r[1] (truth table 0111, cost 1) = x | y
r[ 4] = ~r[0] (truth table 1100, cost 1) = ~x
r[ 5] = ~r[1] (truth table 1010, cost 1) = ~y
r[ 6] = ~r[2] (truth table 1110, cost 2) = ~(x & y)
r[ 7] = ~r[3] (truth table 1000, cost 2) = ~(x | y)
r[ 8] = r[0] ^ r[0] (truth table 0000, cost 1) = x ^ x
r[ 9] = r[0] ^ r[1] (truth table 0110, cost 1) = x ^ y
r[10] = r[0] ^ r[2] (truth table 0010, cost 2) = x ^ (x & y)
r[11] = r[0] ^ r[3] (truth table 0100, cost 2) = x ^ (x | y)
r[12] = r[0] ^ r[4] (truth table 1111, cost 2) = x ^ (~x)
r[13] = r[0] ^ r[5] (truth table 1001, cost 2) = x ^ (~y)
r[14] = r[1] | r[4] (truth table 1101, cost 2) = y | (~x)
r[15] = r[0] | r[5] (truth table 1011, cost 2) = x | (~y)
```

If we leave `&^` in, it's also possible to remove `|` without the worst case getting any worse, and indeed the only impact is that we need two ops to compute `|` as `x ^ (y &^ x)`:

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = r[0] & r[1] (truth table 0001, cost 1) = x & y
r[ 3] = ~r[0] (truth table 1100, cost 1) = ~x
r[ 4] = ~r[1] (truth table 1010, cost 1) = ~y
r[ 5] = ~r[2] (truth table 1110, cost 2) = ~(x & y)
r[ 6] = r[0] ^ r[0] (truth table 0000, cost 1) = x ^ x
r[ 7] = r[0] ^ r[1] (truth table 0110, cost 1) = x ^ y
r[ 8] = r[0] &^ r[1] (truth table 0010, cost 1) = x &^ y
r[ 9] = r[0] ^ r[3] (truth table 1111, cost 2) = x ^ (~x)
r[10] = r[0] ^ r[4] (truth table 1001, cost 2) = x ^ (~y)
r[11] = ~r[8] (truth table 1101, cost 2) = ~(x &^ y)
r[12] = r[1] &^ r[0] (truth table 0100, cost 1) = y &^ x
r[13] = ~r[12] (truth table 1011, cost 2) = ~(y &^ x)
r[14] = r[3] &^ r[1] (truth table 1000, cost 2) = (~x) &^ y
```

```
r[15] = r[0] ^ r[12] (truth table 0111, cost 2) = x ^ (y &^ x)
```

This is not true of &, ~, and ^; removing any of them causes some binary Boolean functions to require three ops. If we remove both | and &^, then we also get some Boolean operations requiring three ops:

```
r[ 0] = x (truth table 0011, cost 0) = x
r[ 1] = y (truth table 0101, cost 0) = y
r[ 2] = r[0] & r[1] (truth table 0001, cost 1) = x & y
r[ 3] = ~r[0] (truth table 1100, cost 1) = ~x
r[ 4] = ~r[1] (truth table 1010, cost 1) = ~y
r[ 5] = ~r[2] (truth table 1110, cost 2) = ~(x & y)
r[ 6] = r[0] ^ r[0] (truth table 0000, cost 1) = x ^ x
r[ 7] = r[0] ^ r[1] (truth table 0110, cost 1) = x ^ y
r[ 8] = r[0] ^ r[2] (truth table 0010, cost 2) = x ^ (x & y)
r[ 9] = r[0] ^ r[3] (truth table 1111, cost 2) = x ^ (~x)
r[10] = r[0] ^ r[4] (truth table 1001, cost 2) = x ^ (~y)
r[11] = r[0] ^ r[5] (truth table 1101, cost 3) = x ^ (~(x & y))
r[12] = r[1] ^ r[2] (truth table 0100, cost 2) = y ^ (x & y)
r[13] = r[1] ^ r[5] (truth table 1011, cost 3) = y ^ (~(x & y))
r[14] = r[3] & r[4] (truth table 1000, cost 3) = (~x) & (~y)
r[15] = r[0] ^ r[12] (truth table 0111, cost 3) = x ^ (y ^ (x & y))
```

So reasonable minimal sets for implementation on existing hardware include &, ~, ^, and &^; and &, ~, ^, and |. The former is more widely supported, and it's hard to argue that the latter is even more convenient.

Topics

Current-source grid

Kragen Javier Sitaker, 2018-04-30 (updated 2018-07-05) (29 minutes)

Your wall socket is 50Hz or 60Hz, 120 or 240 volts rms ac, but most importantly it is a voltage source. That is, the voltage remains constant at, say, 235 volts, regardless of whether you're drawing 0 amps or 20 amps from it. Actually, this is a little bit of a lie; the voltage does reduce somewhat as you draw current from it. Maybe the resistance of the wire in the wall is 1Ω, so if you're drawing 20 amps, the voltage falls to 215 volts. And as your house draws more current, the voltage on the whole circuit to your house from the transformer falls a bit, though probably not as much as the voltage drop from the wires in your wall. But the voltage is *relatively* constant.

This configuration gives rise to a number of phenomena we're familiar with, not all agreeable:

• All of our unswitched outlets are wired in parallel. Switches are wired in series with the devices they control.
• If we touch the prongs of an electric plug as we plug it in or unplug it, we get a nasty shock — potentially a deadly one.
• Negative-resistance devices like fluorescent tubes and neon lights — as well as fixed-voltage devices like LEDs — require auxiliary electronics to operate at all without exploding.
• If an electrical appliance has its insulation fail inside, even if it's a low-power appliance like a VCR or LED lamp, its carcass can become electrified — resulting in either a circuit breaker tripping (if the ground wire is properly wired) or a deadly shock hazard.
• We can step voltage up for transmission and distribution using transformers, but the transformers are heavy and bulky, made from laminated electrical steel.
• Even an electrical outlet that is providing no energy to anything is potentially hazardous if a toddler sticks hairpins into it.
• 50Hz or 60Hz interference is picked up by everything; under some circumstances, many harmonics also appear. These are audible and frequently cause trouble with audio electronics.
• As little as 100 mA of this current can be fatal, and it's easy for human skin to have the <2.4kΩ resistance necessary for this to happen.
• The insulation thickness and separation between the conductors is sized for the voltage that is always present, with a healthy safety factor. But the thickness of the copper conductor itself is sized for a worst-case current that may never be present, again with a healthy safety factor.
• As at least one teenager has learned, throwing eggs into an electrical substation can cause a dangerous explosion.
• The temperature a heating element reaches depends on its length. If we reduce a heating element to half its length, we double the power it dissipates, quadrupling the power density. In addition to making calculations somewhat trickier, this means that sticking a knife in the toaster will likely melt the heating element, ruining the toaster.

• Every standard electrical cable carries this deadly voltage difference, and if cut, can easily melt the cutting tool and cause an electrical fire.
• Powering a low-voltage device (like any transistor-based electronics) requires either one of those heavy transformers mentioned above or a much more complex (and noisy and potentially unreliable) switching power supply.
• Fixed-speed single-direction electrical motors (up to 50Hz or 60Hz, i.e. 3000 or 3600 rpm) are very simple and extremely reliable, but on single-phase power, require either an inefficient shaded-pole arrangement or a large and failure-prone starter capacitor to guarantee a particular direction of rotation.
• Many switching power supplies have very large inrush currents when first plugged in, which frequently results in flying sparks from the prongs of the electrical plug.
• We protect our houses and appliances by wiring circuit breakers or fuses in series with them. When the current exceeds the rating of the safety device, it disconnects, creating an open-circuit condition and its attendant inductive voltage spikes, but cutting the current to zero. However, this generally only cuts the hot side of the supply; the problematic appliance remains connected to the neutral side, which can still carry a hazardous voltage.
• Short circuits are a bad thing which can result in electrical fires and even plasma explosions. Open circuits are innocent and totally normal, occurring whenever we unplug something or turn it off.
• If you start to receive a significant electrical shock, you lose control of your muscles, which may cause you to firmly grasp the source of the shock, increasing the danger.
• It's basically impossible to shield the extremely-low-frequency emissions and prevent them from escaping a building or penetrating a sensitive measuring instrument.

None of these are inherent to electricity in general. If we used an RF current source rather than a extremely-low-frequency voltage source, all of the above items change.

What is a current source?

First, what is a voltage source?

A voltage source is an energy source with a constant voltage across the load, regardless of the current. Its output impedance is zero. The voltage of your wall socket is a close approximation of a voltage source; its impedance is on the order of 1Ω. It has a limit to how much current it can supply — even leaving aside the fuses and circuit breakers, to draw 240 amps from your 1Ω 240V source, you would need a short-circuit load, which means that none of the power actually reaches the load, and also you cannot reduce the impedance any further in order to draw more current.

A current source, by contrast, is an energy source with a constant current, regardless of the voltage. Its output impedance is infinite. A photovoltaic cell in constant light at a voltage well below its maximum output voltage is a close approximation of a current source, because each photon it converts generates an electron-hole pair regardless of the voltage; I think its impedance is on the order of 1MΩ. Any current source has a limit to how much voltage it can supply — if you open-circuit the photovoltaic cell, its voltage will rise to about 0.7 volts and stay there. At that point you cannot reduce the

conductance any further in order to get higher voltages.

This is one of many applications of a certain duality in electronics in which we interchange serial and parallel, voltage and current, resistance and conductance, and I think capacitance and inductance, and maybe frequency and period, and everything comes out the same.

The study of electrical currents began with batteries, which are pretty close approximations to voltage sources. Current sources, on the other hand, are generally thought of as artificial or theoretical constructs. But in today's world of power electronics, our voltage sources are usually complicated circuits, and we can build current sources just as easily. Stick-welding machines and TIG welding machines are commonly designed as high-power current sources, under the name "constant-current source", or "CC", for example.

Why are RF current sources an interesting alternative?

It turns out that if you use a current source rather than a voltage source for your mains power, and run it at a higher frequency, there are a number of interesting results. In particular, all of the bullet points in the first section above become false, and there are a number of potential safety, cost, and complexity benefits — although of course we won't know if these work out in practice until we have real experience with them. Current sources defy the expectations we have built up over centuries of working with voltage sources — among other things, about what is safe and what is dangerous. To take the simplest example, short-circuiting a high-power current source is totally safe and actually the right way to turn off whatever it's powering. Open-circuiting it, by contrast, can produce a hazardous voltage.

The extremely low frequencies we're accustomed to using date from the late 19th century, when they derived from the frequencies of rotation of the steam-driven generators used to produce the power. This approach is still used today, with generators' rotation synchronized to the grid frequency before putting them online. But supplying power at higher frequencies has a number of substantial advantages in safety, electronic noise, and equipment size, cost, and complexity.

In addition to potential uses in houses and industry, this system should also be useful in spacecraft, where wiring harnesses commonly contribute an alarming amount both of very expensive mass and of very expensive power consumption.

Basics

Consider a system supplying one ampere at 32768 Hz, the frequency of a watch crystal, to every wall outlet, with current-source compliance up to 1000 VAC.

The outlets on a circuit are wired up in series — the current flows first through one appliance, then another, then another. When no appliance is plugged in, a short-circuit shunt is mechanically imposed across the two terminals of the outlet, so current flows through the outlet with no significant voltage drop. When all the outlets on a circuit are short-circuited, the total resistance of the circuit might be

1Ω, so the 1A will produce 1V of voltage drop and 1W of power consumption.

If a 10W appliance is plugged in, once its contacts have made contact with those of the outlet, its plug mechanically removes the short-circuit shunt, introducing the appliance into the circuit. The appliance's impedance of 10Ω produces a voltage drop of 10V across the plug (once the current source's output voltage rises to match), delivering 10W to the appliance.

A switched outlet or appliance has the switch in parallel to it. When the switch is closed, there is no voltage across the appliance, and consequently no current through it, or a tiny residual current due to the tiny voltage drop across the closed switch. Opening the switch forces the current to flow through the appliance instead.

A short circuit between an outlet and the neutral wire removes power from the appliances downstream from it, but causes no other problems. An open circuit, however, could cause a hazardous voltage, and therefore safety measures must be taken to shunt current past the open circuit, just as fuses or circuit breakers must create an open circuit in a voltage-source system to interrupt hazardous currents.

All the normal wires are the same tiny size, 24 AWG, 510 μm in diameter, which is enough to carry 1A without getting warm, but not so thick as to waste any significant copper due to the 400-micron skin depth in copper at 32768 Hz. Higher-current internal ac wires in some appliances need to be made of copper tape of less than 1 mm thick. The thickness of the wire does not depend on the length of the run, because the voltage drop is inconsequential; the current source will automatically raise the voltage to compensate, so that the load will receive the full 1A anyway, as long as the run isn't so long as to approach the current source's compliance limit (1000 V, thus 1000 Ω at 1 A.)

Physically, the outlets are strange. The two prongs are each 20 mm long, with their first 15 mm insulated, as in France; they are positioned a rather large 40 mm apart, with a ground prong in the middle. Behind the outlet, the attachments for the tiny wires are on strange stalks that curl back around. All of this is for high-voltage safety in the unlikely case that a single outlet is called upon to supply the full 1000 V limit of a circuit; it needs the creepage allowance not to form a conductive, but high-resistance, path between the electrodes.

Despite the possibility of using a single wire from one outlet to the next, in fact the return wire runs alongside the hot wire as a lightly twisted pair for the whole circuit in order to reduce EMI. It isn't connected to anything until the last outlet in the string.

Each circuit is fed from a separate transformer, a bit smaller than the usual circuit breaker, on a higher-voltage-compliance constant-current "bus", which is wired in series, not in parallel as usual. The transformers are center-tapped, and the center tap is grounded, so the net voltage surrounding the twisted pair of wires is zero, and of course their net current is also zero. This reduces EMI and keeps the voltage from any point to ground to a minimum. Each transformer also has a safety shunt attached to it.

Safety

First, in at least a couple of ways, the system I'm describing here is

potentially *more* dangerous than the traditional system. 240V is often not enough to kill you, depending on how dry your skin is. 120V almost never is. But one ampere through your arms is traditionally considered very likely to kill you, and the current source will happily apply 1000 volts to you if you are suddenly the only path through its circuit and that is what is needed to drive an ampere through you. 1000 watts applied to your body is also very likely to kill you.

Also, kilovolts can do surprising things that the more usual voltages do not do, jumping through air and burning tracks through dust on surfaces and whatnot.

However, there are a variety of ways this system is inherently safer, as well as a number of safety features that can be added.

First, the vast majority of circuits will not have hazardous voltages present on them at all, because they will not have hundreds of watts of load, so they won't need hundreds of ohms of impedance. And even those circuits that do have hazardous voltages will generally not have hazardous voltage differences within a single appliance.

??? I'm conflicted about whether grounding the center taps is a good idea. Without that current path to ground, there would never be any hazardous voltages relative to ground in the system, only potentially hazardous voltage differences within the circuit. But without it, grounding an enclosure will not produce a detectable ground fault if a hot wire comes in contact with the enclosure.

As explained above, short circuits are not hazardous in a current-source system — the current source will only supply its usual current to them, rather than an unbounded current, and they will dissipate no energy. Open circuits produce a hazardous voltage but no immediate fire. The real danger is near-open circuits of a few kΩ, which could potentially dissipate a few kW.

I think one amp is not enough to sustain an arc in air (???), which would eliminate the usual risk of arcing, despite the high voltages.

Using RF is a safety advantage because currents at these frequencies cannot (???) penetrate deeply into the human body, instead staying on the skin, and so they cannot cause muscular contractions (???) or cardiac fibrillation. They can still cause burns, and RF electrical burns are notorious, but that requires a lot more energy. Unfortunately, this system is capable of supplying that energy.

While it's harmless and probably useful to include traditional series overcurrent fuses in the system — if the current deviates significantly above 1 A, you have a bad problem in your current source and it would be a good idea to disconnect from it — they won't detect hazardous voltage soars or open circuits, which are the kinds of problems that could arise from electrical faults inside your house.

Grounding the cases of appliances provides a more effective safety measure against hot cases than in the traditional voltage-source system. In the standard system, a hot wire contacting the grounded case produces a ground fault, which fires the circuit breaker. In this system, by contrast, it shorts out half of the transformer and the upstream appliances, effectively dividing the circuit into two circuits of 500 mA each, joined at a shared ground point. The enclosure remains grounded and thus safe to touch; there are no sparks and no danger of electrical fire. However, nothing on the circuit will work properly until the faulty appliance is removed, because it will be getting a quarter of its usual power.

The simplest safety shunt against open circuits is just a buzzer-type SPST relay; its normally-closed contacts in series with a normally-closed pushbutton short the load, and its winding in series with the load holds those contacts open. If at some point the load goes open-circuit, the current through the load and the coil will cease, the contacts will spring closed, and the load will remain shunted out of the circuit until someone pushes the pushbutton†. This fails safe in case of coil failure and in case of power failure; it may be inconvenient to have to reset all your circuits after a power failure, but it's better than having to replace all your appliances. It may be possible to tune this circuit to reliably detect the ground-fault half-current case, too.

The other problem the above circuit has is that it doesn't actually limit the voltage; it just responds to the source's inability to sustain its current in the face of overwhelming resistance. In the case where the source is actually capable of hitting 10kV, it might fail to activate because the deadly overvoltage has burned a track across the outlet and is efficiently heating it to incandescence.

A simple truly-overvoltage-driven alternative would be a large gas tube. A fluorescent lamp tube would sort of work, but its discharge maintenance voltage is high enough (in the range of 100V) that it would probably cause major damage to the faulty load.

A fully-solid-state alternative might be a diac in series with a small inductor and a NC pushbutton, driven from a bridge rectifier in parallel with the load. If the diac goes into conduction due to overvoltage, it should crowbar the circuit rather effectively until someone presses the button; the inductor keeps it in conduction as the voltage crosses through zero 65536 times per second. This should give you under 10V across the circuit. (e.g. the Littelfuse K1400GURP SIDAC can handle an amp steady-state and will crowbar down to 1.2 volts, with a breakover voltage of 130–146V; you could use 7 of them in series, giving you a breakover of 910–1022 volts, and a voltage of 8.4 volts, plus the bridge and inductor voltages. It costs 37¢ in bulk.)

An alternative relay circuit would use a latching relay activated by current through, say, a calibrated spark gap, MOV, diac, or gas tube. It could latch either mechanically or, by virtue of energizing its own coil, electromechanically. This last has the dubious advantage of resetting automatically after power failures, including after being shunted out of the circuit by an upstream shunt.

† At which point the circuit potentially goes really open-circuit if the load still isn't repaired, thus causing further safeties to trip further upstream, so maybe this design is inadequate. Or maybe you just need two of them, maybe controlled with a make-before-break pushbutton that opens one circuit after closing the other.

Appliance design

Everything is topsy-turvy in the current-source world, but it's overall simpler. It's fine for things to short out when they fail; it's problematic and possibly dangerous for them to create open circuits, for example by breaking the filament of an incandescent lightbulb. If your lightbulb filament breaks, it will trip the safety shunt, and you will probably want to turn off all the lightbulbs on the circuit, reset the safety, and then turn them on one at a time until you figure out

which one is tripping the safety.

Resistive heating is simple. 1Ω is 1W. To get a given temperature under given cooling conditions, you use a given filament; you get the same temperature no matter how much of it you use, but more filament gives you more power. You should probably mount it on some kind of conductive backing with light insulation in between so that it becomes a short circuit if it melts, not an open circuit.

More resistance gives you more power; this is precisely backwards from the situation with voltage sources, where load resistance and power are inversely proportional.

3V 1.5W LEDs can be driven directly from the regulated 1A with no resistor. Smaller LEDs will burn out; larger LEDs will only shine 1.5W. Large fluorescent tubes of about 100W can, too, with no ballast or starter, as long as they can cold-cathode start at a low enough voltage, and fast enough to keep the safety shunt from tripping. 50W tubes should work if half-wave rectified — with one diode in series and another opposite diode shunting the combination. But most LEDs and fluorescent lights will require some kind of power supply.

Voltage-step-up transformers can be used to step down the current for some of these devices, such as small LEDs. Voltage-step-down transformers are probably useful to step up the current for larger resistive elements, so that they can use thicker wire.

Low-voltage low-power power supplies are really easy; a tiny 1A 47-microhenry inductor has an impedance of $\omega L = 4.8\Omega$ and thus a voltage across it of 4.8V (ac rms), which is 6.8V peak. Stick a silicon bridge rectifier across that, dropping 1.4V, and you can charge a capacitor to 5.4V dc at any current up to 100mA average or so (at which point the inductor voltage starts to drop because you're stealing its current). For 10% ripple at 100mA, the capacitor itself only needs to bear the 100mA for 15.3 microseconds without dropping more than 0.5V, which means 3 μF is adequate; an LC filter would allow you to use a smaller capacitor and also get better stability. If you want a regulated voltage, you might want to use a slightly larger inductor to get more like 9V peak, then drive a 7805 off it. Then it would be okay to steal more current from the inductor.

In any case, you won't have any 50Hz or 60Hz ripple. You'll have 32768Hz ripple, plus some harmonics like 65536Hz and 98304Hz, which are a lot easier to filter out.

Combining the two above, you can quite reasonably use a voltage-step-down transformer to feed a 22μH inductor 5A, then rectify, filter, and regulate, to get a 60% efficient 1A 5V regulated USB power supply in five components.

In the simplest case, of course, you can get a voltage by using a resistor rather than an inductor, but it will dissipate power. Or you could use an unpolarized capacitor of a few μF.

Inrush currents from hooking up capacitors to line voltage no longer exist. Instead you have inductive voltage spikes when you switch an inductor into the 1A circuit. You may want a snubber network to tame this, but that may be unnecessary — you aren't going to design in an inductor with more than 1kΩ of reactance, which means more than 10mH. $\frac{1}{2}LI^2 = 5$ mJ, which may not be enough energy to cause any real problems.

Switching power supplies are still a thing, despite the greatly

increased convenience of old-style transformer power supplies, and they need no inductors. If you were using dc instead of ac, a switching power supply amounts to switching between a dead short and a capacitor-diode series. You run your voltage-regulated circuit off the capacitor, and the diode keeps the capacitor from discharging during the time that the power is turned off with a short. You can do this with ac if you use a bridge rectifier or just a reverse-protection diode. For lower-power supplies, you could just use a rectifier diode to generate a voltage dissipatively and switch that voltage onto a capacitor at times.

Squirrel-cage induction motors should be barely feasible: 800 poles around the stator should give you 164 revolutions per second, which is 9840 RPM. The "squirrel cage" should probably be a millimeter-thick copper or aluminum sheet. I don't think you can use laminated electrical steel in the rotor at these frequencies. On the plus side, the starter capacitor can be quite small.

Universal motors are not feasible at all without rectification.

Wire diameter, resistance, and cost

The higher peak voltages allowed by a current-source system, as well as the automatic compensation for wiring losses, allow us to use much lower maximum currents and therefore thinner wire than in the traditional system. Instead of building the copper for the worst case and the insulation for the average case, it allows us to build the copper for the average case and the insulation for the worst case. Insulation is much cheaper than copper.

24AWG copper wire is 510μm in diameter and 84Ω/km, and it's recommended for currents up to 3.5 A. 84Ω/km means 84mΩ/m and thus, at 1A, 84 mW/m, which is not totally insignificant in terms of heating, but not really dangerous either. It means you can go over 5 km round trip with this wire before you hit the source's 1000 V compliance limit.

A bigger issue at these frequencies is likely to be stray inductance. 1Ω is only 9.7 μH. The twisted pairs should reduce this problem, but they won't eliminate it.

Copper's density of 9.0 g/cc means this works out to 1.8 grams per meter of wire, or 3.6 g if you run the neutral wire next to the hot one. But that doesn't count the insulation, which needs to be safe at 1000 V.

100 turns of this wire on an inductor work out to a coil, say, 5.1 mm long and 5.1 mm thick, for maybe 15 mm of diameter on the whole inductor. Transformers might be a little larger. None of that takes into account the insulation, though, so probably the reality will be several times that.

1000 V probably needs over 16 mm clearance between exposed conductors and 64 mm creepage distance, although I should look up the standards.

As explained in the "appliance design" section, the higher frequencies allow the use of much smaller capacitors, inductors, and transformers, which dramatically reduces cost and weight. The current-mode design allows us to get whatever voltage we want by adjusting a reactance; the use of rf ac allows us to get whatever current we want with a tiny transformer.

Etc.

are too high for humans to hear, so they cause less problems with audio equipment; and they

Topics

Dilating letterforms

Kragen Javier Sitaker, 2018-11-04 (15 minutes)

So I was thinking about how to dilate and erode letterforms, in order to generate a range of fonts from some scanned hand-drawn characters. This has a couple of applications. First, hand-drawn characters may be fairly nonuniform in line thickness, and you may want to make the line thickness more consistent; some kind of nonlinear erosion to a skeleton, followed by dilation, can give you uniform line thickness. Second, scanned data may have flecks, and various compositions of dilation and erosion are well known to eliminate flecks below a critical dimension. Third, you may want to apply calligraphic effects to letters, such as changing the pen angle or nib width, or providing a nib width to pencil-drawn characters.

Thresholded-blur-based dilation and erosion

One of the algorithms that occurred to me was to blur the characters and then threshold the blurred characters, perhaps at a much higher resolution than they were scanned at, converting each original pixel to a discrete impulse in the center of a much larger empty space before blurring. This, in a sense, provides a vectorization mechanism for a bilevel image; you can blur and threshold the characters at an arbitrarily high resolution, and thanks to Nyquist's sampling theorem and the convolution theorem, you'll always get the same results as long as your blur is the sampled approximation of whatever true analytic blur you're blurring with, and as long as it doesn't have any significant frequency components above the Nyquist frequency, whether that's a Gaussian, a sinc, or something else.

However, this contains a pitfall. What happens if we look at the contours of a blurred set of impulses like this? If we fill in the empty area with zeroes, as the theory says we should, then we'll find that the positive contours around positive impulses are convex, as are the negative contours around negative impulses, but as we cross zero going away from a positive impulse toward an adjacent negative impulse, we find our contours becoming concave. If we have a checkerboard of ±1 impulses, for example, and a circularly symmetric blur kernel, then the positive contours will be near-circles around the +1 impulses, while the negative contours will be near-circles around the -1 impulses. Only at 0 do we find the contours running in straight lines, dividing the checkerboard into positive and negative squares.

So I propose that what we should do is that, before upsampling, we should add or subtract a constant to the impulses so that our desired threshold will be, in the transformed value space, 0.

If the blur kernel has limited support, the contours in each cell or "tile" of the grid will depend only on some limited neighborhood of impulses, and of course the threshold. If the support is one pixel or less (of the original image), it will depend only on the four neighboring pixels, like Perlin noise. Like Wang tiles, the tile is guaranteed to match its neighboring tiles by virtue of shared edge colors (corner-pair colors, in this case). Even if the kernel support

takes into account a 4×4 area, for bilevel input images, it's feasible to precompute all the possible tiles. Cubic spline kernels will have continuous second derivatives, which means their contours will also have continuously changing curvature.

Furthermore, it's possible to compute points on the contours directly, without resorting to actually computing the pixels of a high-resolution upsampled image. If the blur kernel is polynomial or piecewise polynomial by tiles, then the blurred image within a given tile is a single polynomial function — bicubic in the case of splines over 4×4 neighborhoods, and in any case a linear combination of parts of the blur kernel.

This generalized erosion or dilation operation should be more capable of preserving smooth curves than the standard iterated morphological operations, but it might also lead to ink blotting at line joins.

I thought that by using an asymmetric blur kernel, it should be possible to convert strokes of uniform weight into strokes of varying width according to their angle, but I'm not sure that's actually true. It's depending on the final thresholding operator to introduce the nonlinearity where, say, vertical strokes convert into more ink than horizontal strokes. And I'm not sure simple thresholding is up to the job.

Erosion to skeletons

Erosion-to-skeletons is a nonstandard morphological operation. Standard erosion can be done on a 3×3 pixel window; if any input pixel in the window is empty, the resulting output pixel in the center of the window is empty. But this can erase lines or dots completely, and indeed such despeckling is one of the uses for standard erosion.

Suppose that instead we use a 4×4 pixel window. Now we can see whether the pixel we're about to erase is part of a line or speck that's already been reduced to a thickness of 2 or 1. In that case, we can preserve a thickness-1 line, although if we're reducing thickness 2 to thickness 1, we have to choose a bias direction in which to move the line: left, right, up, or down. My hope is that we can neutralize this bias by running successive passes of the algorithm with biases that cancel out, for example using the Thue-Morse sequence to alternate directions.

Pipelined morphological operations

In the 1970s and early 1980s, the ERIM Cytocomputer, a special-purpose parallel architecture, had the fastest implementation of the Game of Life, although it was primarily used for machine vision applications. Rather than dedicating one processor to each board region, it dedicated one processor to each time step, pipelining cells from one processor to the next. In this fashion, a Cytocomputer with a 16-stage pipeline (I don't remember if this was a normal size) could output one Life cell every clock cycle, 16 generations later than the generation that had been fed into the pipeline, with slightly over 16 rows of latency. I don't remember what the clock speed was, either, but I think it was a few megahertz.

You might think this would require a massive memory for each processor, but in fact each one only needs a couple of reasonable-sized FIFOs. To compute the cell marked O, it needs to consider the

previous-generation states of the cells marked X and the cell marked O, and needs to retain the previous-generation states of the cells marked .:

```
                  XXX.....................................
..................XOX.....................................
..................XXX
```

In a pipeline, each pipeline stage is computing one scan line plus two pixels behind the previous one; note that most of the pixels are duplicated in two different FIFOs:

```
                  XXX.....................................
..................XOXXX...................................
..................XXXOXXX.................................
....................XXXOXXX...............................
......................XXXOX.................................
........................XXX
```

The pixels were 8-bit bytes, the FIFO size was either fixed or capped at 1024 or 2048 or 4096 or something, and in addition to the Life rule, the Cytocomputer was capable of a variety of machine-vision-relevant operations such as thresholding, edge-detection (I don't remember, probably at least Canny), dilation, and erosion, which like the Game of Life can all be computed as a finite function on a 3×3 neighborhood, and of course in these applications, each stage of the pipeline could be programmed to perform a different function. You programmed it with a bletcherous scripting language on an attached general-purpose computer.

I think you can do something similar with SSE on a modern CPU, but perhaps many pixels per cycle instead of one, and maybe only one pipeline stage per instruction. The idea, though, is to stream pixel data into the L1 cache (from another cache or from main memory), apply a whole generic pipeline of such local operations to it, and then stream it back out of L1 cache with the whole set of operations applied.

Digression: tiling

Usually for this kind of processing you want to tile the pixel data to reduce the size of the necessary FIFOs, although that makes the indexing somewhat more complicated. "Tiling" means that you turn the usual two dimensions of image data into four dimensions: an X and Y to locate the tile, and an x and y to locate the pixel within the tile. For example, with 8×8 tiles, the pixels in a 16×16 image might be in this order instead of the usual one:

```
  0   1   2   3   4   5   6   7  64  65  66  67  68  69  70  71
  8   9  10  11  12  13  14  15  72  73  74  75  76  77  78  79
 16  17  18  19  20  21  22  23  80  81  82  83  84  85  86  87
 24  25  26  27  28  29  30  31  88  89  90  91  92  93  94  95
 32  33  34  35  36  37  38  39  96  97  98  99 100 101 102 103
 40  41  42  43  44  45  46  47 104 105 106 107 108 109 110 111
 48  49  50  51  52  53  54  55 112 113 114 115 116 117 118 119
 56  57  58  59  60  61  62  63 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 192 193 194 195 196 197 198 199
```

```
136 137 138 139 140 141 142 143 200 201 202 203 204 205 206 207
144 145 146 147 148 149 150 151 208 209 210 211 212 213 214 215
152 153 154 155 156 157 158 159 216 217 218 219 220 221 222 223
160 161 162 163 164 165 166 167 224 225 226 227 228 229 230 231
168 169 170 171 172 173 174 175 232 233 234 235 236 237 238 239
176 177 178 179 180 181 182 183 240 241 242 243 244 245 246 247
184 185 186 187 188 189 190 191 248 249 250 251 252 253 254 255
```

With 4×4 tiles, instead you would have this order:

```
  0   1   2   3  16  17  18  19  32  33  34  35  48  49  50  51
  4   5   6   7  20  21  22  23  36  37  38  39  52  53  54  55
  8   9  10  11  24  25  26  27  40  41  42  43  56  57  58  59
 12  13  14  15  28  29  30  31  44  45  46  47  60  61  62  63
 64  65  66  67  80  81  82  83  96  97  98  99 112 113 114 115
 68  69  70  71  84  85  86  87 100 101 102 103 116 117 118 119
 72  73  74  75  88  89  90  91 104 105 106 107 120 121 122 123
 76  77  78  79  92  93  94  95 108 109 110 111 124 125 126 127
128 129 130 131 144 145 146 147 160 161 162 163 176 177 178 179
132 133 134 135 148 149 150 151 164 165 166 167 180 181 182 183
136 137 138 139 152 153 154 155 168 169 170 171 184 185 186 187
140 141 142 143 156 157 158 159 172 173 174 175 188 189 190 191
192 193 194 195 208 209 210 211 224 225 226 227 240 241 242 243
196 197 198 199 212 213 214 215 228 229 230 231 244 245 246 247
200 201 202 203 216 217 218 219 232 233 234 235 248 249 250 251
204 205 206 207 220 221 222 223 236 237 238 239 252 253 254 255
```

(This kind of tiling also made it possible to interact with many-megapixel images in real time on 1980s graphics workstations. Perhaps your LANDSAT scene was 50 megapixels and your machine only had 8MiB of RAM. Without tiling, in the format the LANDSAT people would deliver you the images, the megapixel you could fit on the monitor was 1000 fragments splattered across 7 megapixels on disk, which was 21 megabytes in RGB or 43 megabytes in its full multispectral glory, and if you want to pan 100 pixels to the right, those pixels are splattered across those same 21 megabytes; so even a slight panning would induce a multi-second disk wait. With 16×16 tiling, your screen would instead contain parts of some 60–70 rows of tiles, which could be accessed in 60–70 seeks: about a second. JPEG image compression was not yet a thing, and even today its use for remote-sensing imagery is dubious.)

8×8 tiles are a typical size, but I suspect that 32×32 tiles (or even bigger, 64×64 or 128×128) might make more sense for most purposes on modern hardware.

(End of digression.)

Bitwise morphological operations

For morphological operations, we might really benefit from bit-packing, where we can get 128 pixels into an SSE register (or 256 pixels into an AVX register). Then, the erosion or dilation operation (depending on interpretation) is mostly `a | a << 1 | a >> 1 | b | b << 1 | b >> 1 | c | c << 1 | c >> 1`, but that only produces 126 correct bits of new state for b; the first and last bits are incorrect.

Note that this is a separable operation: we can do it first horizontally and then vertically. So we could, for example, compute `a`

`| a << 1 | a >> 1` — plus the correction for the first and last bits — once, and use it to compute results on all three scan lines.

Topics

Gaussian spline reconstruction

Kragen Javier Sitaker, 2016-06-05 (updated 2016-06-06) (5 minutes)

You can get a very good approximation of the Gaussian function by convolving several box or simple-moving-average filters, in the same way that you get a good approximation of it with a spline, in a way that I don't quite know how to name right now, but which is exactly equivalent to convolving several box filters.

From a computational perspective, notable features of this procedure include that box filters require no multiplications, require a very small number of operations per sample (only two additions and a subtraction, plus a FIFO buffer of the size of the box), and can be computed exactly with integer math in which the intermediate results overflow, as long as the final result does not. XXX is that even true? WTF

Precisely converting discretely sampled signals back into bandlimited continuous signal involves convolving them with the sinc function (the Fourier transform of the unit pulse) which is inconvenient because it decays very slowly, meaning that doing the convolution directly (in the spatial domain) requires summing weighted contributions from quite far away to get reasonable accuracy. If you just do a zero-order hold, turning the sampled curve into stairsteps, you are in effect convolving the modulated Dirac comb that is your discretely sampled signal with a unit pulse, thus filtering its frequency response with a sinc spectrum, which is disastrous for frequencies above about half Nyquist.

One approach to reconstruction filtering is to use Nth-order splines that pass through y=1 at x=0 and have zeroes at all other integers. (These are sometimes called cardinal B-splines, by analogy to the sinc "sinus cardinalis" function, but that terminology is arguably wrong and at least confusing.) These have very compact support, and they approach sinc in the limit where $N \to \infty$. But they definitely require multiplication.

Derivatives of the Gaussian include the so-called Mexican Hat wavelet, Ricker wavelet, or Marr wavelet, which is its negated second derivative. This is vaguely close to sinc: it has a big round bulge in the middle with a zero on each side of it, then local minima, and then it asymptotically approaches zero as it goes to infinity.

I have this idea that the second derivative of the convolution of a signal with the Gaussian is equal to the convolution of the signal with the second derivative of the Gaussian, because both differentiation and convolution with something are linear time-invariant transforms. But now I'm not sure if that's true.

The Mexican Hat Wavelet page on Wikipedia says, "In practice, this wavelet [in multiple dimensions] is sometimes approximated by the difference of Gaussians function, because the DoG is separable." If this is true (it's not backed up by the reference given) then it seems like a better way to convolve things with that wavelet in most cases would be to use the procedure I've outlined above to approximate the convolution using the Gaussian-then-differentiate approach. Convolving five box filters gives you a quartic approximation of a Gaussian; its second derivative is then a quadratic approximation of

the Mexican Hat Wavelet. Note that this entire computation requires no multiplication.

Higher derivatives might also be useful, as they oscillate more times and hit more zeroes before dying out; they are the Hermite functions multiplied by the Gaussian. Their zeroes aren't really in the right places, but that error might not be large enough to matter; H_4 has zeroes at ±sqrt((3±sqrt(6))/2)), which is about ±0.523 and ±1.650. That doesn't oscillate nearly fast enough to be a good approximation for sinc. The higher-order Hermite polynomials' roots have a similar problem: their zeroes are roughly evenly spaced, while sinc has an double wide space at x=0 where there would normally be a root.

These derivatives do, however, look very much like delicious zero-phase FIR narrow bandpass filter kernels, which are more or less like the Gabor kernel. So they may not be that useful for signal reconstruction, but maybe they could be very useful for narrow bandpass signal processing!

I thought maybe it was just as bad for reconstruction that the higher-order Gaussian derivatives vanished very quickly, but maybe that's just an unavoidable artifact of a reasonable degree of locality, which is not an altogether bad thing.

(I got a lot of this information from http://bmia.bmt.tue.nl/people/bromeny/MICCAI2008/Materials/005%20Gaussian%20derivatives%20MMA6.pdf, which even mentions the use of Gaussian derivatives as bandpass kernel filters.)

Topics

- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Splines (p. 3727) (6 notes)

Virtual instruments

Kragen Javier Sitaker, 2015-11-09 (3 minutes)

Playing with Carolina's idea of an expanded theremin.

There are two major questions: inputs and outputs.

Inputs

Minority-Report-style lights on hands in front of a webcam?

Kinect body scanning in real time?

Arduinos with accelerometers?

Power-glove-style strain gauges?

Theremins use capacitive sensing. This is clearly a possibility, but it is harder with high-speed electronics around, since the capacitance signals you're looking for are very small. An actual theremin circuit might be the best way to measure proximity.

Ultrasound signaling is another traditional approach for VR equipment.

Outputs

Some of this is tricky because the input channels we're considering are possibly pretty high latency: webcam plus image processing algorithms, say. So it might make some sense for the musician to control the tempo and type of rhythm rather than the exact timing.

(Some prototyping will reveal how big a problem this is in practice. You could imagine that a QVGA 320x240 image wouldn't really take that long to process, and might be adequate.)

The possibility of the music coming out is, fortunately or unfortunately, many-dimensional. Even a single note can be loud or soft, high or low, bent up or down, of many different timbres, and of many different envelopes, and at any time. An entire melody has many notes, and the relationships they bear to one another are potentially very complicated.

So one crucial question is how many low-latency high-bandwidth dimensions of input we can achieve: two with each hand? Another with angle? Another with finger separation? Other joint orientations?

One possible mapping is with radius for volume or timbre, and angle for pitch. Going around the circle more than once would go up or down a whole octave.

Another possibility is moving virtual objects around a space, maybe a time-frequency representation where time is streaming past you.

Another kind of exploration would adjust parameters of different state machines with your movements: a melody generator, a rhythm generator, a timbre generator or two. Maybe, at its simplest, you could have one direction to move in to seek change, and another direction to move in to stay the same; or you could have a sort of pie menu thing where you're always selecting one of the parameters to vary, and your distance from the center controls how fast it varies. It's crucial for the feedback here to be within a few hundred milliseconds in order for the interface to be discoverable.

Topics

On influencers

Kragen Javier Sitaker, 2019-05-16 (3 minutes)

Someone Tweeted today that it was absurd to have "influencer" as a job title, because it presupposes success — you might as well say you had been hired as a "bestselling novelist" instead of an author, for example.

There are a lot of job titles that are like this, though. It's commonplace to hear professors of philosophy use the word "philosopher" to mean "professor of philosophy", for example, as if Socrates had been a professor, or Einstein had been a gravitational field, or as if a tenure committee could confer a love of wisdom on a fool. "Hacker", too, is a title of acclaim, earned by achievement, not a hobby or a job title; calling yourself a "hacker" is one step below calling yourself a "genius", which is, of course, also used as a job title nowadays, by Apple.

There's a persistent meme that "engineer" is a similar sort of arrogant self-praise, since, ever since the disaster in Texas, various places have formal licensing requirements for "engineers" similar to those for braiding hairs or painting nails, with the ostensible purpose of preventing any more children from being exploded by poorly-designed gas systems or other machines. But an "engineer" has been, since at least the 15th century, one who makes engines; the licensing requirements are new and far from universal. Calling your creation an "engine", however, *is* a kind of self-praise, at least etymologically — you're saying it's *ingenious*, as the cognate word "engine" has meant "clever scheme" (and thence "machine") for over a millennium. Moreover, you're etymologically attributing this cleverness to your inborn qualities, thus "in+genitus", rather than hard work or good luck.

Surely we could go further with these flattering and optimistic job titles, though. Managers might feel left out of the fun, since their title implies they're just getting by; let's call them "organizational geniuses", or if, that makes them worry that you'll ask them to fix your iPhone, "master strategists". We could call taxi drivers "motorsport champions", and cooks "gourmet chefs" — though I think the world may be ahead of me on that one, since I'm pretty sure I've met self-proclaimed "gourmet chefs" who didn't have even a single Michelin star or prep cook. Oil-change mechanics could be "automotive engineers", though that doesn't really capture the same self-satisfied aspirational flavor as "influencer".

Topics

Designing a Scheme for APL-like array computations, like Lush

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

Here's a second whack at the problem of designing a Lisp dialect that lets you write array operations in terms of pointwise transformations, by means of static dependent type inference, in order to avoid run-time type and bounds checks.

First, array indexing. Arrays are a kind of function, but unlike other functions, you can find their bounds with the expression (bounds anArray). Arrays always take only a single argument, which can be either an integer within their bounds, or a vector of integers within their bounds.

Second, because vectors are so fundamental, there's a special syntax to construct zero-based vectors: [a b c d ...], which kind of looks like Scheme's #(...) syntax but doesn't quote, so it's semantically more like Python's list syntax or Squeak's { a. b. c } syntax.

So if you have a vector x and you want its fourth element, you can write (x 3) or (x [3]). As special syntactic sugar, if you leave out the space before the [, you get an extra layer of list wrapped around: x[3] is equivalent to (x [3]).

The (define ...) form from Scheme supports an extra feature not found in R5RS, which is already in MzScheme --- the thing being defined can be arbitrarily deeply nested on the left side. So all three of these definitions are equivalent:

```
(define x (lambda (y) (lambda (z) (+ y z))))
(define (x y) (lambda (z) (+ y z)))
(define ((x y) z) (+ y z))
```

This is so that you can define array-valued functions conveniently:

```
(define (matrix-multiply m n)[i j] (sum k (* m[k j] n[i k])))
```

syntactic sugar for (define ((matrix-multiply m n) [i j]) (sum k (* m[k j] n[i k])))

(bounds anArray) returns the bounds of the array as a vector of 2-element vectors, each indicating the minimal valid index and one more than the maximal valid index. So a 2x3 array that's all zero-based would have a bounds of [[0 2] [0 3]].

It is an error to construct non-rectangular matrices, because they don't have bounds that can be expressed in the above form.

For each array-valued function, the compiler infers a function that constructs the bounds of the resulting array from the bounds of its arguments. The bounds of the result are the widest possible bounds that the compiler can guarantee will not result in out-of-bounds accesses to any other array. Some arrays constructed by functions may be infinite, such as the generalized identity matrix (or Kronecker delta?):

```
(define identity[i j] (if (= i j) 1 0))
```

There's a (narrow newbounds array) form that can be used to artificially narrow the bounds of some array. It ensures that the new bounds don't include any values excluded by the old ones.

```
(define (nidentity n) (narrow [[0 n] [0 n]] identity))
```

Of the matrix-multiply function earlier, we can infer:

• m and n must both be two-dimensional;
• the first dimension of m must have the same range as the second dimension of n;
• the first dimension of the result has the same range as the first dimension of n;
• the second dimension of the results has the same range as the second dimension of m.

There's a (valid? expr) form which checks whether expr would cause a bounds error, without actually evaluating expr; the compiler can use this to infer bounds-construction functions.

...

Topics

• Programming (p. 3658) (286 notes)
• Programming languages (p. 3656) (47 notes)
• Arrays (p. 3326) (17 notes)
• Lisp (p. 3552) (9 notes)

Minimal GUI libraries

Kragen Javier Sitaker, 2015-11-14 (updated 2015-11-15) (5 minutes)

So I was thinking about what's needed to make a GUI library. You can get a substantial amount of it out of this piece of `propfont.c`:

```
/* This is sort of like a two-dimensional, dynamically-typed pointer
 * into pixels.
 */
typedef struct {
  char *pixels;
  int pixel_bytes, line_stride;
} pixel_buf;

static inline char *index_pixbuf(pixel_buf buf, int x, int y) {
  return buf.pixels + y * buf.line_stride + x * buf.pixel_bytes;
}

static inline pixel_buf offset_pixbuf(pixel_buf buf, int dx, int dy) {
  pixel_buf result = {
    index_pixbuf(buf, dx, dy),
    buf.pixel_bytes,
    buf.line_stride,
  };

  return result;
}

static inline void fill_pixbuf(pixel_buf buf, char *ink, int w, int h) {
  for (int x = 0; x < w; x++) {
    memcpy(index_pixbuf(buf, x, 0), ink, buf.pixel_bytes);
  }

  for (int y = 0; y < h; y++) {

    memcpy(index_pixbuf(buf, 0, y), index_pixbuf(buf, 0, 0), w * buf.pixel_bytes)o
o;
  }
}
```

That doesn't give you clipping, but with those 23 lines of code, any function that takes a `pixel_buf` parameter can be persuaded to draw at any position on a BIP pixel buffer of the pixel format it's expecting. You could enhance it to track right and bottom edges of rectangular windows with another couple of lines of code, which would make `fill_pixbuf` more convenient to call in many cases. And it's going to be pretty efficient.

However, if you write a windowing system on top of primitives like these, whether structured-mode or immediate-mode, you are probably going to end up with a kind of 1980s-looking window system with no antialiasing or transparency that's entirely immune to hardware acceleration, although if you avoid overdraw, you might be able to manage reasonable frame rates anyway. Nowadays, we can do

better!

Modern GUIs have transparency (alpha), gradients, rotation and distortion, filtering, béziers, antialiasing, animation, and terrible performance. I think we can do better.

First, both 1980s and modern GUIs almost entirely lack texture. But texture is a ubiquitous attribute of actual physical objects, and serves to orient us to movement in a subtle way. It also has a tendency to reduce the visual impact of dirt on the display or RF interference on the video signal.

Second, both 1980s and modern GUIs are still mostly oriented toward paraxial rectangles, and don't have indirect illumination (with the exception of drop shadows in modern GUIs on text and elevated windows). This orientation robs us of a lot of the structural cues we're familiar with from the physical world. They also don't take advantage of the possibility of zooming, in part because of font hinting and slow rasterization.

Third, 1980s GUIs, modern GUIs, and web UIs all lack a great deal compared to traditional typesetting, from a certain point of view. They lack letterform variation, columnar layout and horizontal scrolling, reasonable hyphenation and justification, hanging punctuation, tactile interface, line width variation, and so on. Of course, they're dramatically better from other points of view: better contrast ratios, animation, interactivity, undo, generativity, and so on.

There's an additional, somewhat selfish question associated with all of this, which is that if you reimplement the standard WIMP interface, it's easy for people who see it to dismiss it as unoriginal or even a copy of existing software with the labels filed off. (Many internet commenters carelessly dismissed Biyubi Fénix as a knockoff of KolibriOS/Menuet, for example, apparently without looking at it.) So even if you can't do something *better*, doing something *different* is significant.

So what might that look like?

I frequently hear surprised noises when I turn on my computer with a full-screen shell window. Old people sometimes mistake it for MS-DOS, but young adults were born around 1995, after MS-DOS was already obsolete. Here in Argentina, many of them didn't even use computers until after 2000.

Most people have still never seen a ZUI (with the possible exception of Prezi), and so simple "infinite" zooming is one way to distinguish it. It also lends itself to animations easily, which is in fact part of Prezi's appeal.

Windowing systems (and the underlying VT100 emulation) are a poor fit for shell commands; you have the tendency to open new windows to run new commands before the old one is done, but then it's hard to find the old one. A sort of notebook interface would work better, where you always immediately get a new prompt, and the output for previous commands can expand *above* your current prompt; perhaps it should be truncated by default, like with a dynamic `head; tail` that also outputs `wc`. (If it gets too large, you might want to either block the process generating it or discard parts of it.) Then you can also take that data and feed it to a new process, without having to re-execute the old one.

Furthermore, in many cases alternate presentations are possible: numeric tables, barcharts, scatterplots; Bret Victor's work with Dan

Amelang has shown some of the possibilities of this kind of visualization, along with incremental dataflow tracing.

Topics

- Graphics (p. 3483) (91 notes)
- Small is beautiful (p. 3714) (40 notes)
- Graphical user interfaces (p. 3489) (23 notes)
- BubbleOS (p. 3352) (17 notes)

An electric furnace the size of a sake cup

Kragen Javier Sitaker, 2017-02-25 (updated 2017-03-02) (10 minutes)

(See also Millikiln (p. 2581).)

There are lots of ways to heat things up to, for example, fire ceramics. Setting things on fire is probably the most fun way, but it's inefficient and hard to control. You can also concentrate sunlight, absorb microwaves, or run electricity through refractory wires.

Suppose we reduce the kiln problem to make it easier: we have no convection, and we just want to heat the kiln to 660°, the melting point of aluminum, using one of the three non-fire methods. And let's say we're satisfied with a kiln about the size of a sake cup: 50mm across, 60mm tall, 118mℓ. How much insulation do we need?

(Note that in the sunlight case, you also need a sufficient solar concentration factor, which works out to be 42 suns, 4.2 million lux.)

Well, of course, that depends on our power budget. At 2500 watts the problem is easy; at 2 watts the problem is hard. As a reasonable medium, let's take 500 watts, which is 2 amps here in Argentina, less than a toaster, hair dryer, or microwave oven, and about half a square meter of sunlight.

With such a small kiln, we aren't going to get a very good approximation by pretending the thermal gradient is constant across our insulating wall, but let's do it anyway.

Our top and bottom walls are $2\cdot\pi\cdot(25\text{mm})^2 \approx 4000\ \text{mm}^2$; our outer wall is $2\pi\cdot 25\text{mm}\cdot 60\text{mm} \approx 9400\ \text{mm}^2$; total is $13400\ \text{mm}^2$. Our total thermal flux is $500\ \text{W}/(11400\ \text{mm}^2) \approx 37\ \text{kW/m}^2$. Over a difference of 640 kelvins, that's about $58\ \text{W/m}^2\text{/K}$. Expressing this as an insulation value (R-value) instead, we have $0.017\ \text{m}^2\ \text{K/W}$. Loose-fill vermiculite is 17 m·K/W (according to Regenerator gas kiln (p. 2653), though I don't know where it got that figure) so this requires about 1.2mm of vermiculite. Other insulators are in the ballpark.

So actually the constant-thermal-gradient approximation is going to be pretty good. I'd go with making it 10 mm or 20 mm of insulation anyway, just in case, and maybe doubling the linear size of the kiln.

500 watts into 118 mℓ works out to a bit over 4 W/mℓ, which is adequate for a few kelvins per second of temperature rise, if somehow the whole microkiln is full of solid and yet thermally homogeneous. Since that would reach full firing temperature in a few minutes, it seems like a more than adequate specific power.

In the sunlight case, say you go with a solar concentration factor of, say, 100, just to be safe, and you also go for a whole square meter, to get a kilowatt. Now you are faced with the problem that to squeeze the whole square meter into the kiln aperture, which ideally faces directly downwards, you must exceed this solar concentration factor; the kiln aperture can't be more than a 500th of a square meter at that kiln size. This is another reason to want to increase the size of the kiln, and by more than a factor of 2.

So, revised design parameters: internal volume is a cylinder 150

mm across and 100 mm tall. Now we have 1.8 ℓ of volume, 0.018 m² of surface area on the bottom to cut a sun hole in, and 0.059 m² of total internal surface area to lose heat through by conduction. We use 20mm of insulating refractory for the walls and floor, which is kind of crappy as insulators go, so maybe its insulation value is only 8 m·K/W; this works out to 0.37 W/K, which means that at 640 K of inside–outside temperature difference we need 236 W of power input to maintain temperature. But we actually have 400 W of power, a comfortable safety margin, which works out to 230 mW/mℓ, which would heat water by about four kelvins a minute if the thing were full of water, but in the more reasonable case where it contains like 200 grams of stuff, we get 40 kelvins a minute, which gets us to full firing temperature (disregarding heat loss through the insulation!) in 15 minutes.

(Hmm, Henan Sinocean says their high-alumina insulating firebrick conducts 0.18 to 0.5 W/m/K, which I guess is insulating at 2 to 5.6 m·K/W, at a density of 0.3 g/cc for the best insulation up to 1 g/cc for the worst. These numbers are very high for insulating substances, barely better than ordinary bricks, but they are only a little worse than the Sheffield Pottery insulating firebrick numbers in Wikipedia and the BNZ Materials insulating firebrick brochure, and I probably can't do better, though I might do worse. So I might need 40mm of insulation instead of 20mm.)

400 watts of sunlight is 0.4 m²; a solar concentration factor of 100 (theoretically capable of reaching 900°) then requires a hole of 0.004 m², which is nearly a quarter of the floor area, which is still probably okay. A solar concentration factor of 100 could also be thought of, in imaging optics, as 10× magnification, or, in antenna design, as 20dBi. Both of these are eminently achievable. You could even reach that level with a trough reflector, though it wouldn't be the easiest way.

Fusión Refractarios in Avellaneda offers imported 0.81 g/cc 0.3 W/m/K 1427° firebricks for $87, which are 229×114×63mm; at $87/(2·229·114 mm²) at 31½mm thick, the cost is $1670 per square meter. I could maybe get two or four of these firebricks and try to stick them together.

Ten meters of Kanthal costs AR$70, whether it's 0.2 mm, 0.3, 0.4, or 0.5, and it's good to 1250° — not ideal for a pottery kiln, but it would do. It's not clear which Kanthal this is; there are various grades of Kanthal; one is Kanthal AF, which is good to 1300° and has a resistivity of 1.39 Ω mm²/m, but doesn't come in such narrow gauges.

How do I figure out how much wire I need? In theory you can get an arbitrarily large amount of power out of a fixed voltage source by putting an arbitrarily small resistance across it, since $P = E^2/R$. However, in this case the crucial fact is not really the resistivity of the wire — as I thought it was — but its surface area! It needs enough surface area to emit the desired amount of power at its safe temperature — at these temperatures, almost entirely as radiant heat.

That's why the Kanthal data page gives its fully-oxidized emissivity: 0.7. So how much wire surface area do I need to emit 400 watts with emissivity 0.7 at, say, 1100°? $0.7\ \sigma\ T^4 = 141\text{kW/m}^2$, so I need 2800 mm² (0.0028 m²), about 5% of the total inner wall area of the microkiln.

That's somewhat alarming! But, it turns out, not fatal.

Ten meters of 0.5 mm diameter Kanthal has almost 7900 mm² of surface area, and so thus, emitting at 1100°, it emits 1108 watts; at 1200°, it emits 1470 W. So really four meters of it (AR$28) should be adequate. (I'm a little skeptical that the price is actually correct, since all the other prices seem to be much higher.)

(Is it bad that I'm ignoring the radiant heat absorbed by the wire? I think it's okay, since it's hypothetically so much hotter than everything else.)

There *is* the engineering question of making the resistance low enough to get enough power out, without making it so low you draw too much power. In this case we want about 2 amps at 220 volts, which requires 110Ω. Four meters of 0.5 mm wire should be about 10 · 1.39 Ω / (π (¼mm)²), which is close — it's 28 ohms. The easy thing to do is just go ahead and use a longer wire: at 7Ω/m, you need 15.5 meters. Or a narrower one: ten meters of the 0.4 mm wire should be just right, and has 12566 mm² of surface area.

At 150 mm diameter, that's about 20 turns of a spiral around, which puts the turns about 5 mm apart up the walls if I don't coil them. Or I could coil them and use, say, four turns.

The volume of refractory is, say, the difference between a 230mm diameter, 180mm height cylinder, and the 150mm diameter, 100 mm cylinder space within: 7478 mℓ - 1767 mℓ = 5711 mℓ. If we take a middle-of-the-road figure of 0.7 g/cc for the refractory, it's 4 kg of clay.

To control the temperature, we need a thermocouple, probably AR$200, probably type K.

What would this microkiln design look like if I wanted to include safety margins to ensure that it would heat up and wouldn't burn out? After all, many unexpected things will no doubt occur — perhaps the brick walls will leak, the Kanthal may be a counterfeit that burns out at 1100° or has the wrong resistivity, the bricks may conduct more than expected, and so forth.

I'm not sure what I would do to give it a *temperature* safety margin so that it wouldn't burn up if it hit 1400°, which is a temperature I'm extremely interested in. Except to make it purely solar, I guess, or arc-driven. The issue is the maximum service temperature of the wire. Kanthal A-1 is supposedly good to 1400°, and Kanthal APM is supposedly good to 1425°. Kanthal A-1 is available on MercadoLibre, but at a 15× higher price.

Topics

An algebraic approach to 3D geometry

Kragen Javier Sitaker, 2014-06-03 (updated 2014-06-29) (22 minutes)

I was thinking about how to model objects for 3-D printing in a more flexible and generative way than OpenSCAD provides, and a very interesting factorization of 3-D geometry occurred to me. It falls a little short of what I was hoping for, but it still seems like it will be somewhat useful.

You can compose all the common primitive solids, plus a large number that are not common in 3D graphics but commonly occur in the world, from a small number of primitives, belonging to a smaller number of types (in the programming sense of data types) and primarily function composition. Surprisingly, it might even be practical to compute things with.

In particular, this approach derives lines, squares, circles, arcs, ellipses, cubes (and parallelepipeds in general), helices, cylinders, cylinder walls, cones (including frusta), spheres, ellipsoids, paraboloids, logarithmic spirals, discs, cycloids, spirographs, machine turning patterns, and tori from seven or eight relatively powerful primitives, each in about two lines of code.

I say this is "algebraic" in the sense that I'm defining a set of primitive objects and operations on those objects that I think will map to the domain I'm interested in in a useful way. You could also say that it's "algebraic" in that it represents the 3-D objects as symbolic expressions containing variables that can be evaluated to get coordinates.

This is derived from some work Nick Johnson and I a few years ago for an algebra of two-dimensional paths for his sand-table plotter, which drew aesthetically appealing patterns in sand with a ball bearing controlled by a magnet below the sand.

Examples

I haven't tested these. See section "The algebra" for a brief explanation of the notation, or read on for a fuller explanation.

```
K(r)  = r...r
Z(v)  = X(K(π/2)) . Y(v) . X(K(-π/2))
TY(v) = Z(K(π/2)) . TX(v) . Z(K(-π/2))
TZ(v) = Y(K(π/2)) . TX(v) . Y(K(-π/2))
SY(v) = Z(K(π/2)) . SX(v) . Z(K(-π/2))
SZ(v) = Y(K(π/2)) . SX(v) . Z(K(-π/2))
SU(v) = SX(v) . SY(v) . SZ(v)
step = 0...1
line = TX(step)
square = line * TY(step)
cube = square * TZ(step)
turn = 0...2*π
twistedcube = square * (TZ(step) . Z(turn))
disc = TX(step) * Z(turn)
```

```
cylinder = disc * TZ(step)
twostep = -1...1
parabola = TX(twostep) . SY(twostep) . SY(twostep) . TY(K(1))
paraboloid = parabola * X(turn)
cone = (Y(K(π/4)) . line) * Z(turn)
frustum = (Y(K(π/4)) . TX(1...2)) * Z(turn)
circle = TX(K(1)) * Z(turn)
helix = TX(K(1)) * (Z(turn) . TZ(step))
arc = TX(K(1)) * Z(0...π)
ellipse = TX(K(1)) * (Z(turn) . SX(2))
torus = (TX(K(2)) . circle) * Y(turn)
cylinderwall = circle * TZ(step)
sphere = circle * X(turn)
ellipsoid = ellipse * X(turn)
helicalramp = TX(1...2) * (TZ(step) . Z(0...8π))
logarithmicspiral = TX(K(1)) * (Z(0...8π) . SU(step))
zoscillation = SY(K(0)) . X(0...8π) . TX(K(1))
spreadingripples = (line . SZ(1...0) . zoscillation) * Z(turn)
spirograph = TX(K(1)) * (Z(turn) . TX(K(2)) . Z(0..12π) . TX(K(-2)))
```

It's based on extrusion

"Extrusion" is a relatively general way of adding a dimension to a manifold. If you have a point, or a set of points, extruding them along some path gives you a curve, or a set of curves; and if you have a curve, or a set of curves, extruding them along some path gives you a surface, or set of surfaces. (If the path is discontinuous, you may discover points becoming sets of curves, or curves becoming sets of surfaces.) For example, you can rotate a point into becoming a circle, and the circle into becoming a torus; or you can translate the circle to make the wall of a cylinder.

A few years back, when I wrote a real-time 3D engine in JavaScript (XXX) with the 2-D canvas, this was where I stopped: extrusion took as parameters an object, a linear 3-D transform, and an iteration count, and gave you back an object made by extruding the original object through that transform an arbitrary number of times. The result was a kind of faceted crude approximation to a torus, although I should really have twiddled parameters a bit to get some kind of more interesting shape, like a spiral or something; if you wanted a finer approximation of a torus, you could halve the rotation angle and double the iteration count, and you'd have more facets with duller angles between them.

You'd think you could use this approach with arbitrary 3-D transformations, taking some kind of "square root" of the transformation in order to make a finer mesh where needed. This turns out to be impossible because even a simple rotation transformation has, in some sense, lost the information about its angle: if it's 0.1 radians around the z-axis, say, we have this matrix:

```
[ [ cos 0.1 -sin 0.1  0  0 ]
  [ sin 0.1  cos 0.1  0  0 ]
  [   0        0      1  0 ]
  [   0        0      0  1 ] ]
```

and if you want "half" of that rotation, it's equally valid to use sin and cos of 0.05, or of π+0.05; either one will work out to the above matrix when you double them. (The fundamental theorem of algebra tells us that in general $z^n = w$ has n distinct solutions for a given n and w, and multiplying by a complex number is a subset of the transformations we're interested in here.)

So to get a smoothly interpolable representation, we can't simply use a structure of 12 real numbers; we need something that preserves more information about the functional relationship between those numbers and the parameter, or parameters, we'd like to interpolate.

So how can we get there from a relatively small and usable set of primitives with reasonably tractable computation?

This extrusion operator relies on a terribly general notion of "path". I mean, PostScript's "path" was already very general: it can be discontinuous and either open or closed, but at least it was just a one-dimensional manifold of two-dimensional points. This notion of "path" seems to be a one-dimensional manifold of twelve-dimensional transformation matrices:

```
[ [ x0  x1  x2  x3  ]
  [ x4  x5  x6  x7  ]
  [ x8  x9  x10 x11 ]
  [ 0   0   0   1   ] ]
```

where the twelve variables inside the matrix are some kind of piecewise-continuous real functions of one real variable, t. In some sense you can see the [x3 x7 x11] column as specifying a path in the more traditional sense, some kind of infinite sequence of points in 3-space, along which a point transformed from the origin [0 0 0 1] could swoop and soar as t changes; the other 9 variables simply describe how a thing moving along the path gets stretched, skewed, rotated, and perhaps reflected.

If we make these twelve variables, instead, functions of *two* parameters, we get a parametric surface, which we can sample as finely as we desire.

Linear real functions

First, and most basic, let's consider the linear real function $y = mt + b$. You can compose this from two one-dimensional manifolds in a variety of different ways; let's say:

```
M(m) = mt + 0
B(b) = 0t + b
```

These two *could* be composed with one another, but it is more useful to be able to add them (pointwise), because that lets us span the whole mt + b space. So the function $\pi x + \pi/2$ can be written as $M(\pi) + B(\pi/2)$. These functions are closed under addition, but not composition or multiplication.

If we consider transforming the interval [0, 1] through these functions, we can see that they can be used to represent arbitrary closed intervals on the number line; in this interpretation, the function above represents the interval $[\pi/2, 3\pi/2]$, for example. Of course there are an infinite variety of other possible interpretations,

but this is the one I will choose.

Frankly, though, this is a little silly; instead of M and B, I will use a single function of two real arguments:

```
I(x0, x1) = (x1-x0)t + x0
```

which I will write as:

```
x0...x1
```

Elementary paths

So let's consider the general 3-D linear transform that changes over time. This is capable of scaling (nonuniform scaling and reflection), skewing, rotation, and translation.

If we have rotations around two axes, we can get rotation around the third by composing them; if we have rotations around all three axes, we can get scaling along arbitrary dimensions by composing 90° rotations with scaling along a single dimension; and skewing happens if you use a non-90° rotation and then scale along a single dimension. Finally, composing arbitrary rotations with translations along a single dimension will give you arbitrary translations.

So we have four elementary "paths", in the sense of path I described earlier — not just a point that varies with some parameter, but a general linear 3-D transform that varies over time:

```
X(v) = [ [ 1    0      0    0  ]
         [ 0  cos v  -sin v  0  ]
         [ 0  sin v   cos v  0  ]
         [ 0    0      0    1  ] ]

Y(v) = [ [ cos v  0  -sin v  0  ]
         [   0    1    0    0  ]
         [ sin v  0   cos v  0  ]
         [   0    0    0    1  ] ]

TX(v) = [ [ 1 0 0 v ]
          [ 0 1 0 0 ]
          [ 0 0 1 0 ]
          [ 0 0 0 1 ] ]

SX(v) = [ [ v 0 0 0 ]
          [ 0 1 0 0 ]
          [ 0 0 1 0 ]
          [ 0 0 0 1 ] ]
```

So what is this v?

You could think of e.g. T as being the entire X-axis, or X as being infinite rotation around the X-axis, in the sense that for any real value of v, they represent a particular real 4×4 matrix that translates the origin to that point on the X-axis or rotates to that angle around the X-axis. And if you take some interval of real numbers [v0, v1] and transform it through T or X, you get some interval of translations along the X-axis or rotations around it.

In particular, if we draw our v from the linear real functions

described in the previous section, then as t varies from 0 to 1, these elementary paths will interpolate continuously and smoothly over some interval.

We can multiply these matrices to compose the 3-D operations they represent. This means we're multiplying these individual elements, which may be linear functions of t or transcendental functions of linear functions of t. This will produce, unfortunately, relatively general algebraic expressions as elements of the matrices, which in the worst case can grow exponentially in the number of elementary paths in the matrix product: each variable rotation introduces two new transcendental functions of a potentially new linear function. (It should be clear that Spirograph patterns can be computed this way easily, so you shouldn't expect much worst-case simplification.)

This is a bit of a disappointment, since I was hoping for something that would occupy bounded space, or hey! at least linear space, after an arbitrarily large number of operations. Not exponential space.

I console myself with the thought that the alternative representation is often a triangle mesh, and it's going to take a pretty big formula to approach the amount of space a triangle mesh uses.

Surfaces

Suppose we have a curve defined as above, some arbitrary composition of elementary curves, each parameterized by some interval of the real number line; now we would like to extrude it into a surface, which might enclose a volume. We can do this by transforming it through some other arbitrary curve, which can stretch and rotate and skew it as needed; all of this can be implemented simply by matrix concatenation, but with the variable t renamed to u in the extrusion path.

This gives us a 4×4 matrix, 12 of whose cells are arbitrary algebraic expressions in t and u, combined with numbers, sin, cos, addition, and multiplication; but we probably really only care about three of those cells, [x3 x7 x11], the ones that give us the coordinates to which the surface has carried the origin.

You might want to transform a surface by composing a path with it, but maybe not a path that depends on t or u; it's not clear what that would mean to me.

The algebra

So we have five basic elements:

- I(x0, x1: real): interval; written as x0...x1
- X(v: interval): path
- Y(v: interval): path
- TX(v: interval): path
- SX(v: interval): path

And two, or arguably three, ways of combining them:

- compose(a: path, b: path): path; written as a . b
- extrude(a: path, b: path): surface; extrudes a by moving it along b; written as a * b
- extrude(a: surface, b: path): surface (or solid); makes two copies of a, one at each end of b, one of them with its polarity flipped, and

connects them with a surface where the edge of a passed as it moved along b. Effectively this involves turning the edge of a into a path and extruding that path as it moves along b.

(Maybe instead: extrude or construct interval with ":", construct interval with "@", apply functions with simple concatenation or ".", compose with simple concatenation, ",", or ";", extrude with "/"? I'm ending up with lots of noise in my expressions. Maybe also use lowercase.)

Internally, this builds trees of the following structure:

• an expression is one of:

• a variable, t or u
• a real
• (a: expression) + (b: expression)
• (a: expression) * (b: expression)
• -(a: expression)
• cos (a: expression), or
• sin (a: expression).
• an interval is an expression.
• a path is 12 expressions.
• a surface is 3 expressions.

You could restrict this further, since e.g. the argument of a transcendental function here will always be a linear one, but that doesn't seem to offer much benefit yet.

Computations on the surfaces

The simplest and most obvious thing you might want to do is to sample the points on a surface to build a triangle mesh, which you can do by just evaluating the expressions for the surface points for different values of t and u. The range for t and u is predefined as [0, 1], so it's just a matter of figuring out how much and where to subdivide the range.

That's a bit of a problem, though. You'd maybe like to keep your sampling mesh more or less uniform in density, so as to avoid wasting triangles on areas without much detail. (On a sphere, for example, there will surely be a point where one or the other of the parameters doesn't give you any extra information.)

You can use bounding-box arithmetic ("interval arithmetic") to find 3-D bounding boxes of parts of the mesh, recursively bisecting the mesh until your 3-D bounding boxes are small enough that you are satisfied with your triangle size. Interval arithmetic is explained in a section below.

Additionally, for smooth shading (Gouraud or Phong), vertex normals can be helpful. You can calculate these by partially differentiating the surface vector with respect to t and u and normalizing the cross product.

Calculating the area of a surface may be useful for some purposes: maybe for paint coverage or heat loss or adsorption or something. It's reasonably feasible to approximate numerically with a simple double integral over the parameter space.

Interval arithmetic

This is a sort of abstract interpretation of numerical formulas,

introduced to me as an exercise in SICP, using [min, max] pairs to produce a conservative approximation of a function's range over some interval. For our case, we can use the following evaluation rules:

• a constant number k becomes [k, k]
• [a, b] + [c, d] = [a+c, b+d]
• -[a, b] = [-b, -a]
• [a, b] * [c, d] = [a*c*, *b*d] if all four are nonnegative; otherwise, use identities to transform the expression into the tractable all-nonnegative form above:

• -[a, b] * [c, d] = -([a, b] * [c, d]); apply if a and b are nonpositive
• [a, b] * -[c, d] = -([a, b] * [c, d]); apply if c and d are nonpositive
• -(-x) = x; apply if possible
• [a, b] = union([a, 0], [0, b]) if a and b have opposite signs; apply if possible
• a * union(b, c) = union(a * b, a * c); apply if possible. If this is applicable, you need to start again from the top because probably one of the new multiplication subexpressions contains a negative-going interval.
• union([a, b], [c, d]) = [min(a, c), max(b, d)] (remember, conservative approximation!)
• sin [a, b] may be:

• [sin(a), sin(b)] if a and b are in the same monotonically increasing interval of sin;
• [sin(b), sin(a)] if a and b are in the same monotonically decreasing interval of sin;
• [min(sin(a), sin(b)), 1] if a and b have a peak between them;
• [-1, max(sin(a), sin(b))] if a and b have a valley between them;
• [-1, 1] if a and b have both a peak and a valley between them.
• Analogously for cos [a, b].

These rules are conservative, but they are precise in the sense that if you subdivide a bounding box into smaller and smaller parts, eventually the bounding box of a union-free formula evaluated over it will have an arbitrarily small range.

Applying these evaluation rules to the three expressions that give the x, y, and z coordinates of points on a surface, with interval values for t and u, will give you a bounding box in x, y, and z for that subsection of the surface. If you use [0, 1], you get a bounding box for the whole surface.

I write these bounding boxes as [a, b] rather than a...b because the meaning of the second is slightly different, but confusingly similar: it's a motion from a to b that varies linearly with t, while [a, b] might oscillate wildly between a and b as t and u vary, perhaps not even reaching them.

Possible extensions

Given a third parametric parameter, you could animate an unchanging object along a path.

CSG — union, intersection, and especially subtraction — would dramatically increase the power of the system. It might be particularly tricky to compute precisely, though.

Piecewise functions would enable you to do things like splines.

You could imagine a sequence(a: path, b: path) operator (perhaps written "a; b") that evaluated as a(2t) on [0, ½) and then b(2(t-½)) or perhaps a(1)+b(2(t-½)) on [½, 1]. I think that the extrude-a-surface-into-a-solid functionality kind of depends on doing this anyway, since the resulting surface is a sort of composite of six different surfaces: the initial and final surface and the four surfaces produced by extruding the t=0, t=1, u=0, and u=1 edges. (Alternatively you could make a "surface" be an arbitrary set of parametric surfaces rather than just one. Also this is making me think I should read Wouter's CUBE engine.)

Piecewise functions potentially introduce a finite number of discontinuities, and many things might then need to get a list of discontinuities in an interval.

Being able to read in geometry from external sources (STL files, DXF files, Hershey fonts, SVG logos, TrueType fonts, heightfields) would enable a lot more stuff.

Being able to query geometrical results, KSeg-style or AutoCAD-style, or even just PostScript-style (flattenpath, pathbbox, charpath, pathforall), would make parametric modeling a lot more reasonable.

If you're just generating an STL file for 3-D printing, the shape is all you need. But for other purposes (displaying onscreen, using in a game, 3-D printing in color) you need a way of adding other attributes to your material.

Another thing you might want to do with a surface, besides look at it or extrude it into a solid, is to "shell" it — build up a solid around it of a given thickness. This is analogous to the "stroke" or "strokePath" operation in 2-D graphics. I call it "shell" because if you use it on a surface that encloses a solid volume, the result is a hollow shell of some thickness. (CATIA calls it "shell" too.)

The approximate inverse operations of "shell" are "fill" (remove hollow spaces inside of) and "erode" (known as "inset" in Inkscape; take a given thickness off the surface all the way around).

What are the equivalents of line dash patterns or halftone patterns in PostScript? Maybe honeycomb (and other) infill, or surfaces perforated to save plastic in 3-D printing? Maybe embossed surface texture?

I want to be able to query the volumes of enclosed or nearly-enclosed spaces for acoustic reasons.

In general it seems like repeating shapes could be useful. Like if you have an ellipsoid and you want to make sixteen of them evenly spaced.

Perlin noise or midpoint-displacement surface roughness might be useful things to have; but how do you add things like that in a clean way?

Topics

Topics to study in 2016

Kragen Javier Sitaker, 2016-10-27 (updated 2016-11-15) (37 minutes)

I have a fairly clear idea of what stuff I want to be studying right now, which has taken some time to congeal. There are 18 major themes: constraint satisfaction, mathematical optimization, logic, interval arithmetic, tensor or array computation, physical system simulation including finite element analysis, automated manufacturing or digital fabrication, incremental computation and caching, reproducibility, archival and publishing, real-time computation, anytime algorithms, probabilistic programming, neural networks, decentralization, self-replicating machinery, self-sustaining software systems, and physical security. These seemingly disparate themes actually have an underlying unity, which I will attempt to explain.

Constraint satisfaction

Constraint satisfaction systems search for a configuration that satisfies some set of boolean conditions. Sometimes the configuration may be continuously variable, like a point in $\mathbb{R}^n$. Other times it may be discrete, like a set of choices for a set of Boolean variables. It may have finite dimensionality (like $\mathbb{R}^n$), infinite dimensionality (like a function $\mathbb{R} \to \mathbb{R}$), or finite but unbounded dimensionality (like a sentence in some formal language). The conditions may be simple and tractable (things like "$x > 0$") or very hairy indeed ("md5(x) == 0xd41d8cd98f00b204e9800998ecf8427e", which looks computationally intractable but is actually fairly easy).

The parts of the configuration are sometimes called "design variables".

Different kinds of constraint satisfaction problems have different algorithms available to solve them. Most of classical algebra and integral and differential calculus consist of algorithms for solving particular kinds of constraints. You're given a constraint like $2x^2 + 4x = 30$ and your objective is to find an x that satisfies that constraint, traditionally by deducing logical consequences from that given constraint until it's trivial to compute the answer. For example, from that equation we can deduce $x^2 + 2x = 15$, then $x^2 + 2x - 15 = 0$, then $(x - 3)(x + 5) = 0$, and $x=3 \vee x=-5$.

You can think of a constraint as a generalization of the notion of a function — it's a function that can "run backwards", so to speak. This can significantly simplify the process of developing a piece of software, thus amplifying the intellectual abilities of its author. Often it is significantly easier to express attributes of the desired results of a program (consider the MD5 example above) than how to compute them. In some such cases, it's even possible for existing constraint-satisfaction software to find the results.

MiniKANREN is one of the more interesting pieces of constraint-satisfaction software becoming available today. The burgeoning field of efficient SAT and SMT solvers such as Z3 contains many others.

Aside from the rapid advances in the efficiency of satisfiability algorithms, larger and larger constraint-satisfaction problems are

coming within reach as memories and CPUs dramatically increase in power. A crude understanding of complexity theory might lead you to think that this won't matter much, because SAT is not known to be in P, so even a CPU that is 256 times faster will only allow you to expand the feasible problem size by 8 bits. But in fact the base of the exponential has been brought down far below 2 and continues to diminish.

Mathematical optimization

Mathematical optimization is the problem of finding a configuration that maximizes (or minimizes) some objective (or loss) function, subject to some feasibility constraints. The classic example — and indeed nearly the only one considered in many textbooks on mathematical optimization today — is linear programming, where the feasibility constraints are linear inequalities and the objective function is linear.

Categories of algorithms for mathematical optimization are called "metaheuristics", which is a somewhat misleading name but gives some clue of how broadly applicable they are. Random guessing, hill climbing, gradient descent, gradient descent with random restarts, and genetic algorithms are widely-used metaheuristics.

Constraint satisfaction and optimization

Optimization can be seen as a generalization of constraint satisfaction in a few different ways.

Clearly if you can find the optimal configuration among all feasible configurations, then you can find at least one feasible configuration, which is the constraint satisfaction problem. So, from this point of view, algorithms to solve optimization problems can solve constraint satisfaction problems too, but it might be easier to say

Also, though, you can conceptualize the constraints of a constraint-satisfaction problem as real-valued functions that are multiplied together, where those functions take the value 1 when the constraint is satisfied and 0 otherwise. Then, finding a maximum of the product is equivalent to finding a solution to the constraints. Many optimization algorithms will have an easier time if those factor functions are instead continuous and between 0 and 1 when the condition is false. And in fact this sort of thing is a common method (the "penalty method") for incorporating constraints into optimization problems for solvers that don't natively handle constraints. The very first constraint-satisfaction system for mechanical design, Sketchpad, apparently used gradient-descent to search for approximate solutions to geometrical constraint systems.

However, constraint satisfaction algorithms can also be used to solve optimization problems if you invoke them many times. If your objective function is y, you first seek a solution for $y > 0$. If successful, you seek a solution for $y > 1$; if successful, a solution for $y > 4$; if successful, a solution for $y > 8$; and so on until you fail, at which point you can approximate the optimum as closely as desired using binary search.

Logic

Logic is a very broad term, but in particular what I'm interested in are theorem provers and predicate logic, which can produce

machine-checkable proofs of mathematical propositions — whether propositions of general interest, such as (most famously) the four-color theorem, or propositions having to do with particular pieces of software.

There is a lot of work currently going on in this space, with systems like Coq, Agda, Idris, Isabelle-HOL, Z3, Mizar, and Epigram being actively developed, and systems like CompCert and seL4 being proved.

Much of the initial work on program correctness came from a desire to ensure that all programs were correct, for reasons having to do with the risk-averse nature of military bureaucracies, rather than any kind of rational cost-benefit analysis. But in practice, many programs do not even have a well-defined specification that they could conceivably fail to conform to, nor is it a good use of the irreplaceable million or less hours of a human being's life to write one. It's often adequate to try things and see what happens.

To me, proving programs correct is of interest for four reasons: compiler optimization, user interface innovations, software security, and embedded systems.

It's interesting for compiler optimization because if an unoptimized program can be proven equivalent to the original program, the correctness of the optimizer or optimizers is a moot point, and indeed some kind of random search over all feasible programs may be adequate (as Sorav Bansal proposed in his thesis, *Peephole Superoptimization*).

(Note that "optimization" here is used in a sense unrelated to the "Mathematical optimization" section above.)

It's interesting for user interface innovations because the interactive process of creating a machine-checkable proof with a proof assistant like Coq is very different from the traditional batch-job-inspired workflow of writing a program in a text editor in a compiled language and then feeding it to a compiler. I don't have enough experience with this process yet to figure out whether it has innovations that can be applied more widely to the process of creating things using computers.

It's interesting for software security because trying things and seeing what happens is never adequate to produce secure software, although it may be adequate to find security holes. Often when a security hole is detected in the field, it is already too late to prevent disastrous effects. So significant effort toward other ways to prevent them seems like it would be justified.

It's interesting for embedded systems for a somewhat similar reason.

If my production database server crashes, the production cluster can fail over to a secondary and save the logs and a core dump, and in the morning I can inspect the core dump and the logs to figure out what went wrong. Then I can fix the bug, recompile the database software, and upgrade across the cluster. The cost of the failure is low (assuming it wasn't a security hole), and it's highly inspectable and reparable.

By contrast, if the software on the Arduino controlling an autonomous toy car, the car may be run over by a truck, and possibly all the Arduino can do to make the error debuggable is to blink an LED. It doesn't have space for a core dump. The cost of the failure is

high, and it's poorly inspectable and reparable. (The situation is even worse if it's an automatic implantable defibrillator.)

So for embedded systems, it's highly desirable to detect errors at compile time rather than run time.

Logic and constraint satisfaction

In some sense, theorem-proving is merely an application of constraint satisfaction: you have some set of axioms and inference rules which you've previously decided are reasonable or at least consistent, and you're searching for a proof which starts from those axioms, is valid according to those inference rules, and ends with your desired conclusion. That's clearly a constraint-satisfaction problem.

But constraint satisfaction is also merely an application of theorem-proving. A constructive theorem prover, given the task of proving that some set of constraints C is satisfiable, will produce a constructive proof, which is a way to construct a configuration satisfying them. But even a non-constructive prover can be used, at least for a finite set of design variables; you repeatedly attempt to prove that a given design variable is in a given range.

This conjunction between constraint-satisfaction systems and logic systems is sometimes called "constraint logic programming".

From another angle, all of classical algebra (the equation-solving stuff, not the modern abstract algebra stuff that grew out of group theory) is the application of logic to constraint satisfaction. So you would think that that kind of theorem-proving logic could be applied to significantly ease more general kinds of constraint-satisfaction problems, where the constraints might include things like arbitrary lookup tables, conditionals, and iteration.

Logic and optimization

Both theorem-proving and mathematical optimization intersect with constraint satisfaction, but their intersections are sort of complementary. Theorem-proving is associated with (I'm being vague here) finite domains, algebraic structures, and various kinds of backtracking search. Mathematical optimization is associated with continuous and therefore infinite domains, analytic structures, and various kinds of iterative approximation.

But of course in day-to-day math, we often use theorem-proving techniques (often with pencil and paper) to transform problems of analysis into tractable forms and even to find closed-form solutions to them. So it seems likely that the two approaches are likely to be complementary in software as well.

Interval arithmetic

Interval arithmetic is a means to bound the approximation error on an approximate calculation by computing on a conservative approximation of the answer. The usual arithmetic operations are defined to operate over intervals of the real line in such a way that the result of the computation is a conservative approximation of the correct result. In general, narrower bounds on the input will result in narrower bounds on the output, although sometimes not as much as you'd hope.

Interval arithmetic has been most investigated as a way to bound the approximation error from floating-point roundoff by setting

different rounding modes, at the cost of considerable performance. However, it has a number of other applications; probably the best known is precise and efficient ray-tracing of implicit surfaces, a technique surveyed broadly in Jorge Eliécer Flórez Díaz's 2008 dissertation, but which has been developed continuously over the last quarter-century. In this application, it *improves* performance by avoiding redundant calculations that will produce the same result.

Intervals and constraints

You can use interval arithmetic to solve constraint satisfaction problems over continuous variables by recursive subdivision of the problem space, in the same way that I outlined earlier for solving optimization problems with a constraint solver, except that you may need to recurse into more than one subinterval. Treating arithmetic as constant-time, this approach is probably logarithmic-time in the required precision and exponential-time in the number of dimensions, and is therefore not practically applicable to problems of high dimensionality.

However, it is the only general algorithm I know of for continuous constraint satisfaction problems, not being limited to finite domains or to constraints of a particular form, such as linear systems or polynomials. (It only requires that the constraints be computable, which turns out to imply a particular kind of continuity.)

It's not completely trivial to get right, because strictly speaking, in the form I've described, it only guarantees that it wasn't able to prove that *no* points in the selected region satisfy the constraints, not that any of them do. For example, a simple implicit function grapher I wrote using interval arithmetic suggests that there may be zeroes anywhere the implicit function has a singularity, because the result of dividing by an interval containing zero is the interval $(-\infty,+\infty)$, which contains 0. But small regions around such a singularity will typically not contain any zeroes or even numbers close to zero.

There are lots of tweaks you can apply to this algorithm, and as with other kinds of recursive search algorithms, even fairly minor improvements can have enormous effects on efficiency, since they exponentially reduce the number of nodes searched. Some of the things I've been thinking about are priority queues of regions by size, a tighter interval representation that approximates results by =a linear function of input variables over a given input interval, three-way partitioning, and choosing partition locations to be closer to the part of the region most likely to be feasible.

Intervals and optimization

By the same token, you can use interval arithmetic to solve mathematical optimization problems by recursive subdivision.

Intervals and logic

I don't know how interval arithmetic combines with theorem proving.

Array computation

Array languages are not merely languages which support arrays, as almost every programming language does. They are languages in which arrays are first-class values that support aggregate arithmetic

and computational operations. They were born in the form of APL in 1957–1962, but in the 1980s spawned S, IDL, and MATLAB, and since then have spawned the languages R, J, K, A+, and Lush, and the Python library Numpy, the Perl library PDL, and most recently the neural-network-focused libraries TensorFlow and Theano.

Typically the arrays are of arbitrary dimensionality, and sometimes they may be called "tensors" or something else.

Fitting a program into the array-programming style requires a different kind of thinking than the mainstream record-oriented or object-oriented style which evolved, as far as I can tell, from 1950s business data processing on machines with a few kilobytes of RAM and tape drives. I find it very awkward.

There are two or three interesting advantages of this way of doing things that induce me to keep trying it:

• Efficient execution. Originally this was only an advantage for interpreted implementations like the original APLs, Numpy, or old versions of MATLAB; it just pulls the interpretation overhead out of your algorithm's inner loop. So C and FORTRAN programmers scoffed, since at best it got you from wasting 99% of your CPU to wasting 50% of it. Nowadays it's becoming an advantage even compared to C, C++, or Fortran (TensorFlow originated as a C++ library) because the array aggregate operations can be compiled efficiently onto a GPU. The array operations are generally explicitly parallel, rather than implicitly so as more imperative code usually is. (Roml Lefkowitz is the first person I remember making this observation.)
• Terser code. By making most loops implicit, arguably the code strips away much of the inessential implementation complexity and reveals the intention of the programmer more clearly. I feel like this can go either way, but this may be a function of my inexperience with the paradigm.
• More abstract and thus more flexible code. Typically the code `a*b` with an array language or library could mean (in Python notation) any of `[ai*bi for ai, bi in zip(a, b)]`, `[a*bi for bi in b]`, `[ai*b for ai in a]` just `a*b`, or even `[[aij*bij for aij, bij in zip(ai, bi)] for ai, bi in zip(a, b)]`, and which one depends on which of the two is a vector or array. This works to your advantage more often than you'd think; if you write a plotting function that takes arrays of X-coordinates, Y-coordinates, and point colors, it will generally just fall out that you can pass in a single color instead of an array of colors when that's what you want. This depends, however, on the detailed design of the aggregate operations provided by a particular language, and sometimes it fails in surprising ways. (And I'm not sure it applies at all to TensorFlow and Theano.)

I've been thinking for a while about how to clean up the semantics of array languages, and I think there's an interesting unification between array languages and logic languages waiting to be birthed.

One interesting possibility is being able to decide, separate from the specification of what is to be computed, that some of the axes of the arrays in question should be materialized in time rather than in space.

Arrays and constraints

One of the most common uses for libraries and languages like these

is in solving systems of simultaneous linear equations, which is the simplest kind of constraint satisfaction and often occurs as part of a larger constraint system (for example, a linear network within a circuit containing nonlinear elements); in Numpy, for example, to solve Ax = b for x, where A is a matrix and b is a vector, you use `numpy.linalg.solve(A, b)`. Scipy, a library built on Numpy, has a sparse-matrices package that can be deployed in the service of solving much larger sparse linear systems.

XXX explain here about logic and array languages

Arrays and optimization

One advantage of array formulations of optimization problems is that automatic differentiation — which numerically computes a derivative or gradient together with the output of a calculation — is easily added to an array library like those mentioned, and indeed TensorFlow and I think Theano provide this functionality natively.

Arrays and logic

Iverson got a Turing Award for inventing APL, and his lecture was entitled *Notation as a Tool of Thought*. He argued that APL was a more formally tractable notation for programs, to which we could more easily apply the principles of symbolic reasoning and manipulation. Certainly programs in APL and related languages are compact, which is a major advantage for symbolic manipulation by hand; could it similarly make formal manipulation more tractable?

Arrays and intervals

I don't think there is any particular synergy between arrays and intervals.

Physical system simulation

From the beginning, one of the most important uses for computers has been the numerical simulation of physical systems — whether wing flutter in Germany with Zuse's Z-3, artillery trajectories and H-bomb explosions with the ENIAC, the car-crash simulations that were the public rationale for the Tera MTA, or the optical systems that modern GPUs were developed to simulate.

I am just beginning to learn about finite element analysis, which is the most broadly applied method for physical system simulation.

Simulation and constraints

To the extent that you can formulate a simulation in a constraint-satisfaction-system-tractable fashion, you can use the constraint-satisfaction system to design a physical system that satisfies your constraints. But this is more useful in the context of optimization.

Simulation and optimization

The most famous results along these lines are topology optimization, which has been used to optimize stiffness, heat transfer, and so on, but which seems to me to still be in its infancy despite a third of a century of development. Still almost nobody is optimizing elastic bodies for specific stress-strain curves, for example, or searching for toolpaths that minimize the heat-affected zone around a kerf.

Simulation and logic

I dunno, I got nothing.

Simulation and intervals

One of the hairy problems with finite element simulation is that it's conducted on a discretized approximation of the actual physical system, not the continuous system itself. The discretization introduces errors into the simulation results, but how do you find out if those errors are big enough to be significant? You can use a finer discretization and see if the results changed much. If they do, you need to go finer. But if they don't, you don't know if you're just on a plateau and an even finer resolution would give you a big change, or not. I'm told this is an especially big problem with fluid dynamics, due to "vorticity, cavitation, backflow, all sorts of wonderful nonlinear effects" (@sigfig).

There has been work on bounding these errors using conservative error approximation techniques, but I don't know about them. The straightforward application of interval arithmetic is not enough.

To the point that it's feasible to bound these errors, it should be possible to speed up preliminary simulations enormously by only simulating them to very loose tolerances. This should speed up optimization through simulation greatly.

Simulation and arrays

Everybody uses arrays for numerical simulations. It's, like, what they're best at.

Automated manufacturing or digital fabrication

Fabrication is the making of physical objects. Digital fabrication is making them from a design in a computer; the most commonplace example is printing out a page on a printer, but it also includes things like integrated circuit fabrication and CNC milling. There's been a lot of excitement around 3-D printing (aka additive manufacturing) in the last few years, but that's only one kind of digital fabrication.

I think digital fabrication is poised to explode over the next few years for a variety of reasons: better sensors (producing better feedback), better design tools (including simulation), and the RepRap-derived community, including things like Thingiverse and OpenSCAD.

Manufacturing and constraints

Much of robotics has to do with finding a feasible plan for achieving an objective under known constraints. Toolpath planning for CNC is one example; a bad toolpath can do things like drive a cutting tool hard against a part of a workpiece it was not supposed to contact.

Even without robotics, of course, things like nesting for panel cutting are constraint problems.

Manufacturing and optimization

In those cases, often some plans are much better than others. A manual panel-cutting plan that requires you to reset the fence on the panel saw three times is better than one that requires you to set it

twelve times, unless it results in using three sheets of plywood where one would do. A toolpath can take more or less time and spend more or less time far from the optimal feed rate.

We often break down design problems into parts separated by abstraction layers in order to make each part tractable on its own. Indeed, this is one of the major forces driving additive manufacturing in industry in the last few years — it allows you to fabricate a much wider range of shapes, increasing the freedom of the design stage. But it seems likely that mathematical optimization should be able to optimize across many such stages at once, and this is likely to provide one or more orders of magnitude improvement in cost/benefit ratio.

Manufacturing and logic

I dunno.

Manufacturing and intervals

Likewise.

Manufacturing and arrays

Likewise.

Manufacturing and simulation

Simulation enables optimization in manufacturing. It also makes it possible to reduce the number of prototypes enormously, conceivably eliminating the necessity for mass production entirely.

Self-sustaining software systems

Self-sustaining systems are those that can compile themselves from source code. PyPy, Squeak, Oberon, and metacompiled Forth are my usual examples.

I'm interested in self-sustaining systems for a few different reasons.

XXX movethislater

Incremental computation and caching

Incremental computation is adjusting the output of a computation to reflect a change in its input in some way that is cheaper than simply recomputing the output from the changed input. A large fraction of our existing software infrastructure is occupied with incremental computation, because it's common to get several orders of magnitude performance improvement from it.

Caching or memoization is one strategy for incremental computation; this works by breaking a computation into deterministic parts with identifiable inputs in a way such that some of the computations have no change to their inputs, and therefore are guaranteed to have the same output as before; then you simply reuse the previously computed result for that part of the computation, thus limiting the work only to part whose input has changed.

Examples include executable files (cached compiler output), font bitmaps (cached font rasterizer output), window contents backing store (cached paint output), the DOM (arguably, cached parser output), any in-memory data structure computed from the contents of a file, and any number of lower-level pieces of information in your code — basically almost anything for which you write a function with the word "update" in its name.

This is a very broad topic, including perhaps the majority of

software in existence, because it's common for caching of the results of computations to convert exponential-time algorithms to linear-time or even constant-time algorithms, or to knock off four or more orders of magnitude from constant factors. A unified system for caching that was capable of doing a reasonable job at solving these problems could increase the power per line of code of software enormously.

(It's commonly said that the only two hard problems in software are naming things and caching (or cache invalidation).)

The few interesting projects on this front are Apache Spark, redo, and Self-Adjusting Computation.

Caching isn't the only way to do incremental computation. It's also possible, though less common, to compute a new output from the old output and some kind of description of the change. This is typically the approach taken with RDBMS indices — when an indexed value is updated in the underlying table, rather than recomputing the index or part of the index from the new table contents, the database adds a new index entry and deletes or invalidates the old one. This kind of incrementalization requires the processing being incrementalized to preserve some kind of homomorphism between the input and output such that the change to the output can be computed from the change to the input.

Incrementality and constraints

There are some incremental constraint systems, which can propagate an incremental change to constraints more efficiently than recomputing a constraint from scratch. Constraint systems implemented by optimization will typically have this property by default. Combinatorial search (for constraints over finite domains) also likely benefits from reusing parts or most of a previously-valid solution.

The other angle on this would be using a constraint system to make a computation incremental. This might work but I'm not sure how.

Incrementality and optimization

As mentioned above, optimization systems that work by iterative approximation will tend to have incremental properties — a small change in the objective function or the constraints will get a big performance boost from having an existing approximately-correct solution.

The other angle on this would be using a mathematical optimization system to carry out general incremental computation. In particular, caching systems face difficult choices about which cached computations to throw away and which to keep; space is always finite, relevant information about what to keep is always incomplete, and good choices can speed things up by many orders of magnitude overbad choices.

Incrementality and logic

Tabled resolution is the standard way to cache logical conclusions in Prolog-derived logic-programming systems. I don't know anything about it.

In more general theorem-proving systems, which I also don't know anything about, I have the impression that tactic choice and proofs can be reused from one run to the next of the proof assistant, saving

the time of expensive searches; as with combinatorial search in constraint systems, this should be a huge win.

Using theorem-provers to support strategies for incremental recomputation also seems feasible, both in informing cache eviction strategies (if you can prove that something will be used again in the near future, it may be a poor choice to evict from cache) and in proving that an incrementalization strategy doesn't introduce bugs. This seems necessary if homomorphism-based incrementalization (as opposed to caching) is to be correct, and would be sufficient to introduce it automatically where applicable.

Incrementality and intervals

One of the more interesting ideas in Flórez Díaz's dissertation is that interval arithmetic can avoid computation and thus dramatically accelerate ray-tracing, especially of animations. If you determine that for $0 \le x \le 0.1$, $0.2 \le y \le 0.3$, $0 \le t \le 5$, some conditional yields false, then you don't have to recompute anything in that x-y area guarded by that conditional when t increases from 1 to 2.

Applying this idea more generally, it may be possible in some cases to compute a description of how *much* some input would have to change in order to change an output.

And, of course, the recursive refinement algorithm I described earlier for using interval arithmetic to solve optimization and constraint problems could in theory benefit enormously from incrementalization, since each subdivision of the problem space changes only one input from the previous iteration.

Incrementality and array languages

Incremental recomputation of arrays or parts of arrays seems like an interesting idea. The array language K implements some version of it. Some relevant considerations:

• Array languages expose many computations to the runtime at a higher level than other languages, which could potentially make it more feasible to do even homomorphism-based incrementalization automatically. The dependencies between values in different arrays are generally more regular than the dependencies generated by arbitrary code, so it should be feasible to keep more of them around.
• If the unit of caching and recomputation is individual elements rather than entire arrays, tracking valid/invalid bits for large arrays should be much cheaper than tracking them for numerous random small objects.

Incrementality and simulation

Incrementalizing simulation of physical systems would be super awesome; if you could get a 1000× speedup on second and subsequent simulations of slight variations of a design by reusing data computed in the first iteration, you could try 1000× as many variations. Straightforward caching won't get you there, because typically any change to any part of the system design immediately reverberates through the system, creating tiny, possibly insignificant changes everywhere. I think this might be possible, on the other hand, under some circumstances, with the interval-arithmetic ideas I mentioned earlier — you may not care to find the precise stiffness of each design, for example, but only to optimize the stiffness by finding

the stiffest variant.

Incrementality and manufacturing

I don't think I have anything to say here that isn't said above.

Incrementality and self-sustaining systems

A coherent incremental recomputation system may be a key to getting a self-sustaining system down to a reasonable size.

XXX movethislater

Reproducibility

Reproducibility is the foundation of science. Science is what can be demonstrated. Irreproducible results cannot be demonstrated; you must take it on faith that the person reporting them is reporting them accurately, and so there is no tendency toward self-correction over time.

We have a limited kind of reproducibility with our current software; often enough, you can download a piece of software, possibly recompile it with current libraries, and have it work on your computer. But often this requires some work to get it running, and even if it appears to work, you don't know if it works the same way it worked on other people's computers in the past. So if they reported some results with it, and you get conflicting results, you don't know why that is.

Nix and Guix are efforts to make software more reproducible by nailing down all its dependencies, and they enhance the reproducibility situation significantly. However, the bootstrap kernel for either of them is fairly large, and they rely on unreliable third-party websites to host the tarballs of the dependencies, with the result that this reproducibility still has big holes in it.

I think we can do considerably better, keeping the initial bootstrap kernel very small indeed — small enough that it can be reimplemented from scratch on a new platform in a few hours.

Reproducibility and constraints

Constraint-solving systems are inherently nondeterministic in a way that imperative systems are not; the solutions they find, and indeed whether they find a solution at all even when one exists, may vary depending on details of their search strategies. This potentially poses problems for reproducibility, since the effects of changes may be hard to track.

For a weak form of reproducibility, this is not a problem, since we can compile a theoretically nondeterministic constraint-solving system into a deterministic imperative form before using it. But a stronger form of reproducibility, in which we can reason about what effects will be produced by a given change, will be frustrated by it, because the effects of altering constraint-solving things will be mostly by virtue of changing which of many possible acceptable solutions will be found first.

Reproducibility and optimization

Optimization is even more sensitively dependent on random nonsense and thus that much more difficult to achieve reproducibility in. Worse, being able to take advantage of things learned during previous optimization sessions may be really crucial to making

optimization-based computation feasible, in a way that it isn't for traditional programming paradigms. To some extent, it may be possible to paper over this using caching approaches to incrementalization.

However, to the extent that optimization can be used as a way

Topics

Simplified computing, down to the level of mining raw materials

Kragen Javier Sitaker, 2015-09-03 (22 minutes)

I've been thinking for some years about building a personal computing environment up from zero: bootstrapping using an existing computer and existing fabrication technology, but quickly getting into self-sustaining technology, along the lines of the VPRI STEPS program or colorForth/GreenArrays, but also along the lines of Open Source Ecology.

So what is needed to, say, send an instant message to someone? What's the smallest amount of code you could write to fully describe the design of this system?

• A **font** to display the message in.
• A **display** to show the font on, and whatever software is needed to talk to it.
• A **keyboard** (or other input device, e.g. a touchscreen), and whatever software is needed to talk to it.
• An **editor** that responds to the keystrokes by editing some text that you can later send.
• A **network interface** to connect to the internet, and whatever software is needed to talk to it.
• Enough of **TCP/IP** to speak whatever messaging protocol you're using.
• A **user interface shell** to allow you to choose between doing instant messaging and other things.
• A **CPU and RAM** to run all this software on.
• A **filesystem** to store your instant message logs, buddy list, driver code, font, editor and other UI code, and so on in.
• A **durable storage device** to persist your filesystem to when the machine is turned off. This might be core, Flash, MRAM, disk, cassette tape, barcodes, battery-backed SRAM, or cuneiform, but you do need *something*.
• A **low-level programming language** with a compiler to write the drivers in. Ideally this language is powerful enough to write the compiler in too, as well as the UI code, filesystem, and so on, but there really aren't any existing languages that are good at this. I think ML probably comes closest, but ML memory usage is somewhat unpredictable (even if you use MLKit regions instead of a garbage collector) which means that you can't write code for which failure is not an option in ML.
• The **IM client** itself.
• **Cryptography** to keep the IMs safe.
• Probably a **semiconductor fabrication system** to make the CPU and RAM.
• Materials processing machines for **smelting, measuring, and purifying** the materials needed for the CPU, RAM, keyboard, display, and the materials processing machines themselves, starting from raw materials.
• **Digging machines** for extracting the raw materials.

• **Fabrication robots** for making the display and keyboard and for assembling the computers.
• **Energy-harvesting machinery** for deriving energy from the environment to run all the other machinery.
• **Waste recycling systems** to keep the materials processing and especially the semiconductor fabrication from poisoning people, animals, and plants.

Font

So far, despite various experiments, the simplest way I've found to design a readable font is as a fixed-width pixel grid. Here's a minimal implementation of ASCII text rendering with an included 4×6 font in 1K of DHTML, including the font itself as a PNG:

```
<body bgcolor="black" onload="r()">
<a href="http://canonical.org/~kragen/sw/dofonts.html">cf</a>
<img src="
iVBORw0KGgoAAAANSUhEUgAAAEAAAAAkAQMAAAADwq7RAAAAAXNSR0IArs4c6QAAAAZQTFRFAAAA
////pdmf3QAAAQxJREFUGNMtjlFqg0AYhIcQikhJryBFgoiHWGQR2QfJIaQPfegZRCQEkZxBZCny
I9sr+LBIkZBDBPFBQs7QNXTgh2Hg+2ewpZh7Ehtsf26S89XYmvHHbgu8ebVJLJPcDlyyGACDhwzg
ifpK4VTomjYnnoSY3LuOkLmQ4XvbIGFI7kF84U6Bpyxzqb7Oyi9dELUn6Z8J6rcbp+C8oKS2lFQS
PvX1qMTyT63Sw0I0VwzUCiK3qDFqobhZhrIRD1m4QNElQs4BzPyndmCZ4xReD+yHNOptoWG3tHRB
QsZOkfQF4WX4CJW/yJV4Rf8kmfkRHzB8m9KJ16CLIBFEwJgLFa2lx2ovmsmYfMU2Fv4AgeVrl8Qw
sFoAAAAASUVORK5CYII=
" id="font" />
<canvas width="512" height="512" id="c"></canvas>
<script>
d = document
function r() {
 t = d.body.innerHTML
 x = 0
 y = 0
 for(i in t) {
  c = t.charCodeAt(i)
  if(32<=c && c<128) {
   fi = c-32
   d.getElementById('c').getContext('2d')
   .drawImage(d.images[0],fi%16*4,~~(fi/16)*6,4,6,x,y,4,6)
   x += 4
  }

  if(x>508 || c===10) {
   x = 0
   y += 6
   if(y>=512) break
  }
 }
}
</script>
```

Other experiments I've tried include Hershey-like line fonts, spline-based fonts, fonts scanned from paper, and so on, but I haven't been able to get any of these implementations down to a kilobyte, or even nearly so. You could maybe argue that I'm unreasonably leveraging bitblit and PNG-decompression here, but I don't think

those actually make a whole lot of difference.

Display

The current standard is an LED-backlit LCD with diffraction gratings to give color to RGB subpixels. This may not be the least demanding possible way to display pictures under computer control; it seems likely that some kind of cathode-ray tube may be simpler to construct.

Spark gaps and spinning mirrors

Perhaps the electronically-controlled display with the least stringent demands on metallurgy and the like would be a small array of tiny carbon arc lamps or spark plugs controlled at near-megahertz frequencies, viewed through a system of spinning mirrors like the ones used in supermarket barcode scanners in order to scan the image of the arc across your field of view, and color-filtered both to eliminate harmful ultraviolet light and to improve contrast.

Suppose you use just an X-axis scanning mirror, with one spark gap per scan line, and you want minimally 200 scan lines (25 lines of readable 8-scan-line text or 33 lines with the less-readable 4×6 font above). A 16-faceted mirror spinning on an air bearing at 32000 rpm, as Michelson used in his 1877–1931 experiments to measure the speed of light, scans at 8528 frames per second. We only really need about 24 frames per second to be readable, even though that's kind of flickery, and you could do that with 1.5 revolutions per second of a 16-faceted mirror, 90 rpm. This is an easily achievable rotational speed even if you're turning the mirror by hand.

Square pixels at 200 scan lines and a 4:3 aspect ratio would be 266 pixels per line; 16:9 would be 355 pixels per line; a 5×8 font at the traditional 80 columns would require 400 pixels per line. 266 pixels in 1/24 of a second is 6384 pixels per second per spark gap, or 157µs of cool-down time; 400 pixels is 104µs (9600Hz). Spark times like these are easily achievable. Neon glow discharge lamps can easily reach this régime, and they last for decades of continuous use and don't pollute the air, unlike carbon-air arcs.

Edgerton was achieving 500ns air-gap spark times in the 1950s by using a thin layer of air as the plasma medium and cooling it with a quartz heatsink, but I think much higher speeds are achievable if the spark just needs to be *visible* rather than illuminating its surroundings brightly enough for photography.

A 500ns spark could potentially paint 2 million pixels in a second, or 33,333 pixels in the 16⅔ms of a 60fps frame, not quite enough for a text display by itself. An array of 20 500ns spark gaps, assuming you could fire them all continuously, could thus give you a ⅔Mpix display at 60fps, nearly the 1024×768 that we're used to.

At the currently popular 16:9 aspect ratio, this would be about 1075×620, so you'd need a vertical or Y scanning mirror that scanned at 31×60Hz, or 1860Hz. If it had 16 facets, it would need to spin at 116¼Hz, or 6975 rpm, which is totally feasible using gears and ball bearings (not even air bearings), but a little bit tricky. Again, though, this is about four times slower than Michelson was spinning mirrors nearly a century ago.

If, instead of scanning an entire column of 20 spark gaps across the visible area in 20-pixel-high bands, you divided the visible area into

20 horizontal bands, each illuminated by a single spark gap, you could have 20 facets in your visual field at once. This allows you to use more facets on your mirror and makes the scan lines more horizontal, but requires careful alignment of the mirror's phase with your viewpoint. It would, however, allow you to rotate your Y-scanning mirror at only 31×60Hz÷16÷20 = 5.8125Hz or 348¾rpm.

Vector displays

The first graphical user interface was Sketchpad, implemented at the end of the 1950s by Ivan Sutherland on the TX-2 at Lincoln Lab. The TX-2 had an attached CRT in which the CPU independently controlled the X and Y coordinates of the electron beam, so it could draw simple shapes on the screen at high resolution, even though it was only capable of drawing a few tens of thousands of points per second. Such "vector displays" survived until the 1980s, and CAD systems of the 1970s were built on "storage tubes" that could maintain the vector image, once drawn, for up to a few minutes, allowing very detailed line drawings.

You could do the same thing with a single spark gap and two mirrors, but this requires controlling the mirrors independently and actively, not just scanning them at a constant speed like a raster display. But it would permit a useful display from a single spark gap modulated even down at kilohertz speeds.

Coil-driven mirrors

Laser shows scan lasers across the screen with mirrors controlled by galvanometers or "galvos": electric coils that magnetically push against a spring to bring the mirror to the desired position. Typical galvos are capable of "20 kpps", twenty thousand points per second, although the definition of this is kind of dodgy.

The Harmonic Series was a project by Brazilian artists Luisa Pereira and Manuela Donoso which substituted regular dynamic speakers (electric coils that magnetically push against a screen) for laser-show galvos, getting bandwidths of up to several kilohertz for displaying audio Lissajous patterns, just as Lissajous himself had done with the "Lissajous apparatus" of mirrors on tuning forks in 1857.

You could imagine a resonant mirror scanning system that scans in a Lissajous pattern rather than the sawtooth pattern we're used to from raster-scan devices; you get higher resolutions around the edges, lower resolutions in the center, and a less regular scanning pattern that the computer can compensate for. However, I think the Pythagorean sum of the scanning frequencies still limits your resolution, equivalent to the scan frequency of the X-scanning mirror in a traditional raster setup. So with the 20 500ns spark gaps at ⅔Mpix at 60Hz described above, you could get by with oscillation frequencies possibly as low as 42Hz or so.

Lissajous rods

A particularly simple way to achieve these Lissajous oscillations is by using a sprung mass that is sprung with two different spring constants in X and Y; for example, a thin rod, fixed at one end, with a rectangular cross-section. (There is a Lissajous Rods exhibit in the Exploratorium demonstrating this; the Lissajous patterns described in the air by the shiny ends of the rods are quite conspicuous.)

Other high-speed light sources: vacuum tubes, LEDs, and lasers

All of the above describes different contortions to get around the possible slowness of a spark-gap light source, which is itself much faster than an incandescent light. But even normal vacuum tubes can turn on and off in the submicrosecond regime; you could use a vacuum tube with a phosphor-coated anode to modulate light output with deep-submicrosecond times. There are a number of electron-excited phosphors with single-digit nanosecond decay times, hundreds of times faster than an air-gap flash, including CdS:In and ZnO:Ga.

(If you're using a vector display to draw relatively static images, a long phosphorescence time might be an advantage rather than a disadvantage.)

Vacuum tubes and exotic phosphors, though, may be hard to fabricate; and if you can fabricate them, it may be more practical to just fabricate a CRT rather than some kind of electromechanical display.

Another possibility, since we're already taking for granted semiconductor fabrication, is LEDs. These are semiconductor diodes made of exotic large-bandgap semiconductors, and, with carefully designed driver circuitry, even regular LEDs can have response times measured in single-digit nanoseconds. And LEDs are much, much more durable than spark gaps. They were first reported in 1927, using point-contact diodes with low-purity silicon carbide, but red LEDs were first published in 1962 in gallium arsenide phosphide; they became commercially available in quantity in 1968. Other red and green LED chemistries include AlGaAs, AlGaInP, and GaP; InGaN and GaN additionally provide green, blue, and violet LEDs, but these were not possible to manufacture until 1994, and the discovery won a Nobel Prize in physics.

Once you, hypothetically, can make LEDs, making semiconductor lasers is relatively straightforward. This has the advantage that you can scan them over a large surface using mirrors to produce a high-resolution projection display.

Also, lasers in general have the ability to produce shorter light pulses than other light sources: once a population inversion forms, stimulated emission can deplete it, and so a wave packet generated by a mirrorless laser can be very short indeed, the so-called "femtosecond laser", when the laser has a large tunable bandwidth. You should be able to trigger the generation of a pulse by dumping a bunch more energy into a barely-subcritical laser, which is more or less how erbium-doped fiber amplifiers work. (This is my vague understanding of this shit; don't bet on it.) So it should be possible to

One possible way to rapidly scan temporally-modulated LED light over an area would be to use a transparent Lissajous rod elastically vibrating; it could be made of, for example, polished glass or acrylic, using total internal reflection to guide the light to its vibrating tip.

High-speed light modulation: LCDs

LCDs do not emit light, but they can switch on and off in tens of nanoseconds using current techniques. I don't know if LCDs are less demanding to fabricate than LEDs, but I suspect so, even though they were developed later.

Slow displays: in the limit you get a printer

Above, I've mentioned the benefits of slowness a couple of times: if your image lasts a long time, you can use even very slow methods for producing it. The Tektronix 4014 storage tube was one example: used for high-resolution CAD drawings despite being connected to the computer over a low-speed serial link. But even a regular long-persistence phosphor would work reasonably well, without the storage-tube aspect; in fact, if you're using mirrors to steer a beam of light rather than an electron beam, you could draw the image on a phosphor screen of, for example, copper-activated zinc sulfide, which is sphalerite/wurtzite, the main ore of zinc in nature.

But you can go slower still. A lot of Unix was developed on Teletypes running at 110 baud, which worked out to 11 characters per second; a 4×6 dot-matrix printer at this speed needs to run its 6 hammers at 44Hz, which is quite reasonable. And if you go vector instead of raster, you get a pen plotter.

Smelting, measuring, and purifying elements

Both current and prehistoric metallurgy depends mostly on iron, aluminum, copper, zinc, tin, and lead; nuclear energy production depends on uranium or thorium. Semiconductor fabrication also depends on silicon, plus trace amounts of dopants, such as boron and arsenic, as well as processing chemicals such as hydrofluoric acid and hydrogen peroxide. Refractory materials are essential to many processes listed.

Small amounts of iron, lead, and especially copper are found "native" on Earth, while aluminum, zinc, tin, and thorium are always found in compounds. Nevertheless, for bootstrapping, it's probably necessary to be able to refine these elements from soil.

Earth's crust is about 28% silicon., and it's in almost all soils, so anywhere you go, you have silicon. Although you can smelt metallurgical-grade silicon from silica in an arc furnace with carbon at 1900°, smelting it for semiconductors is hell, since you need to get impurities down to <1ppb in order to control its semiconducting properties, using either CVD from trichlorosilane or repeated zone melting. Even less-demanding photovoltaic silicon needs <1ppm impurities.

Earth's crust is about 8% aluminum, which, too, is everywhere; but it's difficult to smelt because aluminum dioxide (sapphire) is so stable. The Hall-Héroult process, orders of magnitude cheaper than the alternative smelting processes, revolutionized the production of aluminum. This involves dissolving corundum (aluminum oxide) in cryolite (sodium aluminum fluoride) and calcium fluoride at 980° in order to be able to electrolyze it.

Earth's crust is about 5% iron. People have been smelting iron from hematite and magnetite using charcoal in bloomery furnaces for millennia, but blast furnaces are much more efficient and produce much better iron.

Earth's crust is about 60 ppm zinc, mostly as sphalerite, which can be purified by froth flotation; roasting sphalerite in air at at least 690° produces SO_2 and ZnO, and the ZnO can be reduced to Zn either with C (which was done accidentally for centuries, producing Zn

vapor) or by electrowinning from water-soluble $ZnSO_4$ (low-current-density process: 10% H_2SO_4, aluminum cathodes, 275–325 A/m², 30–35°). Historically this smelting was done in hermetic clay retorts to prevent the zinc vapor from escaping.

Earth's crust is about 50 ppm copper.

Earth's crust is about 14 ppm lead.

Earth's crust is about 6 ppm thorium, but you don't need very much thorium, because burning thorium in a nuclear reactor produces about 79 TJ/kg of energy.

Earth's crust is about 2 ppm tin, which makes it hard to extract.

Plasma pyrolysis and oxide distillation

One possible answer to the problem of extracting various elements from the complex mix of mineral compounds in soil is, rather than using individual reactions customized for each separate ore, simply vaporize the whole shebang into a plasma, mix with oxygen, and then distill out the various oxides. Even tungsten, tantalum, rhenium, and osmium boil below 6000°, and every other element below 5000°; everything melts far below this temperature, with tantalum hafnium carbide melting at 3942°.

So, if you can heat soil up to 6000°, you can probably get them to mix with a large volume of preheated oxygen and oxidize them fully. You should be able to start the process with a plasma arc torch (regular plasma arc welding torches reach 28000°), then continue heating with inductively-coupled plasma heating. Various companies are currently developing lower-temperature variants of this process under the name "plasma arc recycling" or "plasma gasification"; Westinghouse's process runs at only 3000°.

As an alternative to inductive heating of the plasma, you can illuminate it; well-ionized plasmas absorb light fairly well. Sunlight can be focused on the plasma using non-imaging optics, which have been demonstrated to reach 84000 suns, 59 megawatts per square meter.

Another alternative means of heating the plasma after initial ionization might include mixing carbon or hydrogen fuel into the feedstock. In theory, hydrogen ought to be able to raise the temperature by about 2800° in the limit of 100% hydrogen.

The mix of oxygen and fairly inert simple elemental oxides coming out of the torch then needs to be distilled. Each oxide has its own characteristic condensation temperature, so you should be able to condense thoria in one chamber, silicon dioxide in another, aluminum oxide in another, zinc oxide in a fourth, and so on. You probably need to do this under low pressure to keep the plasma from cooling down too much before depositing its condensates.

How much energy does this need? Heat capacity of plasmas is complicated but generally around 100 kJ/(kmol·K) in the temperature range we're considering, so heating by 6000 K should require about 600MJ/kmol. If we figure that the majority of the plasma is oxygen, then each mole is 16 grams, so we need up to about 40MJ/kg to heat the plasma, some of which we may be able to recover as we cool it back down.

Supposing we mix oxygen with the vaporized soil in a 4:1 ratio by mass, we're spending 200 MJ per kg of soil, or 33 TJ per kg of thorium, assuming that our soil is only averagely rich in thorium.

That's less than half of the energy that thorium will produce in a reactor, which still leaves us precariously net positive, but suggests that some monazite deposits (about 10% thorium, i.e. 100 000 ppm, five orders of magnitude above average) would go a long way. (It's also possible that the modal thorium concentration is well below the mean, which would mean that most points on the globe had thorium concentrations too low for purification by this method to be a net energy producer.)

Topics

2025 manufacturing and economics scenario

Kragen Javier Sitaker, 2014-04-24 (24 minutes)

It's 2025. I want a bicycle to use to visit my friend Yésica in La Plata, a city near Buenos Aires, where I live. I type "recumbent touring bicycle" into a model search engine and get a page full of pictures of different models. I click on one of them, a model called the Matagallos, and adjust the parameters on the resulting page: my inseam is 95 centimeters, I weigh 100 kilograms, I will be carrying 50kg of luggage with me on the trip, I prefer dull brick red to the default bright blue, and because I'm pretty strong and not in a hurry for this trip, I prefer lower material cost to lower weight. My computer consults databases of parts available nearby to fulfill the model's requirements; only 1.8mm spokes of dubious metallurgical quality are conveniently in stock near where I live in Argentina, so it ups the spoke count on the wheels from 36 to 48 to compensate.

I review the model from different sides on my screen, using a finite-element simulation to simulate how it will travel over bumps and potholes, and find a problem: because this model was originally developed for lighter riders, when the rider-weight parameter is set all the way up to 100 kilograms, it happens to work out with an unreasonably low ground clearance — I'll be bumping my ass on the ground. I call my friend Violeta for help, sending her a link to the model. She shows me how to adjust the formula for one of the strut lengths to add a ground-clearance constraint, and then we test the modified model to ensure that the results are reasonable across the range of parameters.

Having validated her modification, I thank her and write up a couple of sentences explaining the problem and the solution, and publish the modification so that other people can try it out. After making a couple of cosmetic modifications (I want the struts all to look like bones, and I don't want to display the model name, because the police near here are nicknamed "gallos", and I don't want them to feel threatened), I look at a list of prices and delivery times from manufacturing services in my neighborhood. I choose one that's 400 meters away and offers me US$102 with a delivery date of three hours in the future.

I click the "submit" button, confirm the transaction on a handheld smart card, and switch back to chatting with Violeta about the vacation she's planning to take with her husband next month, visiting Arecibo in Puerto Rico.

Three and a half hours later, my doorbell rings, and there's a truck with my bicycle, assembled by robots from whatever materials and parts happened to be on hand, using more or less traditional processes of drawing, forging, cutting, drilling, lashing, gluing, nailing, and welding. By chance, a substantial part of this particular bicycle is made of wood recovered from discarded pallets. I key in the delivery PIN to get the truck to release the bicycle, lift the bicycle off the truck, and press the "accept" button on the truck, so that it can move on to its next stop. Mine are the first human hands that have touched it.

My ride to La Plata is uneventful — the full-body fairing keeps the rain off me, and the speakers have amazing bass — and the bicycle mostly performs well, but I notice a grinding noise coming from the front wheel. As I pedal, I search for "grinding noise matagallos front wheel" in a search engine, and it turns out somebody else had the same problem last year. They figured out that the problem was an over-tightened wheel bearing, and explain how to check, so I stop to check. Turns out my Matagallos got assembled with a cartridge bearing, so it can't be tightened or loosened at all, but it seems to have some kind of sand in it. I do a search for "sand bearing" and the number of the manufacturing service, and it turns out that several other people using them have had the same problem, in products as diverse as drills, fans, and food processors, although I'm the only one to run into it with a bicycle.

Once I arrive in La Plata, I stop by a manufacturing service that I'd contracted with on the way; they have a new front wheel ready and waiting, and in a couple of minutes install it on the bicycle, taking the defective wheel as partial payment of US$5. Since I've reported the bearing grit problem to the service near my house, they accepted the blame and paid the remainder of the cost, US$3. I leave them a

good review, explaining the problem and resolution, and tag the other people who had reported grit problems so they'll see it.

At Yésica's house, she and I pick a couple of interesting recipes off a recipe blog, tone down the spiciness for her Argentine palate, and submit it to the robot chef around the corner, paying it US$1 for the service. Half an hour later, a bell rings to notify us that the food is ready; we stroll around the corner to pick it up and bring it back home.

Waking up in Yésica's bed in the morning, I'm pleasantly surprised to find that my (Violeta's) improvement has been accepted into the mainline Matagallos model, and I'm now listed as one of its contributors. The maintainer accepts my argument that the credit belongs to her, not me, and replaces my name with hers. I publish a short experience report, describing the performance of the Matagallos on the pothole-laden road from Buenos Aires to La Plata.

But now it's time for me to get to work. A few blocks away, a couple I don't know are seeking a mediator for an unspecified marital conflict related to their autistic child, and as I happened to be in town and with good reviews for similar mediations from some of my friends, they requested that I meet with them. I consult their reputation — they seem to be trustworthy — so I accept their proposal of a US$800 fee for an hour and bike on over to their house.

There are some things that robots still can't do, you see, and those things have gotten more expensive.

Before I return home by bus, I sell my new bike as scrap to a local manufacturing service for US$97.

(The above scenario is based on some things Richard Stallman wrote, but it isn't an attempt to represent his point of view. Violeta is a real person, but Yésica is not.)

XXX why did you sell the bicycle?

Commons-based peer production

In the last couple of decades, we've seen a really remarkable shift in production from the market economy into ad-hoc volunteer networks engaging in something Yochai Benkler calls "commons-based peer production": people contributing resources on a voluntary basis to a common product that everyone can use without restriction. Examples of such products are Wikipedia; Stack Overflow's collection of technical questions and answers; the worldwide repository of data available through BitTorrent; the worldwide tourist lodging system CouchSurfing; the FreeBSD system that is the basis of Apple's MacOS and iOS operating systems; the WebKit browser engine that drives Chrome, Safari, the Android browser, and the iOS browser (known as Mobile Safari); and arguably the collection of videos on YouTube, the social graph and other information on Facebook, the corpus of photos on Flickr, and the contents of other such "user-generated content" sites.

I say "arguably" because these proprietary sites aren't governed by the same liberal commons-protecting licenses that enable the communities I listed earlier, so I'm not sure if they fit Benkler's definition, but much of the same dynamics seem to apply. Whether you upload zero, one, or a thousand photos to Flickr, you can still look at the commons of all the photos on the site, produced by your peers. (I'm embarrassed to admit I haven't managed to make it through Benkler's book yet, so I'm not totally sure where he draws the line around "commons-based peer production".)

This is a remarkable phenomenon. One after another, we see market-economic projects collapsing when they have to compete with commons-based peer production, much as the Soviet Union eventually collapsed economically when its system of production could not compete with the capitalist bloc.

Apple had 20 years of experience as one of the most innovative and often financially successful producers of computer software, employing many of the best programmers in the world, when it decided to cancel its in-house next-generation operating system project and switch to NeXTStep, much of which is free software. Apple's own internal software development project simply could not compete with the commons-based peer production of FreeBSD — although so far its user interface seems to be holding its own against Ubuntu.

Wikipedia, similarly, completely crushed Encyclopedia Britannica, Microsoft Encarta, and the other lesser-known commercial encyclopedias.

Never before has capitalism found itself face-to-face with a system it can't compete with economically. The only feasible response, taken by Wikia, YouTube, Facebook, and so on, seems to be to try to build a "commons" that in reality is privately owned by you, so everybody else does the work of constructing the economically valuable product, while you take home the profits. People seem to be starting to realize that this is not such a hot idea, but it's going to be a while before we can fix the problem. But that's not what I want to talk about now.

You can also argue that this trend is unjust, because it takes economic rewards away from creative people or from the majority of the population, which is also a really interesting discussion to have, but that's also not the discussion I want to have in this post. Bookmark the idea for later, because I want to talk about it in a later post.

How far can commons-based peer production go?

It's reasonable to ask how far this trend of replacing market economic production with commons-based peer production can go, assuming it's not already played out. Wikipedia, for example, has replaced the encyclopedia market, but there still seem to be lots of other books out there. A lot of the fiction books can maybe be replaced by fanfic sites; Wikipedia and the WikiBooks project of the Wikimedia foundation are working to replace other nonfiction books, as is Stack Exchange (the family of sites including Stack Overflow); BitTorrent and YouTube are replacing TV to some extent; Android is largely replacing Microsoft Windows; but what about the implications for things that aren't pure information? Physical things like recumbent bicycles, chairs, highways, swimming pools, clothing, food, housing, clean water, headphones, and antibiotics?

Let's investigate how physical things are created

Well, those physical things are all produced by the following unreasonably general process:

- **raw materials** are
- combined and modified by **energy** under the control of
- **existing durable physical things**, which are guided by
- a **design** or plan which is chosen ahead of time,

• then adapted according to **feedback** from the production process, to a greater or lesser extent; we could call this "improvisation".

In the case of a bicycle, for example, the raw materials include wheels, a frame, a seat, and a chain; the energy moves the parts into place relative to each other and tightens the nuts needed to hold them together; the existing durable physical things probably include some wrenches and a chain tool; the design consists mostly of the mechanic's choice of which of these raw materials to use; and feedback is used, perhaps, to discover that the chain chosen was too short and needs to be lengthened by adding some links, or to replace the desired double-walled aluminum front wheel with the single-walled steel front wheel that happens to be in stock.

But each of these "raw" materials is itself a physical object, and is produced in the same way. The frame is made from raw materials of metal tubing, consumable welding electrodes, paint, headset bearings, brakes, the pedal assembly (I forget what this is called in English; in Spanish it's the "caja pedalera"); the energy needed to combine them includes the energy to cut the tubing, melt the welding rods, aerosolize the paint, and tighten the pedal assembly and headset bearings; the existing durable physical things include the paint sprayer, the arc welding buzzbox, and some more wrenches; the design describes, among other things, exactly how long each piece of tubing should be, which alloy it should consist of, the shape of the cuts to make, and how much metal to deposit in welding; and the adaptation in response to feedback may involve, for example, re-welding spots on the weld that melted through the first time.

If you carry out the recursion all the way, you find that the raw materials for a bicycle consist of most of the 89 or so natural elements: the tires are made of carbon, nitrogen, oxygen, hydrogen, and sulfur, for example, while a steel tube may be made of iron, carbon, chromium, molybdenum, nickel, and silicon, and an acrylic fairing is just carbon, hydrogen, and oxygen. The existing durable physical things necessary to process these raw elements into a bicycle currently include blast furnaces, zone melting facilities, hydrocarbon cracking towers, other chemical process plants (including their catalysts), welders, cutting tools, and so on, each of which uses energy in its own way, although largely to heat things up, cool them down, or move them around. (We can imagine machinery that used a narrower range of processes and of materials, but making an acceptable bicycle entirely out of them seems like it will be a big challenge.) The design covers everything from the crystal structure of the wheel hub to the overall bicycle weight.

A sandwich is made of the same raw materials, although in somewhat different proportions, but much of the design is provided by genetic material, and much of the energy is provided by the sun through photosynthesis. Plant growth is really interesting because, although it's a fully automatic process, the final design of the plant can be extremely varied as a result of adaptation to its environment ("feedback" or "improvisation" in the framework above), both in its physical form and in its chemical content. The same variety of beans can taste quite different depending on how much you water them and what kind of soil they grow in.

An interesting feature of looking at things this way is that almost none of these raw materials are typically depleted. Helium can be

depleted by escaping the atmosphere, and a few of the natural elements (like radium and radon) decay into other natural elements, but they're being constantly produced by thorium and uranium, which are decaying but not at an appreciable rate. And since bicycles and sandwiches are made of the same elements, you can recycle bicycles into sandwiches, or sandwiches into bicycles.

This framework even applies to, for example, manufacturing a copy of Firefox. The raw materials are a small area of ferromagnetic oxide on a spinning disk platter; a small amount of energy is used to magnetize them in a particular sequence, under the control of the disk head and, eventually, a whole computer; the design is the bits of Firefox that have been downloaded over the web and reside in memory, awaiting being written to the disk; and there's a certain amount of feedback involved in choosing which disk sectors to write. This may sound absurd, but the process is essentially the same as printing a book on your laser printer, except that the raw materials are a little harder to recycle.

Traditionally, in a manufactured product, the raw materials were drawn from a relatively small set of natural materials and refined alloys, perhaps a few hundred for most products; the energy was provided by human muscles; the existing durable physical things were general-purpose tools; the design was produced, or perhaps remembered, by an master craftsman; and feedback was also exercised by the craftsman.

In this framework, then, the First Industrial Revolution was, more or less, simply a result of replacing the source of energy with a more inexpensive source of energy: coal producing steam. The Second Industrial Revolution, centered on mass production, was more or less a matter of reducing the need for feedback to a minimum, largely provided by the introduction of many specialized existing durable physical things ("tools") but also by improved measurements that reduced the variation between objects, and a much wider variety of raw materials. The Japanese dominance of manufacturing was largely because they reincorporated ubiquitous feedback, but at a different scale — the workers worked to use feedback to optimize the process, not to fix an individual produced artifact.

So of these five factors of production, all five were originally very expensive; making #2 cheaper created the British Empire; shifting some cost from #5 to #3 created the Second Industrial Revolution and the American Century; and making #5 cheaper instead of eliminating it created the richest country in the world, at least if you measure by life expectancy.

So, within this framework, here's what I predict will happen to the cost of manufacturing in the next few years:

• Raw materials, in the sense of elements, are becoming progressively more available, because people keep mining them and not putting the scarce ones back in the ground. Even when they do put scarce ones back in the ground — for example, the indium in indium tin oxide coatings on discarded LCDs, or the cadmium in nickel-cadmium batteries — they mostly put them into specific small locations, where they should be recoverable later.
• Energy is already really cheap, as it has been for a century, but it's going to get a lot cheaper soon, because as of 2013, photovoltaic

power plants have gotten cheaper than fossil-fuel plants in sunny parts of the world. Without subsidies. And a substantial part of their remaining cost comes from the energy needed to make them. Almost none of it comes from the cost of the elemental raw materials, primarily silicon and aluminum.
• Existing durable physical things have their cost determined by the cost of manufacturing them, so they fall out of the analysis.
• Designing things is still really hard, but designs are just informational goods, and it turns out that commons-based peer production completely kicks the ass of the market at that. Sometimes.

• Eliminating feedback was the core of the Second Industrial Revolution because feedback required human attention. But computers are feedback machines, and now they're really cheap! Industrial process control was a major use of minicomputers starting in the 1970s, and computer-based automation (computer-based in order to handle feedback) has been the key to the dramatic increase in manufacturing productivity per manufacturing worker over the last 30 years. We can predict that this trend will continue, and as a result many of our Fordist instincts about how to optimize productivity will turn out to be wrong. (I say this despite the fact that Toffler-style "mass customization" has been a highly-hyped business failure for nearly 30 years now.)

So it seems likely that the main cost of manufacturing in the near future will be raw materials and energy, with some sort of multiplier for the durable tooling needed; but only insofar as we can sustain commons-based peer production for manufacturable designs.

Commons-based peer production will take over most of manufacturing from markets

That is, within a few years, by far the most expensive part of manufacturing a bicycle, or nearly anything else we manufacture, will be designing it. And that design effort will probably be mostly done through commons-based peer production, not through a market, because market economies are generally unable to reach a level of productivity sufficient to compete with commons-based peer production in areas where we have figured out how to do commons-based peer production.

What could prevent this?

The distributional equality issues I mentioned in passing previously are a big problem. They could keep us locked in an outdated 20th-century industrial production model.

There's the usual problem of diffusion of complex innovations: manufacturing services like the ones I envision above are only useful if there's a sufficient volume of automatically-manufacturable parametric models, people who know how to tweak them for their needs, and software for creating, modifying, and evaluating them; and each of these other three things depends in the same way on the other three. So diffusion could take some time.

Environmental collapse, or other forms of societal collapse, could cut off long-distance communication, which would effectively decimate the people you could collaborate on models with. It could

also cut off the long-distance trade needed for the mass-produced "processed" materials that can't yet be produced locally, such as integrated circuits.

Robotic automated custom manufacturing could remain dramatically more expensive than mass production. We see this, for example, in integrated circuit manufacturing: IC customization is certainly possible, using lasers, electron-beam etching, or "programming" (i.e. blowing fuses), and in fact I think most ICs undergo at least one of these processes, but the vast majority of the IC manufacturing process is done with X-ray lithography, the most mass of all mass-production processes, which commonly produces trillions of identical parts (such as Flash memory cells) in a single operation. And semiconductor manufacturing is one of the most highly automated of all current industrial processes.

Purely competitive concerns could result in people sabotaging each other's productive capacities. Israel might have a strong motive to destroy any general-purpose manufacturing service in Palestianian-controlled parts of the West Bank, for example, while the Palestinians might have a strong motive to do the same for general-purpose manufacturing services in Israeli settlements in the West Bank. Taken to the extreme, we could see assassinations of researchers, like those carried out by the Mossad against Iranian scientists, those carried out by the Unabomber, and the famous assassination of Gerald Bull.

Topics

Ultralight tunnel personal rapid transit

Kragen Javier Sitaker, 2019-03-11 (15 minutes)

I thought I'd written about this somewhere, but I can't find it, so I'm writing it again.

Adequate personal rapid transit would give low-density cities most of the benefits of high-density cities. In particular, what I have in mind are smooth 1.5-meter-diameter dirt-floored tunnels every 50 meters with pods — really electrified multi-person hybrids of horizontal elevators and bicycles — traveling through them at up to 28 m/s (100 kph or 60 mph in archaic units) under the control of a centralized just-in-time route planning system. Unlike a typical subway system, the pods move out of the track to park at a station; there are stations every 50 meters along each tunnel.

The pods are powered by overhead low-voltage DC power cables — 40 volts, for example, so it's almost impossible for a human to receive a dangerous shock. At 28 amps, 40 volts supplies 1120 watts (1½ horsepower in archaic units). A 400-kg fully-loaded pod traveling at 28 m/s would have 154 kJ of kinetic energy, which would take 2.3 minutes to build up at 1120 watts, so trips of 5 minutes or more would reach full speed. A 4.6-minute one-way trip, then, would be 3.9 km; a half-hour trip (supposing a maximum of 28 m/s) would be 47 km, so a city organized around such a transport grid could grow to a diameter of 47 km before any trip was over half an hour, not counting the half-minute on each end to walk the 35 meters or less to the nearest station.

Tunnel construction

The tunnels, being underground, don't create any noise or pedestrian hazards on the surface. If they're built by trench-and-cover methods, they would be super cheap; the trench could be only 2 meters deep, or 5 meters to add crossover tunnels and in cases where it's considered important to keep the surface free of noise and vibration. It's necessary to have smaller drainage tunnels below so that seeping rainwater doesn't fill up the tunnels. In areas where the water table may reach the level of the tunnels, despite the best attempts of drainage tunnels and sump pumps to lower it, bare dirt isn't a viable option, and the tunnel floors would need to be at least dirt grouted with lime, or something like that. Also, unless the tunnels were actually waterproofed, they would be unusable when the tunnel was actually below the water table.

Energy efficiency

With regenerative braking, only the energy required to overcome friction would be lost; the energy to accelerate the pod is recovered when it slows down, using the same motor/generator that accelerated it.

Tires and bearing capacities

Low-pressure 25 PSI (170 kPa) bicycle tires, like those used on

beach-cruiser bicycles, are gentle enough on the ground to not crumble most soils. (Soil bearing capacities generally used in civil engineering range from 75 to 600 kPa, with "compact sand" being 300 kPa.) Supporting 400 kg on such tires would require a ground-contact patch of 0.023 m² or 230 cm² (35 in² in archaic units). If the tires are 5 cm wide, then you need a bit more than 46 cm of tire contact length, which is easily achievable with two or more tires.

User experience

When you enter a transport station, you specify your destination station, as with Uber Cab. If a pod is already available at that station, you just get in; otherwise, a pod is dispatched to you, and after you get in, it's dispatched to your destination. Dispatching works by planning out a route using available tunnel segments, then reserving that route so that no other pod will use it until you're done, to prevent collisions even if something goes wrong. Since there are plenty of tunnels (20 per kilometer, even without any stacking!), and the trips are so quick, it should be easy to build enough capacity that it's practical to reserve the whole circuit end-to-end before starting.

The pods are long and low; users recline in a nearly-horizontal position, one behind the other, or with children in the laps of their caretakers. This allows the tunnels to be only 1½ meters tall. The pods aren't collective vehicles like buses and trains — you can be alone, or share it with your family or friends going to the same destination. So you don't need to make any stops in the middle to pick up or drop off other people, the way a bus or elevator does.

Comparisons with existing systems

By contrast, Buenos Aires, where I live now, has the best public transport of any city I've ever lived in. To take a bus downtown, I need to walk three blocks (6 minutes) to the bus stop, wait an average of 5 minutes (but sometimes 10) for the bus, wait on the bus anywhere from 30 to 60 minutes to go the 4 km to downtown. So it takes, let's say, 51 minutes on average, with an enormous variance. In the same amount of time, the tunnel PRT system could take you anywhere within 82 kilometers — not just downtown, but uptown too.

Consider a five-kilometer segment of a six-lane highway of cars as a point of comparison. It can accommodate about 1.5 cars per second in each direction, with about 1.2 people per car, for a total of 1.8 people per second per direction. When there's no traffic jam, they can travel at 28 m/s, so their latency is about three minutes, although that doesn't count the time to get on and off the highway and park. It's about 20 meters wide, noisy, impassable to pedestrians and wildlife, requires constant attention from drivers, and needs a couple of meters of concrete and stones to support it.

What does the tunnel PRT system need to reach 1.8 people per second per direction over a three-minute route with end-to-end reservations?

If we pessimistically assume only one person per pod, it needs 100 tunnels in each direction. If the tunnels are stacked three deep (5 meters), that's a width of about 1.7 km. That pod will weigh no more than 150 kg, rather than 400 kg, so it can reach full speed in 53 seconds, and needs another 53 seconds to decelerate. In that case, it

will be able to travel 3.6 km in those three minutes, which is a bit shorter than the 5-km car highway, though not much. (Traveling the full 5 km requires another 50 seconds and consequently another 28 tunnels each way to reach the same throughput.)

If we instead have fully loaded 400 kg pods carrying four people each, it only needs 25 tunnels in each direction, or 830 meters of width, but in 1½ minutes, the heavier pods only reach 22 m/s, so they only cover 2.0 km in three minutes. Traveling the full 5 km requires 5½ minutes, and thus 150 tunnels in each direction. XXX calculation error

However, generally six-lane highways are few and far between in the urban landscape; parallel highways are usually on the order of 10 km apart. So this tunnel PRT system could actually support highway-like volumes and rapidity of travel everywhere, but without the noise, pollution, traffic jams, barriers, and parking problems, and possibly without even the danger, though safety issues are notoriously hard to predict.

As another comparison, consider the B line of the Buenos Aires Subte (our subway), which is 11.8 km long, which it traverses in 30 minutes, and the most popular line. It carried 6'035'183 passengers in January 2019, the latest month for which statistics are available, 195000 per day on average. About one-twenty-fifth of the monthly total across all lines is an average workday — the weekends are much less trafficked. As of 2017, 25% of the total across all lines is between 16:00 and 19:00, when the trains are jam-packed full. The city has adopted a goal of a train every 3 minutes during rush hours, although of course any irregularity in service can stretch the delay to many times that; with such short headways, even in the absence of other problems, bus clumping is likely.

If we apply the older systemwide averages to the current numbers for the B line, we estimate 240'000 passengers per workday, of which 60'000 are between 16:00 and 19:00, an average of 20'000 per hour, 5.6 per second. The use of the B line is concentrated toward downtown, and especially in the first seven stations, but we can pessimistically assume that usage is uniformly distributed over the whole line, giving an average trip length of one-third the length of the total, 3.9 km and 10 minutes. The tunnel PRT system outlined here would need 4.6 minutes for a 3.9-km trip, thus running twice as fast. Let's pessimistically assume that all the passengers are going in the same direction (which is close to the truth at rush hour), requiring an equal number of empty return pods.

If you have four passengers per pod, you would need 760 PRT tunnels to provide this level of service. If these were spread out over a kilometer at 50-meter intervals, this would require tunnels stacked 38 levels deep, 57 meters! This is about twice as deep as the deepest part of the existing B line; the original trench-and-cover tunnel reached its deepest point at 17 m, but the newer stations are deeper.

(Alternatively, you could just put the tunnels next to each other with only the occasional space in between for a station, which would spread them out over a kilometer and a half; or go for in-between measures, like three-tunnel-deep stacks a couple of meters apart.)

At the other extreme, if you have one passenger per pod so that it weighs only 150 kg, the trip would be quicker (3.2 minutes), but you'd need 3000 tunnels.

Because the trips are so much shorter and the pods are not linked up into inflexible trains, the total number of pod seats needed to deliver this level of service is about an order of magnitude lower than the number of subway train seats; pods totaling only a few thousand seats would be sufficient to provide hundreds of thousands of rides per day.

Scaling laws

The scaling laws for invisible thoroughfares like those described above are perfect for constant-speed vehicles: to go twice as far with end-to-end reservations, you need twice as many tunnels, which means the people are traveling distributed over a thoroughfare area twice as wide, so the thoroughfare has a constant aspect ratio regardless of its length. However, by including occasional stops — say, every 10 minutes for single-person pods, or every 30 minutes for fully-loaded pods — the amount of traffic can scale arbitrarily high. This more or less corresponds to what we would call a "headway" of 10 or 30 minutes in a traditional mass-transit system.

Power distribution

Putting 28 amps on the 40-volt DC cables is probably best achieved with a higher-voltage distribution system (insulated, perhaps buried, and perhaps AC) stepped down and rectified if necessary with a buried transformer every few hundred meters; if you try to do the distribution directly over kilometers at 40 volts, the cable losses will be high.

Acceleration and jerk

For comfort as well as limiting floor and tire wear, you'd probably want to limit accelerations to, say, half a gee, but with such small motors, this is only a problem at the very beginning and end of the trip; once you're up to 1.5 m/s (310 ms at half a gee), the 1120-watt power limit is more tightly constraining than the half-a-gee limit, even at 150 kg. (At 400 kg, the transition happens even sooner, at 570 mm/s, 116 milliseconds into the trip.)

You'd want to tilt for curved tunnels; this is easily accommodated, even for widely variable speeds, if the tunnels are circular in cross-section and the pod wheels are placed so as to touch this circle. The tightness of the curves at a given speed is limited by the centripetal acceleration — both for passenger comfort and to limit tire loading, you want to limit that. Centripetal acceleration is just v^2/r, so if you limit it to half a gee as well, the curved tunnels for changing direction at maximum speed should have a radius of curvature of 160 meters or more ($(28\ m/s)^2/(4.9\ m/s^2)$); to limit jerk, the tunnel curvature should change smoothly into and out of the curve.

Alternatively, you could use tighter-radius curves to change direction at lower speeds; for example, at the beginning and end of the trip.

Effects on social structure

Since such a system would enormously reduce the cost of living in lower-density "suburbs", we can expect that people would opt to move to areas of lower populational density, though perhaps not

proportionally. Because it would eliminate the usefulness of cars within its ambit, walking would become much safer and more pleasant. We can guess that this would motivate people to optimize their neighborhoods for walkability, making them more parklike, and perhaps to increase the distances between potentially dangerous industrial facilities and personal spaces (such as houses) or public spaces (such as shops).

Topics

Bayesian and Gricean programming

Kragen Javier Sitaker, 2015-08-20 (3 minutes)

Bayesian logic on graphical models

Cox's theorem (or at least Jaynes' interpretation thereof) says that Bayes' theorem is the only generalization of classical (or, maybe, intuitionistic?) propositional logic to continuous truth-values that satisfies some relatively straightforward conditions. What would it look like to program with Bayesian rather than Aristotelian logic? Is this the same as probabilistic programming or different?

I suspect that there's an interesting connection here with interval arithmetic and non-monotonic reasoning. Consider the propositions, from a HN discussion yesterday, "Birds can fly, but penguins can't [and penguins are birds], but Harry the Rocket Penguin can [and he is a penguin]." If we interpret each of these propositions probabilistically rather than by Aristotelian logic, things work out correctly: if we are told Hermione is a penguin, we might try to compute P(X can fly) from P(Hermione is a penguin)=0.99, but then we must deduce whether penguins can fly. In the absence of P(X can fly | X is a penguin) = 0.99 from the explicit statement above, and for the 1% chance that it's false, we are reduced to attempting to compute P(X can fly | X is a penguin) from P(X can fly | X is a bird) = 0.99 and P(X is a bird | X is a penguin) = 0.99, which tells us that penguins can fly with about 50:1 odds. But fortunately we have this other piece of evidence, which is that we've been *told* penguins can't fly. (Given that, do we conclude that Hermione is 99.02% likely to be flightless, or more like 67%?)

(A Gricean programming language, where the programmer is assumed to have followed the maxims of quantity and relation and not specified unnecessary information, might be interesting... but that's very speculative!)

Generalizing this a bit, you could consider the value of *any* variable to have some probability distribution; the traditional engineering and science use of "X±Y" confidence bounds is merely a simplification of the distribution, a simplification which (one interpretation of) interval arithmetic deals with. But in reality we have, say, a Gaussian lump in the probability distribution centered on X with width 2Y (or Y/12 or whatever), and near-zero probability elsewhere. This may pose problems for dynamic typing, since in the general case, taking as a specific example a computed latency for an internet connection, we would need to consider not only whether the latency was between 20 and 25 milliseconds, but also whether the latency was 20+3i milliseconds, 5 meters, a turnip in a field in Laos, or the Stoic conception of virtue as expressed by Marcus Aurelius. (Sound Bayesian reasoning prohibits you from excluding possibilities a priori, since doing otherwise leads to pathological reasoning breakdowns. Unfortunately, it would seem to require that we *start* from what appear to be pathological reasoning breakdowns.)

Practically speaking, we probably need to approximate the

posterior distribution with some kind of sampling, as with particle filters, but perhaps in many cases we can sample *intervals* of the probability space rather than *points* in it.

Topics

Disk oscilloscope

Kragen Javier Sitaker, 2017-04-10 (updated 2017-06-20) (3 minutes)

High-speed analog-to-digital conversion is very difficult, so a common way to make high-speed oscilloscopes is to store the signal after the trigger in an analog form for long enough to analyze it at leisure. This is of course a description of how entirely-analog oscilloscopes work, but for example a multi-gigahertz digital oscilloscope vendor in the early 1990s told me his scope stored the data in an internal CRT — a sort of analog version of the Williams tube — until it had time to analyze it.

Oscilloscopes are a particularly tricky kind of thing to build out of random electronic crap you find in the junk pile because that crap typically doesn't include any ADCs over 10Msps (some scanners contain 6Msps ADCs), and you really need at least 40Msps or 60Msps for an entry-level 20MHz oscilloscope. (Keep in mind that an analog 20MHz oscilloscope isn't incapable of viewing signals above 20MHz; that's just its 3dB attenuation frequency. Sub-nanosecond signals will probably be phase-shifted and badly attenuated but they'll still be there.)

So it occurred to me that maybe a discarded obsolete hard disk could bridge this gap. Suppose we're talking about a current 15krpm Seagate Cheetah with its 204MB/s data transfer rate, which (if it's on one head) implies that the waveform at the disk surface includes significant, reliably recoverable components at up to 800MHz. The disk is rotating at 250Hz. Once a waveform is recorded, it is then repeated at the read head over and over again, every 4 milliseconds, until either the head is moved to another track or a new waveform is recorded. We have 4 milliseconds of waveform recorded, which would amount to 3.3 million cycles of the highest frequencies recorded and could thus be fully digitized in about a second using the 6Msps scanner ADCs I mentioned earlier; but in a much more typical case, you only care about a few hundred or thousand sample points after the trigger event. And you can digitize a few of them on every revolution until you have them all digitized.

Considering digitizing 1000 points at 40 million samples per second, well, that's 25 microseconds, which is 150 samples at 6 megasamples per second. You can digitize points #0, #7, #14, #21, and so on on the first revolution of the disk; #1, #8, #15, #22, etc., on the second; and in this way after 7 revolutions of the disk (28 ms) you have digitized the whole event. Digitizing at higher effective sample rates, or using a slower ADC, would require proportionally more revolutions.

Even ordinary disks (5400 rpm, 50 MB/s) should still be capable of functioning effectively in this role.

A problem with this pretty picture is that disks are not really designed for analog signal integrity, and so the signal may be corrupted with noise and subject to hard-to-characterize nonlinearities. And of course you need to degauss the track before recording small analog signals on it.

(See also files TV oscilloscope (p. 1253), VCR oscilloscope (p. 213) , Laser printer oscilloscope (p. 449), and CCD oscilloscope (p. 1861).)

Topics

Writing hypertext is still a pain

Kragen Javier Sitaker, 2016-02-18 (6 minutes)

Writing hypertext is still a pain, and an autocomplete dropdown box from a statistically-ranked history of things you're likely to want to link to would improve the situation dramatically.

Writing hypertext is still a pain, 26 years into the WWW project and 71 years after the invention of hypertext. Typically I navigate to a page, use ^L^C to copy the link, alt-tab to another window where I'm writing something in Emacs (God help me if it's in another tab of the same browser) and type something like `<a href="^Y">some title</a>`. If I want some kind of indication of what the link is, like an image preview or something, I'm a bit out of luck in most environments, although at least Fecebutt and Twatter are helpful there if the page has OpenGraph metadata.

Consider this item from org-mode's help file, though:

> org-store-link is an interactive compiled Lisp function in 'org.el'.
> It is bound to C-c l. (org-store-link ARG)
> Store an org-link to the current location. This link is added to 'org-stored-links' and can later be inserted into an org-buffer with C-c C-l.

That sounds a bit saner, doesn't it? It's kind of like what Flock was trying to do in 2005 with their "shelf", where you would put things you were later going to link to or embed, later drag-and-dropping them onto any random textarea.

Even Vannevar Bush's 1945 UI design for linking in the "Memex" he proposes in "As We May Think," the very first proposed hypertext system, is far less cumbersome than what I usually use at present:

> All this is conventional, except for the projection forward of present-day mechanisms and gadgetry. It affords an immediate step, however, to associative indexing, the basic idea of which is a provision whereby any item may be caused at will to select immediately and automatically another. This is the essential feature of the memex. The process of tying two items together is the important thing.
> When the user is building a trail, he names it, inserts the name in his code book, and taps it out on his keyboard. Before him are the two items to be joined, projected onto adjacent viewing positions. At the bottom of each there are a number of blank code spaces, and a pointer is set to indicate one of these on each item. The user taps a single key, and the items are permanently joined. ...
> Thereafter, at any time, when one of these items is in view, the other can be instantly recalled merely by tapping a button below the corresponding code space. Moreover, when numerous items have been thus joined together to form a trail, they can be reviewed in turn, rapidly or slowly, by deflecting a lever like that used for turning the pages of a book. It is exactly as though the physical items had been gathered together from widely separated sources and bound together to form a new book. It is more than this, for any item can be joined into numerous trails.

There is another, perhaps ignored design that is currently in use for writing hypertext documents that is substantially less clumsy: autocomplete. This is what Fecebutt uses for tagging people or linking to Fecebutt pages in posts, and also what modern IDEs like Eclipse, Jupyter, or IDEA use to help you type long function names. You begin typing the name of an entity to link to, perhaps introduced with a magic character like @ or something, and the system instantly consults a dictionary of possible link targets, orders the possible completions (maybe including possible errors!) by likelihood, and displays the top few in a drop-down list right next to where you're

typing, each accompanied with some kind of preview.

Of course you could use the same approach to actually navigate to possible link targets, and then use a Memex-style link-creation button to link one thing you're looking at (or typing) to the thing you just navigated to. But I think this is less fluid than the dropdown/autocomplete approach.

Ideally, you wouldn't need to explicitly add things one by one to the list of possible link targets; instead, merely visiting them or even seeing a link to them drift by on Twatter would add them to the set of relevant links. You'd have to compensate for the large volume of possibly relevant links by having a reasonably accurate statistical ranking of which things are most likely to be linked to at any given time and for a given "search string"; also, searching through your own, perhaps private, commentary on a link, and that of your friends, should provide additional clues as to which tags are most relevant to a link.

This isn't quite enough for transclusion or quoting, because that is going to require some explicit user action to indicate the extent of the text, pixel data, etc., to be quoted. I think that probably the right solution there is to use an annotation UI to attach marginal notes to highlighted passages, then use tags and titles included in the marginal notes, as well as the contents of the highlighted passage, to feed the search engine. The persistent identity of a link target in this system might become a tricky question as existing works are modified.

Note that a single web page might contain many possible link targets, even without transclusion — for example, one for each header in a long article.

Topics

• Human–computer interaction (p. 3493) (76 notes)
• Hypertext (p. 3512) (13 notes)
• Editors (p. 3426) (13 notes)
• Memex (p. 3571) (2 notes)

Solar dehumidifier

Kragen Javier Sitaker, 2016-08-11 (5 minutes)

There are different kinds of dehumidifiers. The most common kind is the refrigerative dehumidifier, which is basically an air conditioner; it uses a heat pump pumps heat from its intake air into its output air, condensing water out of it in the middle, yielding liquid water.

Another kind that's potentially simpler is the desiccant dehumidifier, which passes the air to dehumidify over a pebble bed of solid desiccant, which absorbs water from it. Eventually, though, the desiccant is full of water, and then it needs to "regenerate" the desiccant, so it connects the desiccant to a different circuit and blows hot air over it to take the water out. The hot, moist air has to go somewhere else besides into the space that it's trying to dehumidify; typically you exhaust it outdoors.

Ideally the hot regeneration air growing more moist flows in the opposite direction from the normal cool air being dehumidified.

(Dehumidifying the air down below the point of comfort offers the opportunity to humidify it again by evaporation, which cools it evaporatively; thus, not only can you dehumidify air by cooling it (and then warming it), but you can cool air by dehumidifying it (and then humidifying it.)

One striking thing about this approach is that it can potentially be done with very primitive materials. It's easier to do if you have plastics, microcontrollers, electric fans, silica gel, and so on, but I think you can do it entirely with stone-age materials, at least if you're willing to accept some compromises.

Ducting

Air ducting can be made from pottery, black-painted if necessary. Salt firing could make the pottery perfectly moisture-proof, and this may be desirable for some parts of the system. Pottery is fragile, and metals or plastics are a better choice where available; even lead would do.

Pumping and Heating

A solar chimney is a simple way to drive airflow using solar energy. Using it to suck air through the desiccant bed implies the loss of some of the desiccated air, but I think it's possible to divert the bulk of it elsewhere.

To heat air, it can be drawn slowly through a sun-warmed duct while it's warm.

Desiccant

The desiccant could be any number of common materials.

Sodium chloride salt is known since ancient times, and is probably adequately hygroscopic, but was too precious for such uses until the internal-combustion engine dropped the cost of mining by orders of magnitude. Various other salts, including sodium hydroxide, calcium chloride, sodium sulfate (mirabilite), and magnesium sulfate (Epsom salt) would also work, but none of them are very common in nature,

and some of them have an inconvenient tendency to deliquesce. Calcium sulfate (gypsum, sold as the commercial desiccant Drierite) doesn't dehydrate until over 100°, and calcium hydroxide until even higher temperatures, which would require inconveniently high temperatures from the regenerating air.

Much more practical, though with a lower minimal humidity level, would be to use biomass as the desiccant. Wood chips, for example, would work, as would coffee beans, which are commonly used for this purpose in saltshakers, or cocoa beans. It's important not to use edible materials; edible beans would attract pests unless they were poisoned. Heavily salting them would probably work without creating a hazard to humans who might consume them in desperation.

Expansive clays such as sodium bentonite would also work well, although you'd probably want to encapsulate them inside some kind of porous solid container, maybe a small unglazed prefired pottery sphere.

Control

Control of the apparatus is achieved by opening and closing valves, which need not seal perfectly. The valves are controlled by varying temperatures, pressures, and humidities in different parts of the device. Varying humidities can be sensed with a twisted hair rope.

Nowadays, if the system has to work when there is no electrical power, we would sense temperature variation using a bimetallic strip; I don't know what you would use in a Stone-Age situation. Perhaps you could use the lengthwise expansion of a long piece of wood referenced against ceramic; Douglas fir lengthens at 3.5 microstrains per kelvin, soda-lime glass at 8.5, and quartz at 0.33. So a 3m long piece of wood anchored to a 3m ceramic pipe would show a difference in length of 64 microns over 20°, which is probably too small to reliably activate a valve.

Tangential expansion is more like 45 microstrains per kelvin, which would bring the difference up to 0.9 mm, but that's also strongly influenced by humidity.

As an alternative to a packed column intermittently regenerated with hot air, you could use an enthalpy wheel, which continuously rotates at about 3 mHz while one side is being regenerated and the other side is being used for desiccation. It's presumably possible to drive its rotation with a light windmill in a sufficiently tall solar chimney, or intermittently using a regular windmill.

Topics

Elastic metamaterials

Kragen Javier Sitaker, 2019-03-19 (17 minutes)

From a materials-engineering point of view, one of the great benefits of organic and biological materials is their elasticity. Of course, organic elastomers like polyisoprene (the largest component of natural rubber latex), high-molecular-weight polydimethylsiloxane (silicone rubber), and polyurethane exhibit the most extreme elasticity, with elastic (or quasielastic) strains† sometimes exceeding 1000%; but other organic polymers have viscoelastic behavior above their glass transition temperatures, which are often quite low, making them very resistant to shattering in everyday use and often capable of elastic strains over 10% and even larger ductile strains. By contrast, common metals are limited to elastic strains of about 1%, although pseudoelasticity can give shape-memory alloys larger strains closer to those of the plastics mentioned above; and common ceramics are closer to 0.1% to 0.01% strain.

Furthermore, biological metamaterials such as wood and sponge can achieve much greater elastic strains than their component materials can. Similar benefits are achieved with crude synthetic metamaterials such as gels, aerogels, foamed metals, aerated cements, and so on.

This is a potentially serious problem for designing self-reproducing automata working from exclusively inorganic feedstocks, an objective I consider important for three different reasons: first, most of the universe, even most of the earth, does not appear to contain life forms, and thus mostly lacks complex organic molecules that could be used to build the automata needed to mine the asteroids; second, it is desirable that any agent controlling such automata not have an incentive to, e.g., deforest large areas of land, in order to build more automata; third, when automata that consume organic matter in order to operate have appeared in the press, public perception has been extremely negative.

Such plastics and elastomers fulfill several very important functions in existing machinery, including vibration isolation (i.e., couplings with a very low derivative of force with respect to displacement), relaxing tolerances, elastic energy storage, and protection from impacts. Though metals and especially ceramics offer higher ultimate strengths than plastics and especially elastomers, members made from metals and (again, especially) ceramics often must be sized orders of magnitude larger than their static and dynamic loadings to resist impact loadings.

Vibration isolation

Advances in vibration isolation have long been crucial to demanding apparatus — Michelson and Morley built their interferometer on a slab of rock floating in mercury, as I understand it, and modern inventions were the key advances that made LIGO possible. But many less-exotic machines need to manage vibration, too. Everyday modern motor mounts are commonly made with combinations of elastomeric supports (under compression) and dashpots.

But there are many other things you can do as well. Supports in compression are necessarily sized to have enough rigidity to resist buckling; this means that even small-displacement vibrations can transmit a lot of force, and thus energy, through them. Supports under flexion, like truck-suspension leaf springs, are not subject to this limitation, but suffer a compensating lever-arm disadvantage. Supports in pure tension can supply much more compliance for a given bearing capacity. This is why a tensegrity structure provides so much more compliance, and thus vibration isolation, than a traditional trusswork.

However, although strings in tension can have a great deal of bearing capacity for their rigidity, as the frequency increases, the string's own density comes into play. This increases the tension on the string for a given displacement whenever the wavelength of the wave (as it would propagate along the string) is comparable to or shorter than the string's length, enabling more vibrational energy to be transmitted through the string. Moreover, the resonance modes of the string can smear out a short-lived vibration over a longer period of time. LIGO† dealt with this in part by using DSP after the fact to filter out vibrations at the vibrational frequencies of the instruments' tension supports.

A different approach to the problem, also used in LIGO, is to use giant-compliance mechanisms made by putting a negative-compliance support (such as an Euler column† near its critical buckling force) in parallel with a positive-compliance support (such as any everyday object). The supports are designed to precisely cancel at the load they must support, enabling that load to "float" over a relatively large range of displacements at near-zero net force.

The energy of a vibration is partly reflected from discontinuities in acoustic impedance; discontinuity-rich environments such as a few meters of dirt and rocks are quite good at preventing the propagation of vibrations, to the point that Elon Musk claimed in his TED talk that neither the US Customs & Border Patrol nor the Israel Defence Force were capable of detecting tunnels dug more than three tunnel diameters under, respectively, the Mexican border and the Gaza Strip. So perhaps a string of beads, each bead connected to the next through a short length of cord, could work like acoustic multilayer insulation to exponentially attenuate transmitted vibrations.

Another possible approach — also at play in hiding tunnels from La Migra — is to use nonlinear interactions to transfer vibrational energy to progressively higher frequency bands. Impacts between relatively rigid particles are one example (say, attaching a box of dry sand to soak up vibrations — we don't normally consider quartz crystals a great damper), but so too are Euler columns crossing their critical stress (which includes ordinary strings alternating between tension and slackness), the thin-shell dynamics of cymbal crashes, and vibrations reaching large stresses in quasielastic substances like rubber.

Relaxing tolerances

Metals and especially ceramics are hard to work with because they demand very tight tolerances. Because their limiting strain is so small, they must be very close to the right shape and position before being brought into contact; typical tolerances for steel machine parts are a few microns, and as I understand it, ceramic parts are even more of a

motherfucker. (This is worsened by the frequent need to do much shaping of ceramics in a green state before final densification.)

Plastics (and, again, especially rubbers) can simplify this problem enormously. If an engine head can seat on a rubber gasket rather than directly onto the cast-iron cylinder, for example, it need not withstand the large forces that would be needed to seal it against the cylinder directly, compensating not only for its manufacturing imperfections but also its thermal warping. In high-vacuum systems, where organic materials generally outgas too much, this role is often taken up by gaskets made from a malleable metal like indium, thus avoiding the need to apply large stresses to glass to seal it against metal.

In another direction, many RepRaps have been built with bits of plastic hose serving as flexible shaft couplings, to allow for some misalignment of shafts. Nowadays, machined springs are more common for this use, which I think is because they last longer.

As an alternative to using plastic gaskets or flexible shaft couplings to compensate for inaccurate parts or assembly, possibly self-reproducing automata can be made from more accurate parts accurately assembled. As an alternative to using them to accommodate for thermal expansion discontinuities, matching thermal coefficients of expansion is possible (and regularly used to seal metal to glass in vacuum tubes); graded-TCE materials may help here.

Elastic energy storage

For the last few centuries, "clockwork" devices have been driven by elastic energy storage, typically in brass watch springs rather than in plastics. Plastics are actually not very good for elastic energy storage, despite what would appear to be higher energy densities: they tend to be viscoelastic rather than purely elastic, losing stored energy to creep relatively rapidly, and quasielastic elastomers like polyisoprene actually store the energy as easily-lost heat rather than strained bond energy as metals do. (For this same reason, rubber makes better motor mounts than would a steel spring of nominally the same stress-strain curve: the rubber dissipates high frequencies.)

Organic materials *are* used for shorter-term elastic energy storage, such as in bows, which must be unstrung when not in use to prevent creep.

Impacts

Impact loadings, as I understand it, are characterized by being limited principally by energy rather than force. If a perfectly rigid sphere of 1 kg is traveling at 1 m/s toward your machinery, it can exert an arbitrarily large pressure or force on the machinery — the collision of two such idealized horrors would produce infinite force and pressure for an instant. But it only has half a joule of energy. If the material it strikes deforms elastically or plastically, the forces actually encountered may be quite low; for example, if it deforms by 1 mm, the sphere stops in about 2 ms, so the force is on the order of 500 N, the weight of only 50 kg. If we suppose that the sphere has a radius of 100 mm, then the pressure is only of the order 30 kPa (5 psi in archaic units). (The precise numbers depend on details of the impact such as the distribution of pressure over the impact area, the

stress–strain curve for the impacted material, its Poisson ratio, its density (!), and so on, so in practice the numbers might be a few times higher or a few times lower.) But if the material only deforms by 100 μm, the impact takes on the order of 200 μs, the force is of the order of 100 kN, and the pressure is of the order of 3 MPa. Inversely, if the material deforms by 10 mm, the impact takes on the order of 20 ms, the force is of the order of 5 N, and the pressure of the order of 300 Pa. Thus, by adjusting the rigidity of the impacted material over two very plausible orders of magnitude, we can adjust its stress inversely over four orders of magnitude.

An additional factor is that very rigid impacts will almost certainly have smaller impact areas than softer impacts, because in a softer impact, the colliding bodies come into contact over a larger area than their initial contact. This factor even comes into play for non-impact loadings, such as tightening a spark plug on a car engine, though ductile substances like lead would also help there.

These are the reasons you can lean your full body weight on a car door window though you can shatter it by flicking a fragment of spark plug at it.

The impact energy a material can absorb, either elastically or plastically, is an intrinsic property of the material: it is almost precisely the integral of its stress over its strain to the limiting strain in question. This gives you some number of joules per cubic meter, or per cubic millimeter. In the case of elastic deformation, this is also the energy density of the material when used as a spring. (Small quibbles may attach from the ability of different structures to recruit more or less of the material's energy absorption capacity, including by way of shear deformations.)

Bending radii and microstructured synthetic metamaterials

So we have seen that compliant materials are crucial for a variety of reasons. How can we achieve them using inorganic materials? One approach is using metamaterials.

If you make a flat plate from a brittle material with 0.02% elongation at break, such as everyday fired clay, assuming linear elastic stress-strain behavior, it will break upon bending to a position where its inner surface is 0.02% shorter than its midline, while its outer surface is 0.02% longer. If it's bent into a circle at this point, the outer surface is a circle 0.02% larger than the midline circle and 0.04% larger than the inner-surface circle. So if, for example, it's 1 mm thick, the radius of the circle is 2.5 m, and its diameter is 5 m. But if you reduce its thickness, the radius of curvature diminishes proportionately.

And that's why aluminum foil and paper can be bent around much tighter curves before yielding and breaking than aluminum plate and fiberboard.

As you reduce the thickness of the material, the bulk approximation does eventually break down, but, interestingly, at micron scales, the material typically shows greater flexibility than the model would predict, rather than less — perhaps because of fewer surface defects and more consistent cooling — until you get to some kind of grain size of the material, which is on the order of 50 μm for everyday fired clay, but could be as small as dozens of picometers for

some atomic and molecular materials.

This is the principle behind the everyday coil spring or knit sock — by structuring ordinary piano wire or cellulose in a particular way, you can get it to yield like rubber, though a steel spring's behavior is closer to ideal elasticity than rubber's — but I think we can take it considerably further. If we make 1-nanometer-thick glass foil, which should be feasibility, we can roll it up in a double spiral; if we want the spiral to be able to unroll flat, we can't exceed the 2500:1 ratio mentioned earlier, so the center of the spiral must be a 5000-nm-diameter circle. If there's 1 nm of space between the glass-foil layers, in a 1.005-mm-diameter roll we can have 250'000 layers on each side, 125'000 per direction. So our 1-mm-diameter roll can uncoil to hundreds of meters of glass foil!

Because only the glass tangent to the roll is exerting an effective force, this spring is a very good approximation of a constant-force spring, like those used instead of counterweights for some sash windows.

This doesn't improve the spring energy density of the mass glass (except by reducing surface defects), but it certainly improves the elongation at break of the assembly: it has gone from 0.02% to several hundred million — several tens of billions of percent.

If we continue to use 0.02% as the elastic strain limit, glass foil thin enough to roll up inside 1 mm must be no more than 200 μm thick.

More complex metamaterials could provide not only tailored stress-strain curves (within the total elastic energy capacity of the underlying bulk material) but multidimensional interactions like auxetic materials. As Merkle's buckling-spring logic thought experiment convincingly demonstrates, the elastic deformation of a massive body with a complex shape can have arbitrarily complex behavior, including Turing-completeness (if the material extends far enough). Recent experiments in computational origami offer promising approaches to this problem.

Temperatures

To look at it another way, much of the problem is that most of the inorganic materials we're familiar with are brittle at room temperature, while many plastics aren't. But window glass is a polymer, too; it's just that its glass transition temperature Tg is higher than what we're used to. Soda-lime glass isn't just compliant once it's orange-hot — it's positively *gummy*. Inversely, we've all seen how rubber behaves when cooled below its Tg with liquid nitrogen — it shatters like glass.

Basically the problem is that we're talking about using materials held together by chemical bonds substantially stronger than many bonds in organic molecules, though not the C-C bonds that hold together blocks of graphite or molecules of polyethylene. So these materials might be more convenient at a higher temperature, perhaps around 400–900 K instead of the 300 we're used to. (Carborundum and diamond may be more viable semiconductors at these temperatures than silicon or germanium.) From an extraterrestrial perspective, Venus is no weirder than Terra, though a bit more expensive to emulate with MLI.

Going the other direction, at 100 or 200 K, perhaps we could use other unfamiliar materials. Water, of course, forms a crystalline solid

at 273 K. I don't know what materials might form glasses at such temperatures; maybe mixtures of common materials (mixtures generally have less tendency to crystallize), or polymers that I'm not familiar with because they're too weakly bound to be stable at 300 K.

Topics

Statement from the Confederation of Teachers

Kragen Javier Sitaker, 2016-10-11 (updated 2016-10-12) (4 minutes)

(Fiction!)

We are the Confederation of Teachers. We exist to guide and preserve society, not just of one group, but of all people, for the benefit of all. We came together because we believe society is important, but society was in grave danger of sinking into the same barbarism that nearly consumed it four generations before our foundation. All of our learning could have been lost; generations could have struggled merely to survive to adulthood, as most have throughout most of history; and society might perish on Earth, its cradle and its grave.

We were formed as a Third World War menaced the world, sending waves of refugees out from Syria, Afghanistan, Iraq, and the Crimea; as the European Union, which had safeguarded peace during those four generations, tottered; as a dangerous madman who loved only power was near being elected the leader of the most powerful state in history; and as the Industrial Age, powered by fossil fuels, steel, and concrete, drew to its filthy close. Suicide and obesity grew more widespread year by year, and as the biosphere collapsed, irreplaceable living species were lost at a rate not seen since the last asteroid impact. Yet human life expectancy had never been greater, literacy had never been higher, and science progressed at a rate never before seen.

We recognized that the existing social structure for guiding the course of civilization, a bureaucratic extension of the violent dominance hierarchies and genocidal territorial dynamics of ape troops, were both manifestly inadequate to the task of preventing total destruction and dangerously unstable in the face of nascent technologies such as nuclear weapons, genetic engineering, ubiquitous communication, space travel, self-replicating machinery, solar energy, universal computation, and artificial general intelligence. So we came together to create new resilient structures so that society could survive the most disruptive era of change in tens of millions of years.

We seek the welfare of everyone, not the advantage of one tribe or nation over another. For this purpose, we study, and out of gratitude to those who have taught us, we teach others, so that they too may become teachers. For this purpose, we consult among ourselves to determine the best course of action for society, and then we guide society to take that course. For that purpose we investigate the truth, each of us constantly seeking to become wiser, more ethical, and more capable, and each supporting the others in this. We serve a higher purpose than any company, state, or political party, because our loyalty is not merely to ourselves but to society as a whole.

We Teachers are all kinds of people: intelligent and dumb, female and male, black and white, young and old, rich and poor, from every country and every social class. So far, we are all human, but if we encounter nonhuman people, we will accept them too. What counts is what each of us contributes to the goals of the Confederation of

Teachers. We do our best to value the contributions of each member without regard to who originated them, struggling against our prejudices which corrupt us into listening only to the words of those we already know and respect.

To achieve these ends, we

Scientia est eorum, quæ sunt cum demonstratione.

Topics

Rediscovering successive parabolic interpolation: derivative-free optimization of scalar functions by fitting a parabola

Kragen Javier Sitaker, 2019-11-26 (updated 2019-11-27) (8 minutes)

There's a relatively simple derivative-free algorithm with superlinear convergence for finding minima or maxima of regular one-dimensional functions, analogous to the method of secants for finding their zeroes. But under almost all circumstances there are better algorithms.

Deriving successive parabolic interpolation

In $1 recognizer diagrams (p. 1264) it is mentioned that the $1 recognizer uses golden-section search, a la *Numerical Recipes*, to approximate the optimal rotation. This algorithm has very slow convergence, similar to binary chop for finding function zeroes.

Is there a way to adapt the method of secants (see Using the method of secants for general optimization (p. 1773)) to approximate the minimum faster? If you have three points (suppose, WOLOG, in order) x_0, x_1, x_2 and the function values $y_0 = f(x_0)$, $y_1 = f(x_1)$, $y_2 = f(x_2)$, then you can calculate some divided differences: $(y_1 - y_0)/(x_1 - x_0)$ gives you precisely the average of the derivative on (x_0, x_1), and similarly for (x_1, x_2). If the second derivative is small, these give us good estimates for the derivative f' at $\frac{1}{2}(x_0 + x_1)$ and $\frac{1}{2}(x_1 + x_2)$. From those two, we should be able to linearly interpolate or extrapolate to find where the derivative should have a zero crossing, and we can sample another point there, $y_4 = f(x_4)$.

This vaguely sounds like Nelder-Mead simplex optimization, but in one dimension, but I don't understand Nelder-Mead well enough to know.

Since this is just looking for a zero of the derivative, it will equally well find maxima or minima, depending on the average second derivative in the neighborhood.

In essence, this amounts to calculating the extremum of a parabola fit to the last three points.

So let's work this out.

```
min_next = (f, x0, x1, x2) => {
    const y0 = f(x0)             // (redundant)
        , y1 = f(x1)             // (redundant)
        , y2 = f(x2)             // f at latest point
        , d0 = (y1-y0)/(x1-x0)   // (redundant)
        , d1 = (y2-y1)/(x2-x1)   // derivative near latest point
        , xa0 = (x0+x1)/2        // (redundant)
        , xa1 = (x1+x2)/2        // where that derivative is
        , d2 = (d1-d0)/(xa1-xa0) // estimate of second derivative,
                                 //     hope it's not zero
        , dx = d1 / d2           // distance from xa1 to
```

```
                                //      extrapolated extremum
    return xa1 - dx             // return extrapolated extremum
}

min_next_n = (f, x0, x1, n) => {
    const xi = [x0, x1, (x0+x1)/2]

    for (let i = 0; i < n; i++) {
        const m = xi.length
        if (isNaN(xi[m - 1])) break
        xi.push(min_next(f, xi[m - 3], xi[m - 2], xi[m - 1]))
    }

    return xi
}

parabolic_extremum = (f, x0, x1, n) => {
    const xi = min_next_n(f, x0, x1, n || 30), m = xi.length
    return isNaN(xi[m - 1]) ? xi[m - 2] : xi[m - 1]
}
```

On simple examples like `x => x * (1 - x) + 0.001 * x*x*x, 0.2, 0.1` it seems to have pretty fast convergence, appearing close to the order φ of convergence like the one-dimensional method of secants. If it happens to run into a horizontal line, though, it fails; for example, `x => x * (1 - x)**2, 0, 1` crashes out at 0.5 because $f(0) = f(1)$. The actual maximum is, of course, 1/3, and a better starting point finds it. (And there's a minimum at 1.)

It seems to have some difficulty with rounding, only producing about 9 places of accuracy in that last example, perhaps because the second-derivative expression above becomes very small.

The lines marked `(redundant)` above are things that, after the first iteration, were already calculated in the previous iteration, so we can avoid calculating them again. What's left over is one function evaluation, five subtractions, an addition, a division by two, three arbitrary divisions, and an operation that can be either a division or a multiplication. To me four divisions per iteration seems kind of heavy-weight.

You might think it would have poor convergence because when we sample the new point there, we are in effect sampling the derivative halfway between that new point and the last old point, so we're only getting halfway to the destination. But we're only delaying convergence by one iteration --- if that derivative-average and the last one point to a place very close to the new point, because the second derivative is almost constant over that interval, the new new point will be very close indeed. So it works out.

It actually can tell whether the extremum it's approaching is a maximum or a minimum, because the estimate that it gets of the second derivative tells it. By replacing a `d2` with `abs(d2)` in the above code, we could make it seek only extrema of one kind.

Unlike the method-of-secants stuff in Using the method of secants for general optimization (p. 1773), it isn't apparent to me how to extend this to functions of vector arguments, functions of more than

one argument, or vector-valued functions.

This is called "successive parabolic approximation" and you should almost never use it

The realization that this amounts to fitting a parabola to the last three points led me to the discovery that the standard name for this method is "successive parabolic interpolation"; Wikipedia explains that it has an order of convergence of about 1.33 and has robustness problems. In particular, though, the fact that it's less than √2 means that it's slower than using Newton-Raphson iteration on a derivative evaluated using automatic differentiation, other than faster, as with the method of secants.

The divided-differences approach given above is, I think, faster than the approach I've found elsewhere of solving the Vandermonde linear system, because it's more incremental. But divided differences is of course a standard way to do polynomial interpolation, and its incrementality is one of its standard advantages.

The only time when it is faster to use successive parabolic interpolation rather than Newton-Raphson iteration is when you can't differentiate the function you're optimizing, even though you believe it to be regular, for example because it's experimentally measured rather than computed, or because your software tools are outdated and don't support automatic differentiation, or because its second derivative fails to exist. (If its first derivative fails to exist, successive parabolic interpolation won't work either.) In such a situation, golden-section search is slower, but sometimes only for high precision; and golden-section search is more robust and doesn't have the rounding problems mentioned above. (The `optimize` function in R uses a mixture of golden-section search and successive parabolic interpolation known as Brent's method.)

So, for example, `min_next_n(x => (1/x - 4)**2 , 0.001, 0.1, 30)` in the above evaluator takes 7 iterations to converge to within 0.1 of the correct result (0.25), 6 more iterations to converge to within 0.01 of the correct result, and 2 more iterations to converge to within 0.001 of the correct result, at which point it starts having reasonable convergence. I think golden-section search would actually be faster up to that point, although I haven't tried it. However, for functions that are well approximated by a quadratic, successive parabolic interpolation is really fast; for example, in `min_next_n(x => Math.sin(x*Math.PI) + Math.sin(x*2*Math.PI), 0.001, 0.1, 30)` it eventually converges to 0.29791559955277436, but it's already at 0.2987 in 5 iterations.

Topics

- Programming (p. 3658) (286 notes)
- Math (p. 3564) (78 notes)
- Mathematical optimization (p. 3611) (29 notes)
- Newton–Raphson iteration (“Newton’s method”) (p. 3595) (6 notes)

• Method of secants (p. 3578) (4 notes)

The AL programming language, dimensional analysis, and typing: do different dimensions really exist?

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

I was reading Raphael Finkel's book "Advanced Programming Language Design", and he mentions a language somebody named Finkel invented in the 1970s called AL. AL had dimensional analysis built in, with four base dimensions: 'time', 'distance', 'angle', and 'mass'.

(As Finkel says, this is the kind of "type safety" that's really needed in everyday calculations. His language did it statically, but you can do it dynamically too, as units(1) does.)

He points out that 'angle' really shouldn't be included since radians are really dimensionless, which leaves us with 'time', 'distance', and 'mass'. But no 'speed', 'force' or 'energy'.

However, they can be derived from the base units; speed is a ratio distance/time — e.g., m/s. Acceleration is speed/time, or m/s^2, and force can be thought of as mass * acceleration, or $kg*m/s^2$. Finally, energy is force * distance, so you can express it in units of $kg*m^2/s^2$. This is in fact how units(1) represents it.

But do we really need all three of 'time', 'distance', and 'mass'? The speed of light provides a natural conversion factor between time and distance, and mass can be equivalently measured as energy, according to the well-known formula $E = mc^2$. So speed is really just a dimensionless quantity, and acceleration is the reciprocal of a time interval, namely the time to reach the speed of light at that constant acceleration; so its units are really 1/s. So force can really be measured merely with kg/s.

Unfortunately this doesn't really help us get rid of mass: energy is now expressed in kg*s/s, or merely kg. So our two relativity-based equivalences (mass as energy, and time as distance) turned out to be merely two facets of the same equivalence.

But I suspect that in everyday calculations, the equivalence of mass and energy is rarely useful; it's far more likely to hide errors than to reduce the amount of work necessary. With time, distance, and mass, the only incommensurable quantities I commonly run into with commensurable units are:

- torque and energy;
- various dimensionless quantities;
- perhaps pressure and stress;
- stress and young's modulus.

Collapsing mass with energy and time with distance creates many more "units collisions".

Topics

Robust hashsplitting with sliding Range Minimum Query

Kragen Javier Sitaker, 2016-09-05 (7 minutes)

Hashsplitting is a process used by bup, `gzip --rsyncable`, and fuzzy hashing used for spam and malware detection. Current hashsplitting systems are vulnerable to malicious input and are also needlessly inefficient. By using the linear-time sliding range-minimum-query algorithm, we can improve its efficiency measurably and its robustness against malicious input dramatically.

Background

Bup and `gzip --rsyncable` and forensic "fuzzy hashing" all use a sliding-checksum algorithm to split their input data into chunks at irregular intervals, determined by the bytes in the neighborhood, in such a way that the chunk boundaries will move with insertions and deletions into the text. Bup calls this process "hashsplitting." This generates, effectively, a pseudorandom number at each byte offset depending on the previous bytes, and tests it against a customizable threshold, or even set of thresholds, in order to figure out where to divide into chunks.

Naïve hashsplit chunk size has a geometric distribution.

Since chunk ending is a memoryless (Bernoulli) process, the distribution of chunk sizes (the "waiting times") follows a geometric distribution, the discrete analogue of the exponential distribution.

The geometric distribution has a very large standard deviation (in the limit, equal to its mean) because occasionally values much larger than the mean will result. For example, here are some interval sizes from a Bernoulli process with parameter .001 [0]:

```
array([2216,  130,  138,  125,  659,  861, 1708,  421, 1794, 1416,  810,
        982,  118,  237, 1755,  383,  841,  632,  200,   85, 1062, 1241,
       1804,  591,  916,  600, 1167,  630,  480,    9,  713, 2656,    2,
        883,  459,   71, 2753,  599,  389, 2187,  593,  442,  348,  764,
       3094, 1371,  481, 2667,  194, 1093,  183,  671,  872,  874,  311,
        345, 1958,  219,   81,  593,  365,  988,  349,  673,  909, 2013,
        267,  387,  503,  146,  339, 4818,  200, 1393, 1556, 1640, 1257,
        816, 1618, 1630, 4719,  699,  693, 2515,  161,  505,  782,  922,
         23,  226,  489, 1757,  933, 1966,  698, 2500,  837,  352, 1508,
        339])
```

XXX can I maybe measure the chunk sizes from bup?

The mean waiting time of this process is 1000 (and the mean of these variates is 964 and their standard deviation 898) but more than a third of them are less than half the mean, and one of them is almost three times that. Two of them are more than four times the mean, which is slightly more than typical.

Geometric chunk size distribution is inefficient.

As noted in the design of bup, for many of the applications of

hashsplitting, although some variability is demanded by the application, this large degree of variability is undesirable.

For example, if the above numbers came from using a sliding-checksum algorithm to divide a slightly modified 96-kilobyte file into roughly-1000-byte chunks and calculate a hash of each, so that we could transmit or store only the newly changed chunks, the changes will be disproportionately concentrated in the larger chunks, which in turn are disproportionately expensive to transmit or store; the 9-byte chunk and the 2-byte chunk are very unlikely to have been the ones that changed! So the cost of a chunk is proportional to the *square* of the chunk size, which, after all, is why we're splitting the input into chunks in the first place. In this case, the extra cost is some 32%.

Current hashsplitting performance is vulnerable to malicious data.

The situation gets worse if the data being handled is actively adversarial. Sliding-checksum algorithms tend to be linear functions of the last N bytes, in order to make them efficient to compute. This means that a data-generating adversary who knows the particular sliding checksum in use can generate data which will either never generate a cut, producing arbitrarily large blocks, or generate a cut very frequently — every four bytes for an Adler-32 (as used by `gzip --rsyncable`), CRC32, or simple 32-bit checksum, for example. Under some circumstances, this "pathological" data could cause serious problems in a program attempting to apply them. Even if the sliding checksum were impractical to calculate preimages for, an attacker could still repeat a "separator" sequence of the size of the checksum's window (64 bytes in Bup) that induces a split.

It would be useful to have a pseudorandom algorithm that still triggered at the same places after faraway insertions or deletions, but whose waiting times didn't suffer from this large variability and vulnerability to adversarial input.

Sliding range minimum query

Range minimum query, or RMQ, is a well-studied algorithmic problem: given some finite sequence H — say, a sequence of hash values — efficiently find the index of a minimum value `H[k]` in some range [i, j), that is, find a `k` such that

```
i <= k < j ∧ ∀n ∈ ℤ: i <= n < j → H[n] >= H[k]
```

(Of course this works just as well for maxima.)

There are a bunch of data structures for this. You can precompute a table of the results in $O(N^2)$ space and time and answer arbitrary queries in constant time, or a "segment tree" of the results in O(N) space and time and answer arbitrary queries in O(log N) time, along with other possibilities.

However, one particular special case is the *sliding* RMQ problem, where the window size `j - i` is a constant; this can be computed for all possible values, with the following simple linear-time algorithm:

```
size = j - i
d = make_deque(max_size = size)
```

```
k = 0
for item in H:
    while d.nonempty() and H[d.rightmost()] > item:
        d.remove_rightmost()

    d.add_on_right(k)
    if d.leftmost() <= k - size:
        d.remove_leftmost()

    yield d.leftmost()
    k += 1
```

Hashsplitting with sliding range minimum query

If you run the sliding RMQ algorithm over a sequence of hash values generated from the text, as long as the values are unique within the window, it is guaranteed to generate at least one split per window size. OH FUCK IT CAN GENERATE ONE SPLIT EVERY BYTE. Is that fixable?

Probably the thing to do is to take, say, every 8KiB window starting from the start of the file, and split at the location of the minimal hash within that window. But that requires no algorithmic cleverness whatsoever.

[0] Python code for calculating waiting times of a Bernoulli process:

```
def bernoulli_process(p):
    while True:
        yield random.random() > p

def intervals(process):
    i = 0
       for item in process:
           i += 1
           if item:
               yield i
               i = 0

x = numpy.array(list(itertools.islice(intervals(bernoulli_process(.999)),
    100)))
```

Topics

Cheap textures

Kragen Javier Sitaker, 2018-10-28 (updated 2019-05-05) (5 minutes)

So I've been thinking about how to render low-computational-cost textures in 2D, for a few different reasons.

Objectives

• Ultra-low-power e-paper computing systems; the objective here is for the energy cost of generating the texture data to be comparable to the energy cost of updating the e-paper to display it. As outlined in Keyboard-powered computers (p. 2220), updating an e-paper display costs on the order of 200 nJ per pixel update, while common 32-bit low-power processors use on the order of 300 pJ per instruction. This means we can afford something like 700 instructions per pixel, if we're content to have the battery life be half of what it could be. But e.g. reducing below 175 instructions per pixel only increases your battery life by 25%.
• Live video generation in software from low-memory computers; the objective here is to be able to display more interesting things on the screen than you have memory space for a full framebuffer. As mentioned in Notes on the STM32 microcontroller family (p. 3176), NTSC composite video has space for about 200 kilopixels in living color, but common 32-bit microcontrollers like the STM32 line have only 4–256 KiB of RAM, despite running at up to 200 MIPS. To the extent that the display contents can be encoded in RAM with many pixels per byte, then rendered in software to a scanline "framebuffer" or a "framebuffer" containing a few scanlines, they can be far richer within these limitations. 200 kilopixels at 30Hz is 6 megapixels per second. At 96 MIPS, that's a budget of 16 instructions per pixel, or 32 instructions per pixel if you're satisfied with a 200×500 display.
• Providing more powerful graphical primitives for GUI systems. Old GUIs, up to around 1998, were so constrained by the slow computers they ran on that they had to update the framebuffer incrementally, because there wasn't enough time to redraw the entire screen in between screen updates. Modern GUIs' programming model and appearance mostly mindlessly apes the GUIs of that epoch, although the IMGUI paradigm, gradients, filters, and transparency are making their way in; Self's cel-animation-inspired effects have made their way into modern GUIs as animated transitions, notably in window managers and jQuery animations; CSS is popularizing arbitrary transform matrices; and Google's Material Design is leading a move to physically-based rendering in user interfaces. But there's a much wider range of possibilities available. Even the CPU of the laptop I'm typing this on averages over 6 64-bit or 128-bit instructions per clock cycle at 1.6 GHz, or 10,000 MIPS, and its LCD screen is only 1920×1080 at 60Hz, or 124 megapixels per second. This is a computational budget of about 80 CPU instructions per pixel. Moreover, its integrated GPU (see Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033)) is capable of doing maybe 50 billion single-precision multiply-accumulates per second (invoked from 12.5 billion instructions), or twice that many in half-precision, and supports OpenCL; if this is correct, this would be a

computational budget of about 400 multiply-accumulates per pixel (invoked from 100 4×SIMD instructions).

• Reducing GUI latency and tearing. A 60Hz screen redraw is already 16.7 ms, which Dan Luu has convincingly shown is already significant in user experience. Double-buffering adds an additional 16.7 ms of latency, and doing it out of sync with the vertical refresh adds an additional random latency that ranges from 0 to 16.7 ms. Typical keystroke-to-screen latencies on modern computers are in the range of 100 ms, and decreasing that by 25% would be a significant improvement. The hardware constraints here are the same as in the previous item: 80 CPU instructions per pixel or 400 GPU operations per pixel.

So, all of these different ways that you can use low-computational-cost texture generation have different computational budgets (175 instructions per pixel for e-paper, 16 for bitbanging NTSC, 80 instructions and 400 multiply-accumulates for the laptop scenarios), but they're all kind of in the same ballpark. Moreover, they're in a completely different world from the Macintosh with its 6-MHz 1-Dhrystone-MIPS 68000 and its 9" 512×342 CRT, which I am assuming refreshed at around 60 Hz, giving it a dot clock a bit over 10 MHz — 10 pixels per instruction. They are, respectively, 1750 times faster, 160 times faster, and 800 times faster (not counting the GPU, which is arguably something like 4000 times faster), relative to the notional "dot clock".

Techniques

• Solid color filling
• Linear gradients
• Polygon filling
• Alternating pixels (vertical, horizontal, or checkerboard)
• LFSR uniform white noise
• Adding images
• Tiling
• Mirrored tiling
• Character generation
• Strength-reduction
• Bit-sliced cellular automata
• Sierpinski textures
• Bresenham's algorithms
• Splines
• 1-D and 2-D palette mapping
• Zoneplates
• Moiré
• Thresholding/signum
• Texture mapping
• Animation

• Transitions
• Temporal dithering
• Dynamical systems
• Domain displacement

• Subpixel jittering

- Filtering
- Perlin noise

To read

- http://www.iquilezles.org/www/articles/simplewater/simplewater.htm
- http://demo-effects.sourceforge.net/
- https://github.com/ocornut/imgui
- http://www.lofibucket.com/articles/64k_intro.html
- http://www.iquilezles.org/www/articles/morenoise/morenoise.htm
- http://jcgt.org/published/0004/02/01/
- https://www.shadertoy.com/view/4ttSWf
- http://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm
- http://erleuchtet.org/~cupe/permanent/enhanced_sphere_tracing.pdf
- https://www.slideshare.net/DICEStudio/five-rendering-ideas-from-battlefield-3-need-for-speed-the-run
- http://web.archive.org/web/20180306233623/https://prideout.net/blog/?p=63
- http://www.microscopics.co.uk/blog/2010/paulstretch-an-interview-with-paul-nasca/
- https://web.archive.org/web/20160325174539/http://freespace.virgin.net/hugo.elias/graphics/x_water.htm
- https://github.com/intel/opencl-intercept-layer/blob/master/docs/kernel_isa.md
- https://nlguillemot.wordpress.com/2017/01/30/intel-gpu-assembly-with-pix-beta/
- https://www.x.org/docs/intel/CHV/intel-gfx-prm-osrc-chv-bsw-vol03-gpu-overview.pdf
- https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-hsw-3d-media-gpgpu-engine_0_1.pdf
- http://renderingpipeline.com/graphics-literature/low-level-gpu-documentation/
- https://doc.lagout.org/electronics/Intel-Graphics-Architecture-ISA-and-microarchitecture.pdf
- https://01.org/linuxgraphics/downloads/stack
-

https://01.org/linuxgraphics/downloads/2018q1-intel-graphics-stack-recipe

- https://linuxhint.com/gpu-programming/
- https://software.intel.com/en-us/articles/introduction-to-gen-assembly
- https://en.wikipedia.org/wiki/Simplex_noise
- http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
- https://mzucker.github.io/html/perlin-noise-math-faq.html
- https://en.wikipedia.org/wiki/Perlin_noise
- https://thebookofshaders.com/11/
- http://web.archive.org/web/20160304052449/http://www.noisemachine.com/talk1/

Topics

- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Graphics (p. 3483) (91 notes)
- Energy (p. 3438) (63 notes)
- Latency (p. 3542) (19 notes)

An almost-in-place mergesort

Kragen Javier Sitaker, 2016-09-07 (5 minutes)

(I'll be surprised if this isn't all in Knuth, but my Knuth is in the US.)

Background

The asymptotically most efficient comparison-based sorting algorithms take O(N log N) time. Mergesort is one of the sort algorithms of this complexity class, along with Quicksort, Heapsort, and the recently discovered Library Sort. It has the desirable property (like Heapsort) that it always takes the same amount of time, unlike Quicksort, which has an $O(N^2)$ worst case. Indeed, you can implement it without conditional execution. Unlike Heapsort and Quicksort, it's also easy to make it a stable sort, which is often desirable (especially for library routines for sorting); and it accesses memory in purely sequential fashion, which is a huge plus as memory hierarchies get deeper, while Heapsort and Quicksort jump all over the place. Mergesort is the only reasonable choice for a comparison-based sort of data on disk.

(Radix sorting algorithms, which don't rely on comparisons, can be O(N) time, and many are; but I won't discuss them further here.)

It has one huge disadvantage, though, which is that to sort a million things, you need memory space for a million and a half of them; you need temporary storage for N/2 elements, which is to say, you need O(N) space. Meanwhile, Heapsort needs only constant space (O(1)), and Quicksort needs O(log N) space.

Solution?

I think there's a solution, but I'm not sure how to make sure it terminates.

One variant of mergesort uses stacks. With three stacks, you can merge sorted runs on S1 and S2 into longer sorted runs on S3 until both S1 and S2 are empty, and then perfect-unshuffle those runs back onto S1 and S2 (one run onto S1, one run onto S2, repeat) before continuing again. This is a little wasteful in that you move each item twice for each merge. Or, with four stacks, you can merge S1 and S2 into runs that you perfect-shuffle between S3 and S4; then you can backwards-merge the S3 and S4 runs, perfect-shuffling them between S1 and S2.

(This originates in the tape-drive days, when your four-kilobyte machine would have to do payroll for your hundred thousand employees. I'm pretty sure I remember that much from Knuth.)

An amusing note is that you don't have to keep track of where the runs are. You can simply refuse to move elements onto the output stack when they would be out of order (because they belong ot the next run), until all the candidates would be out of order, at which point you start a new run.

But suppose you're doing this in memory. As S1 and S2 are shrinking, S3 and S4 are growing by the same amount; it's just that the growth is distributed unequally. You could maybe put S1 and S3 in the same memory space, but growing toward each other with some

spare space in between, unless they collide.

But most of the time, especially in the early stages part of the process, you're fine. At the very end, you need to split the final run between S3 and S4 (or S1 and S2) unless you want to reallocate the space.

But this suggests a solution: what if, when your output stack would overflow, you switch to the other output stack in the middle of the run instead of reallocating? You'll make two small sorted runs instead of one big one, so you won't be making progress on that run, but at least you're not going backward. But it probably won't happen too often, and you're guaranteed to have space on one stack or the other. And if at some point you have to switch back to the first stack (because the other one is full) you won't be creating a new run there; you'll just be extending the one you started earlier. So the number of sorted runs never grows, but always shrinks.

I'm too tired to code this up right now, but I have a suspicion that it will always terminate; but that if the gap size is too small, it might not terminate in a reasonable period of time.

Topics

Cardboard furniture

Kragen Javier Sitaker, 2019-08-01 (updated 2019-08-11) (15 minutes)

I've been watching lots of videos about making cardboard furniture, with an eye to using recycled cardboard from the neighborhood to make some furniture. There are many different techniques: gluing in ribs with hot glue, laminating multiple layers with PVA glue thinned with a bit of water (unaccountably always with the flutes in the same direction), cutting matching notches to slide pieces together at right angles, using single-linerboard corrugated (usually used for packing) to run around curved contours, paper tape like you use on the back of pictures to hang to cover up joints, découpage with glossy magazine paper to add surface strength and a flat surface for painting, etc.

One point frequently made in the videos is that corrugated cardboard is much easier to crease parallel to the flutes; so, for example, if a shelf is supported at the ends and loaded in the middle, the flutes need to run lengthwise to prevent it from creasing and failing.

Pizza box test and thoughts on strength

I just did a quick test with Mina and a pizza box we were discarding. 16 layers of the one-ply corrugated cardboard of the pizza box were sufficient to resist her strongest efforts to break the beam by pressing down on it in the middle, supported near the ends by my hands; but 8 layers were not. The beam was about 40 mm wide and about 300 mm long, and the flutes ran lengthwise. In the end she was only able to "break" the 8 layers by virtue of twisting the beam in the middle, at which point it failed by creasing — decreasing the beam's moment of inertia by bringing its surfaces close together. The pizza box is about 2 mm thick, but this was probably not very significant in this test.

This suggests that maybe to resist her weight with only one layer, such a rectangular beam that's protected from twisting might need to be four times as wide, perhaps 160 mm wide. The U-shaped bend used by the Chairigami chairs to support body weight could perhaps use 110 mm of depth of "web" on each side; if it were notched to 55 mm depth to match a 55 mm notch on a vertical support, that might be adequate. To reduce crushing at the point of contact at the bottoms of the notches, the tab cut out to make the notch could be folded over rather than cut off, thus increasing the contact area and making it not be entirely edge.

The cardboard box resource

Four of the eight corrugated-cardboard boxes I have handy here, left over from Mina's move, are about 4 mm thick, 1300 mm in circumference, and 360 mm in height (and of course the flutes run heightwise), plus another 170 mm of top flap and 170 mm of bottom flap. The other four are apparently the same thickness but somewhat smaller. They have all suffered some minimal damage from the moving process. Since this is Argentina, not the US, they don't have edge-crush-test or bursting test ratings printed on the cardboard.

(This works out to 1300 mm × (360 + 170 + 170) mm × 4 = 3.6 m² of cardboard in the large boxes and perhaps another half of that in the small ones, for a total of 5.4 m².)

Chair ergonomics

We're seated on some 250-mm-seat-height chairs that we both agree are a bit too low for comfort; their seats are about 550 mm square, which for her is a bit too wide, and for me is a bit too narrow. The kitchen chairs, which are maybe a bit too tall, are about 450 mm tall and a somewhat uncomfortably small 350 mm wide. So something like the 360-mm height of the boxes could maybe work well.

More material properties

If I try to support this body's weight (110 kg?) on a 48 mm × 140 mm section of folded cardboard strip from one of these boxes, it barely crushes; at 64 mm thick it had no trouble. This suggests an edge crush pressure of between 120 and 160 kPa or, in medieval units, 17 to 23 psi. So the 8 mm square sections of folded tab I was thinking to use to support notch bottoms will safely hold 8 N, or only about 800 grams of weight. But that's okay, because the majority of the weight will be borne on the edges of the "legs", not the notch bottoms.

A normal singlewall corrugated cardboard box in the US is rated for 32_lbs/inch edge crush test, in medieval units. If it's 4 mm thick, that's 1400 kPa, about ten times the number I got from this test. This makes me think that probably my test is unreliable because I am pretty sure this cardboard is more than 10% of the strength of normal cardboard.

I'll take the geometric mean of these numbers, 400 kPa, or 1.6 kN/m.

How many legs?

Suppose we need to support 200 kg over 300 mm of seat length, front to back. (The whole seat might be 550 mm long, but maybe the weight is not evenly distributed; and maybe I'm sitting down on it hard.) Each 4-mm-thick "leg" provides 480 N of crush strength over that distance.

So we need about four "legs" per person.

The Chairigami dude arranges this by using two double-layer legs (that is, with three layers of linerboard separated by two layers of flutes, rather than two layers of linerboard separated by a single layer of flutes). But if I space the four legs evenly instead, I can shorten the unsupported span length of the seat between legs (to 550 mm ÷ 4 = 138 mm), and avoid having to laboriously laminate layers of cardboard with glue.

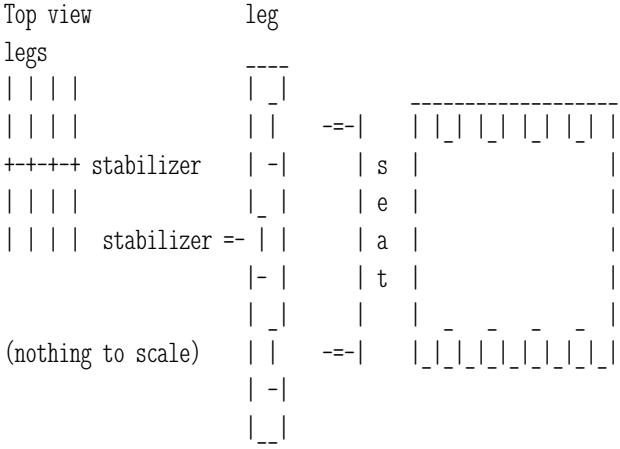
Earlier I suggested that a 300-mm unsupported span might require 110 mm of "web" depth supporting the seat to keep it from collapsing. This shorter span might require only 50 mm, and thus only 25 mm of notch depth.

Initial test plan

I can do an initial test with "legs" much shorter than the 360-mm

height of the desired final product, since the initial objective is crush testing. The seat flutes need to run at right angles to the legs, so I'll probably have to make the seat out of two overlapped pieces to get to the full 550-mm width, since 360 mm is the longest transversely-uncreased flute length I have available; each overlapped piece will have a "top" that is 550 mm long, plus 50 mm of "web" on each side with four 25-mm-deep folded-tab notches in it, which fit into matching notches on the legs. The legs are, say, 650 mm in the cross-flute direction, and 100 mm in the along-flutes direction, and each one has two 25-mm notches on its top, 550 mm apart. Probably I'll need some kind of additional lateral stabilizer to keep the legs from all bending to one side or the other; a single deeper notch in the bottom of each, perhaps 30 mm, can allow the insertion of a single 100-mm-long, 550-mm-wide stabilizer with four 70-mm-deep notches.

A very crude ASCII-art drawing:



Probably I should assemble a paper model first at about 10:1 scale, then assemble that out of cardboard.

If that doesn't turn up any unexpected problems, a full-height stool is probably next (with a bit of under-kick, rather than vertical leg edges), then something with a back. The full-height stool will use four 360 mm × 650 mm legs, a 550 mm × 360 mm stabilizer, and two 360 mm × 650 mm seat scales (perhaps reused from the initial test), for a total of 1.6 m².

If that works out well, I'll know enough to figure out how to scale up to a couch with a back.

Notes from paper model

I assembled a 10:1 paper scale model of the full-size stool (except that the paper is about 0.1 mm rather than 0.4 mm as it would need to be). I learned a number of things.

Making the seat out of two overlapped 36-mm pieces led to a notch only 1.6 mm away from the edge of each, which made it unnecessarily weak and hard to assemble. Moving the two center legs toward the middle would solve this problem.

The "web" flaps at the ends of the seat were short enough at 5 mm that they were hard to assemble. Making them 10 mm or 20 mm (100 mm or 200 mm in full scale) would be a lot better. An unappreciated factor was that those 2.5 mm (25 mm in full scale) notches are the only thing resisting forward-and-back movement of

the legs under the influence of bending of the stabilizer or twisting of the seat around the vertical axis relative to the floor.

The Chairigami chairs bend the seat over the corner of the support and insert its support web into a slanted notch in the front of the vertical support. I probably need to do this in order for the seat not to be extremely uncomfortable for your legs hanging over the front. They also divide the seat into two or more seat sections bent into independent webs.

Another few things occurred to me, though, when looking at the model.

First, maybe the "crush" part is only relevant at the very top and bottom of each leg, in which case maybe I can get double strength by just folding the top and bottom of the leg over so that there's a short width of double cardboard along the top and bottom edges. The middle of the leg would still have to support the compression without crumpling, but that might be a less demanding task. This might allow the stool to be strong enough with just a single pair of legs rather than two pairs of legs. As an extra bonus, Mina's cardboard already has two transverse creases 360 mm apart.

(I'm not sure crushing failure actually works like that, but it seems like it would be worth a try.)

Second, instead of cutting apart the cardboard to make two separate legs, I can perhaps just give it two bends, U-style, to make a pair of legs with a web between them at one end, maybe the back. The web serves part of the purpose of the center stabilizer, but mostly the idea is that making two creases is easier than making one cut. (And if the legs don't have to be parallel, you can make it one crease rather than two, bending it V-style rathe than U-style.) Taking this to the extreme, you could serpentine a long piece of cardboard back and forth in this way, but I don't think the cardboard I have is long enough for that.

Third, though perhaps mutually exclusive with the fold-over-the-top idea, the seat can be contoured by making the tops of the legs curved rather than flat. This would also add extra bending strength to the seat in the direction parallel to the curvature, at least once you press the seat cardboard down into the curves by sitting on it.

Fourth, if the notches are too deep, everything is fine because the bottom surfaces sit on the flat floor and are brought into alignment even if the notch bottoms don't meet; but if they're not deep enough, the cardboard tears. Conclusion: plan to cut them too deep. It might be possible to improve the crushing thing a bit by expanding out each notch into a triangle at the bottom and folding it over, but probably at a heavy cost to other aspects of structural strength.

There was also a slight design-for-assembly problem, in that each of the two seat halves was almost symmetrical and could be glommed onto the legs in a number of different places and in two different orientations. Making the front and back web flaps obviously different lengths would help with reducing the ambiguity.

This was a sufficiently fruitful exercise that I think I should increase the number of prototypes to include a 5:1 cardboard model as well.

Semi-digital fabrication

I don't have a CNC cardboard-cutting machine, or in fact even an X-Acto knife and cutting mat, so I cut the paper model by sharpening a steel kitchen knife tip and cutting on a kitchen cutting board. While this did work, it required me to lay out the design beforehand, using a pencil, adding a substantial fraction of a millimeter of error, and I probably added another couple of millimeters of error with the knife.

Mina has an inkjet printer which I think can do 600 dpi on A4-size paper. That's 42-micron resolution. We can print out designs on this paper and paste it onto the cardboard, then cut and crease the cardboard by hand, following the printed lines.

The cardboard is, however, substantially larger than the paper, and we can't print the paper all the way to the edge ("full-bleed"). So we will probably have to print out multiple sheets of paper, cut full-bleed pieces out of them (in what need not be a particularly precise fashion), and paste them in an overlapping shingled pattern onto the cardboard. An A4 paper is 2^{-4} m², so each square meter of cardboard might require between 16 and 32 printed sheets; the stool above might be about 30. This may be more trouble than it's worth for such a simple design.

Laser-cutting might be a good option, though.

Topics

The miraculous low-rank SVD approximate convolution algorithm

Kragen Javier Sitaker, 2019-08-14 (updated 2019-08-15) (31 minutes)

When reading papers for Real-time bokeh algorithms, and other convolution tricks (p. 2661), I ran across a paper by Tim McGraw on a powerful convolution algorithm. The algorithm took me a while to understand, and it turns out it isn't original to McGraw; it was extensively investigated in the 1970s and 1980s. It's an absolutely astounding technique, and I think it has much broader applicability than is widely appreciated.

Much of the below is somewhat speculative because *I've only just tried the algorithm*. It's very possible I'm misunderstanding its limitations. But it gives me the first general attack on the problem in Sparse filters (p. 834), and I have a lot of reading to do!

The profound and wide-ranging importance of convolution

The *convolution theorem* is one of the most important theorems in the theory of signals (in the sense of "digital signal processing" or "Signals & Systems", not in some kind of semiotic sense). It says that any linear, shift-invariant system is fully characterized by its impulse response, because you can add up a bunch of shifted and scaled copies of that impulse response to compute its response to some arbitrary signal. (Or, in continuous domains, integrate.) This has applications both theoretical, in proving theorems generally applicable to linear, shift-invariant systems, and applied, in computing the result of a linear, shift-invariant system through one or another convolution algorithm.

Significant linear, shift-invariant systems include nearly all acoustic systems (in the time domain), circuits made of linear components (voltage sources, current sources, resistors, transmission lines, capacitors, and inductors — or their idealized versions, anyway), nearly all optical systems in the time domain, and imaging optics in the spatial domain as well. Many systems that are not in fact linear or shift-invariant can be locally approximated as linear shift-invariant systems, though not all. (You can't get the Doppler effect out of a locally-shift-invariant model, for example.)

Furthermore, because convolution in the space or time domain is equivalent to pointwise multiplication in the frequency or Fourier domain, we can do things like frequency filtering, blurring, and sharpening by using convolution.

That point about "sharpening" deserves some sharpening. In theory, any convolution with no zeroes in the frequency domain, which is mathematically almost all convolutions, has an inverse convolution — you just take the reciprocal in the frequency domain and Bob's your auntie! In theory this means that you can undo the degradation caused by almost any known convolution (for example,

defocus blur) by applying an inverse convolution, a process known as "deconvolution". However that inverse does not in general have a finite impulse response. And, because some frequency-domain components of the original convolution may be very small indeed — the high-frequency components of a blur, for example — their reciprocals can be very large, which makes the problem ill-conditioned — that any noise in those frequencies will be enormously amplified by the inverse filter. Wiener filters are the usual compromise solution to this problem.

In two dimensions the impulse response is sometimes called the "point spread function", and sometimes, especially in imaging optics, it's also called the "output transfer function". In the context of computing a convolution it's also called a "kernel". Older papers sometimes call it the "response function" or "amplitude response".

Also, the probability distribution of a sum of random variables is the convolution of the probability distribution of the individual random variables, from which you should be able to see that convolution is commutative and associative. The damned thing just pops up everywhere!

Convolution defined mathematically

Mathematically, the definition of discrete convolution is almost comically simple; using * for convolution:

$(f * g)_t = \Sigma_i f_i g_{t-i} = \Sigma_i f_{t-i} g_i$

Here i ranges over all possible values and t might be some D-dimensional index as well as just an integer. For example, when we're convolving images, it's an (x, y) pair. g might be the impulse response (the "kernel") of some system we're simulating on input f. (You can easily verify that, if $f_0 = 1$ and $f_t = 0$ for all other t — that is, f is a discrete unit impulse — it merely reproduces g.)

(Continuous convolution is the same thing but with an integral.)

Yay convolution!

Despite this extremely broad spectrum of existing applications, I think convolution is actually an underappreciated operation that could be applied much more widely than it is. See the "convolution" section of More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) and Convolution applications (p. 2930) for more detail. One reason it isn't applied more broadly is that it's computationally pretty expensive. And that's where the earthshaking discovery of SVD convolution comes in!

Singular-value decomposition convolution

XXX This section is repetitive

The amazing SVD convolution algorithm uses a "low-rank linear approximation" to approximate convolutions of an image with arbitrary kernels. I first ran across it in Fast Bokeh Effects Using Low-Rank Linear Filters (McGraw, 2014), where it's used to simulate camera bokeh. The basic idea is that you approximate an arbitrary filter kernel, as McGraw says, with a sum of a few separable kernels, which you derive by using SVD (see below for a brief explanation if you're as naïve about linear algebra as I am) on the original kernel. Usually most of the singular values are very small, so you can throw them out.

The algorithm treats the rows and columns of the point spread function you want to approximate as the rows and columns of a matrix, and then uses the singular-value decomposition to find a "best" rank-N approximation of that matrix. You could think of this as approximating each column of the filter kernel as a linear combination of N principal-component columns, which are chosen to represent as much column-to-column variation as possible. (Or analogously for rows.)

There is nothing in SVD convolution that is specifically limited to bokeh; it is a very general technique for efficiently approximating any arbitrary two-dimensional convolution! The speedup you can get without unreasonable degradation depends very much on the kernel you're trying to approximate, but very often kernels contain sufficient similarity between rows or columns to permit an excellent approximation with only a few separable terms.

SVD convolution is specifically two-dimensional, because it relies on the singular-value decomposition (SVD) to compute the N separable filters whose sum is the least-squares-closest rank-N approximation to the original filter kernel. Each of these filters is represented by a pair of vectors, one a horizontal convolution kernel and one a vertical convolution kernel, whose outer product is a matrix that is the convolution kernel of the composition of those convolutions. The sum of the N of these matrices corresponding to the N largest singular values forms the optimal approximation.

In the bokeh paper, McGraw typically got visually very good results with bokeh kernel approximations of rank 3 or greater.

Singular-value decomposition (SVD)

(You can probably skip this if you know linear algebra well, but I don't.)

Singular-value decomposition is a generalization of eigendecomposition† to nonsquare matrices; it decomposes some arbitrary matrix M as a product of three matrices $\mathbf{M} = \mathbf{U\Sigma V^*}$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal (or more generally unitary) and $\mathbf{\Sigma}$ is diagonal. One of many interesting ways to view the result is as a series of least-squares-optimal approximations of the original matrix whose terms are *separable* matrices, meaning that they can be expressed as the outer product of some pair of vectors. Specifically, they are the outer products of corresponding columns of $\mathbf{U}$ and $\mathbf{V}$, scaled by the members of the diagonal of $\mathbf{\Sigma}$; because in general a matrix product $\mathbf{AB}$ can be seen as the sum of the outer products of the columns of $\mathbf{A}$ with the corresponding rows of $\mathbf{B}$.

†*Eigendecomposition* is the decomposition of a square matrix using its *eigenvectors*, a term whose only advantage is as a shibboleth for exposing Malcolm Gladwell — the Spanish term *autovector* is much more informative. I only mentioned this because if you already know about eigenvectors, the above explanation of SVD will be easier to understand.

Since those columns of $\mathbf{U}$ and $\mathbf{V}$ are unit vectors, the elements of $\mathbf{\Sigma}$ are what tell you how big the contribution of each of these separable matrices is to the final matrix sum, so you can get the best N-term approximation by taking the columns corresponding to the N largest elements of $\mathbf{\Sigma}$, which are by convention ordered to be first. It turns out that this is the optimal N-term approximation in the sense of the

Frobenius norm — that is, in the sense of differing by a matrix with the smallest Frobenius norm (the sum of squares of all the elements).

In Numpy the SVD is found at numpy.linalg.svd, which is a wrapper around LAPACK's `*gesdd`, which mostly works by calling `*bdsdc`, both by Ming Gu and Huan Ren. (The `*` is the data type in question.)

Separable filters

A separable filter is a two-dimensional convolution you can compute by first doing a one-dimensional convolution on each row and then doing a one-dimensional convolution on each column of that result. (Or vice versa, since, as it turns out, convolutions commute.) This is great because if you have a convolution kernel that is $w \times h$ and you try to calculate the convolution by brute force, you need $w \cdot h$ multiply-accumulates per pixel, but you can do a separable filter with just $w + h$ multiply-accumulates per pixel. So if you have an 11×11 kernel you can get by with 22 multiply-accumulates instead of 121. Big win!

(That is, your horizontal one-dimensional convolution uses 11 multiply-accumulates on pixels in the same scan line to calculate each pixel of the intermediate image, and then the vertical one-dimensional convolution uses 11 multiply-accumulates on intermediate-image pixels in the same column to calculate each pixel of the final image — 22 in all.)

It turns out that doing this pair of 1-D convolutions is equivalent to doing a 2-D convolution with the outer product of the two convolution kernels, as you can easily calculate.

There are three very popular separable 2-D filters: box filters, double-exponential filters, and Gaussian filters. Box filters and double-exponential filters are popular because you can calculate them in just a few operations per pixel, like, about three along each axis. Gaussian filters are popular because they're circularly symmetric, and they're the *only* separable circularly symmetric filters; also, there are a fair number of physical phenomena with Gaussian behavior, but they get applied pretty often in wildly inappropriate ways. The most common way to approximate a Gaussian filter is actually to run a few iterations of one of the other two, which works because of the Law of Large Numbers.

Since the outer product of two vectors $u \otimes v$ is separable in this way, and any matrix product $\mathbf{UV}$ is a sum of separable terms — each the outer product of a column of $\mathbf{U}$ and the corresponding row of $\mathbf{V}$ — any decomposition of your convolution kernel into a product of two matrices gives you a way to express it as a sum of separable kernels. The outstanding advantage of SVD is that it gives us a decomposition where *as much as possible* of the result comes from *as few as possible* of these outer-product terms.

History

As I said above, SVD convolution was explored extensively in the 1970s and 1980s. The seminal paper is probably Treitel and Shanks, "The Design of Multistage Separable Planar Filters", in 1971:

> A two-dimensional, or planar, digital filter can be described in terms of its planar response function, which is in the form of a matrix of weighting coefficients, or filter array. In many instances the dimensions of these matrices are so large that

their implementation as ordinary planar convolutional filters becomes computationally inefficient. It is possible to expand the given coefficient matrix into a finite and convergent sum of matrix-valued stages. Each stage can be separated with no error into the product of an *m*-length column vector multiplied into an *n*-length row vector, where *m* is the number of rows and *n* is the number of columns of the original filter array. Substantial savings in computer storage and speed result if the given filter array can be represented with a tolerably small error by the first few stages of the expansion. Since each constituent stage consists of two vector-valued functions, further computational economies accrue if the one-dimensional sequences described by these vectors are in turn approximated by one-dimensional recursive filters. Two geophysical examples have been selected to illustrate how the present design techniques may be reduced to practice.

The paper doesn't use the term "singular-value decomposition", perhaps because it was new at the time, instead explaining how to derive the SVD from eigendecomposition. By 1975 papers were using the term.

In 1980 Sang Uk Lee did his dissertation on it, "Design of SVD/SGK Convolution Filters for Image Processing", citing Treitel and Shanks 1971 and also Twogood and Mitra's 1977 "Computer-Aided Design of Separable Two-Dimensional Digital Filters", which also cites Treitel and Shanks. Andreas Antoniou and what I assume are his students, such as Wu-Sheng LU and Hui-Ping WANG, continued publishing on the subject through the 1980s, and, for example, in 1990 published "Design of Two-Dimensional FIR Digital Filters by Using the Singular-Value Decomposition", in which they find ways to further modify the filter to decrease the computational load, one using LU decomposition.

Mitra, Grosen, and Neuvo published a couple of papers in 1985 on extending the algorithm to one-dimensional signals by partitioning them into equal-sized chunks.

Work continues today, perhaps at a reduced pace; Atkins, Strauss, and Zhang published "Approximate convolution using partitioned truncated singular value decomposition filtering for binaural rendering" in 2013 on a way to filter audio to produce the binaural sensation of space, and in 2014 McGraw published the paper where I learned about the technique. McGraw doesn't cite this earlier work and seems to be unaware of it; he may have invented the technique independently 40 years later.

Of all of the above, I find Lee's dissertation to be by far the most readable, perhaps in part because it's jargon-compatible with me. The earlier work largely uses terminology I find confusing, and the later work assumes familiarity with the earlier work. The Treitel–Shanks paper is a close second, despite the alien jargon, because it's very well written, but of course it only covers developments up to 1971.

A toy example

With Numpy and SciPy, it's pretty easy to test the algorithm out; the following took me about half an hour with IPython. It's probably better to use IPython or Jupyter, with `%pylab inline`, so you can use `matshow` to see the values as images.

First, let's compute a flat circular convolution kernel, like a typical camera bokeh:

```
>>> import scipy.signal
>>> import numpy.linalg
>>> import numpy
```

```
>>> r = range(-8, 9)
>>> x, y = numpy.meshgrid(r, r)
>>> circle = (x**2 + y**2 < 64)
>>> # matshow(circle)
>>> print('\n'.join(''.join(map(str, 0 + row)) for row in circle))
00000000000000000
00000111111100000
00011111111111000
00111111111111100
00111111111111100
01111111111111110
01111111111111110
01111111111111110
01111111111111110
01111111111111110
01111111111111110
01111111111111110
00111111111111100
00111111111111100
00011111111111000
00000111111100000
00000000000000000
```

Let's generate an image to convolve with that circle kernel — all zero except for a couple of bright points at (2, 2) and (8, 13). Numpy array coordinates are in (row, column) order, which is to say, (y, x).

```
>>> p = numpy.zeros((16, 16))
>>> p[2, 2] = 3
>>> p[8, 13] = 2
>>> # matshow(p)
>>> p
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 3., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 2., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

We can do the convolution with brute force using scipy.signal.convolve2d:

```
>>> perfect = scipy.signal.convolve2d(p, circle)
>>> # matshow(perfect)
>>> print('\n'.join(''.join(map(str, row.astype(int))) for row in perfect))
```

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000003333333000000000000000000
00000333333333330000000000000000
00003333333333333000000000000000
00003333333333333300000000000000
00003333333333333300000000000000
00033333333333333300000000000000
00033333333333333300000000000000
00033333333333333322222220000000
00033333333333335522222222200000
00033333333333355522222222220000
00033333333333355522222222220000
00033333333333555522222222222000
00003333333333555222222222222000
00003333333333555222222222222000
00000333333333522222222222222000
00000000000000222222222222222000
00000000000000222222222222222000
00000000000000222222222222220000
00000000000000222222222222220000
00000000000000022222222222200000
00000000000000000222222200000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
>>> perfect.max(axis=0)
array([0., 0., 0., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 5., 5., 5.,
       5., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 0., 0., 0.])
```

Now let's compute the same result using SVD convolution instead.

As it happens, at 17×17, the circle only has four distinct nonzero columns, so, considered as a matrix, its rank is 4. Consequently, it has four nonzero singular values, not counting rounding errors:

```
>>> u, s, v = numpy.linalg.svd(circle)
>>> s
array([1.33371921e+01, 3.01484321e+00, 2.04182621e+00, 1.36417508e+00,
       3.34375584e-16, 1.28668530e-16, 7.48637395e-17, 3.74505958e-17,
       5.96041076e-32, 4.06050202e-33, 3.67797309e-33, 5.31269989e-34,
       2.72274806e-49, 1.79207201e-49, 1.91581056e-50, 0.00000000e+00,
       0.00000000e+00])
```

numpy.linalg.svd gives us **Σ** as a vector, as seen above, and **V** in its transposed form ready to multiply. So we can, for example, reconstruct a rank-3 approximation of the circle and measure its error as follows:

```
>>> circle3 = (u[..., :3] * s[:3]).dot(v[:3, ...])
>>> # matshow(circle3)
>>> ((circle3 - circle)**2).sum(), (circle**2).sum()
(1.8609736397047545, 193)
```

We can use the same brute-force approach to convolve this approximation with the image, which doesn't save us any computation time, but does validate that the low-rank approximation, this sum of three separable kernels, does some kind of a reasonable approximation:

```
>>> approx3 = scipy.signal.convolve2d(p, circle3)
>>> # matshow(approx3)
>>> approx3.max(axis=1)
array([0.        , 0.        , 0.        , 2.96621523, 3.09214447,
       3.28014338, 3.28014338, 3.09214447, 3.09214447, 3.23906193,
       4.69144233, 5.27890672, 5.27890672, 5.06351288, 5.34157302,
       5.34157302, 4.46037644, 2.96621523, 2.06142965, 2.06142965,
       2.18676225, 2.18676225, 2.06142965, 1.97747682, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        ])
>>> print('\n'.join(''.join(map(str, (row + 0.5).astype(int))) # rounding
                    for row in approx3))
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000033333330000000000000000000
000012233333332210000000000000000
000023333333333200000000000000000
000023333333333200000000000000000
000333333333333330000000000000000
000333333333333330000000000000000
000333333333333332222222000000000
000333333333333455222222221000000
000333333333333555222222222000000
000333333333333555222222222000000
000333333333335555222222222000000
000023333333335542222222222000000
000023333333335542222222222000000
000012233333334431222222222000000
000000033333332212222222222000000
000000000000002222222222222000000
000000000000002222222222222000000
000000000000002222222222222000000
000000000000002222222222222000000
000000000000001222222222221000000
000000000000000002222222000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
```

So now let's try the algorithm more or less for real. First, we convolve each column of **U** with each row of our picture `p`, thus getting one horizontally-convolved intermediate image for each term of our approximation. There's probably a way to do this purely with Numpy without using Python interpretive `for` looping:

```
>>> hterms = numpy.array([[s[j] * numpy.convolve(u[..., j], p[i])
                          for i in range(len(p))]
                         for j in range(len(u[0]))])
>>> #matshow(hterms[0])
```

Here are a couple of rows from the first approximation term:

```
>>> hterms[0][1:3]
array([[  0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ,  -6.00495857,
         -9.16629667, -10.54084337, -10.54084337, -11.44132655,
        -11.44132655, -11.44132655, -11.44132655, -11.44132655,
        -11.44132655, -11.44132655, -10.54084337, -10.54084337,
         -9.16629667,  -6.00495857,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ]])
```

Then we can convolve each row of `v` (**V***) with each *column* of its corresponding intermediate image, thus generating a stack of images that are the terms of our approximation; the sum of the first three of these is equal to the result of our brute-force convolution with the approximation from earlier, except for rounding error:

```
>>> terms = numpy.array([numpy.array([numpy.convolve(v[j], hterm.T[i])
                                      for i in range(len(hterm.T))]).T
                         for j, hterm in enumerate(hterms)])
>>> approx3_fast = terms[0:3].sum(axis=0)
>>> # matshow(approx3_fast)
>>> abs(approx3_fast - approx3).max()
4.884981308350689e-15
```

And, because our kernel in this case is only of rank 4, the rank-4 approximation, summing four separable kernels, is exactly equal to the correct convolution result, again except for rounding error:

```
>>> abs(terms[0:4].sum(axis=0) - perfect).sum()
4.3509640335059885e-13
```

I say that was *more or less* for real because I calculated all the terms, not just the first four terms that I used, and I was using interpretive

loops rather than trying to get `convolve2d` to do one-dimensional convolutions, so running it exactly as above on a larger image would be *slower* than the brute-force approach.

Direct applications

SVD convolution can be applied directly to sharpening and blurring images, to Wiener filtering of images that have been corrupted by suboptimal optics, to object recognition and convolutional feature extraction in images, and to computing approximations of the performance of optical systems. In the context of mathematical optimization, for example of simulated optical systems, a computationally inexpensive approximation is immensely valuable, because it allows many more optimization cycles to run.

For applications like object recognition in images, it's common to have many more candidate filter kernels than images. In this case, it might be more productive to use a low-rank approximation of the original image rather than of the filter kernels, which is guaranteed to work because convolution is commutative.

Extensions

What useful extensions of SVD convolution might be possible, but aren't simply straightforward applications of it?

To more dimensions

The D-dimensional equivalent problem is well-posed: find N sequences of D vectors whose D-dimensional outer products sum to form the optimal approximation to the original D-dimensional convolution kernel. It can't be solved directly with SVD, but maybe there's a way to apply SVD more indirectly to a couple of dimensions at a time, or maybe you could use a recursive iterative-approximation algorithm that subtracts off the D-dimensional outer product of the average vectors of the residual along each dimension. If not, I'm optimistic that, e.g., gradient-descent variants like Adam or quasi-Newton methods are sufficiently powerful to find a good solution.

I ran across a 2009 paper by Oseledets and Tyrtyshnikov, "Breaking the curse of dimensionality, or how to use SVD in many dimensions", which explains something called "the *Tree-Tucker* format" using "Tucker decomposition"; this sounds similar to the above. I haven't finished reading it.

To one dimension

If $D = 1$, then the problem as posed above is trivial: the convolution kernel is a vector, and it is its own best vector approximation, $N = 1$. To get a more useful result, we would need to find a way of somehow approximating the convolution with shorter vectors.

One possibility is to "word-wrap" or "raster" a one-dimensional time-domain sequence of $a{\cdot}b$ points into a two-dimensional signal of dimensions $a \times b$ where a is small compared to the size of our desired kernel, ideally close to its square root. Then you raster the kernel analogously into an $a \times c$ shape. If you do wraparound onto the previous and next a-element "scan line", a two-dimensional convolution on these two-dimensional signals is precisely the same

thing as the original one-dimensional convolution. If the kernel has substantial periodicity, the way bandpass convolution filters do, SVD may yield a good low-rank approximation if a is a multiple of its period. But if the columns of the kernel are perfectly uncorrelated, no good low-rank approximation will exist.

So, computing the original time-domain convolution by brute force required ac multiply-accumulates per sample, and now we can do it in $a + c$ multiply-accumulates per sample per term — if we're using a rank-3 approximation, $3(a + c)$ multiply-accumulates per sample. This is a win if $3(a + c) < ac$ — if $a \approx c$, that's roughly $6a < a^2$, which is true if $a > 6$, or more generally, is more than twice the rank of the approximation.

Unwrapping the above computation back into one dimension, we can view each term of this algorithm as first doing a size-a one-dimensional convolution, then a *sparse* size-ac one-dimensional convolution, with c taps at intervals of a.

(I think all of the above is in the Atkins et al. paper from 2013 that I mentioned above, but I haven't really read it yet.)

A key observation here is that there's no particular reason for these terms to use the same value of a, and in fact it's probably advantageous for them to use *different* values of a, because the first pass using a_0 will probably suck up most of the energy at frequencies that fit neatly into a_0 — the residual error will be particularly low around those frequencies. So it might be a good idea to use a sequence of strides a_1, a_2, etc., zero-padding the kernel if necessary, to get a better "low-rank" approximation. The autocorrelation function of the residual kernel is probably a good guide to picking those strides, although this greedy algorithm might not produce optimal results.

Also, since this reduces one linear convolution to two or more cheaper linear convolutions, it can be applied recursively — for example, you could reduce a 1000-tap kernel to a 10-tap kernel and a 100-tap kernel, then reduce the 100-tap kernel to two 10-tap kernels. This is clearly cheaper to compute if you're only using the first vector from the SVD, but you might need more than one! If you use more, we're not talking about *two* 10-tap kernels in the final stage, but about *many* final 10-tap kernels — 4 of them if you use two components each time. Still a big win — $2{\cdot}10 + 4{\cdot}10 + 4{\cdot}10 = 100$, much less than 1000 — but less so.

If you do this kind of recursive decomposition (which, incidentally, can also be used on the one-dimensional kernels that result from the two-dimensional algorithm), the SVD no longer gives you fully optimal results, because you aren't precisely using the first principal components — you're using some approximation to them, leaving some extra error. This suggests that some kind of iterative algorithm like the greedy algorithm described above will probably produce better results.

(I think you can probably approximate the first principal component reasonably efficiently by using the PageRank algorithm on **M*M** and **MM*** to get their largest eigenvectors, rather than computing the full SVD.)

On prefix-sum images (uh, probably not)

One of the annoying features of the low-rank approximations produced by SVD is that they don't deal very well with diagonal

edges or gradients in the kernel. I wonder if you could "precondition" the image by running horizontal and vertical prefix sums on it (allowing wraparound on overflow) and apply compensating finite-difference operators *on the kernel* before computing the SVD approximation. Without the approximation step, this would be precisely equivalent, because the integration and differentiation operators cancel, and convolution is associative. But perhaps by removing the large-scale structure from the kernel, leaving more local differences, the SVD approximation would have less error.

As an extreme example, consider a kernel like this contrived rank-5 specimen:

```
 1  2  3  4  5
 2  4  6  8 10
 3  6  9 12 15
 4  8 12 16 20
 5 10 15 20 25
```

Upon applying a horizontal backward differences operator to it (the inverse of horizontal prefix sum), you get this rank-1 result:

```
1  1  1  1  1
2  2  2  2  2
3  3  3  3  3
4  4  4  4  4
5  5  5  5  5
```

And on differencing that vertically, you get this singular matrix:

```
1  1  1  1  1
1  1  1  1  1
1  1  1  1  1
1  1  1  1  1
1  1  1  1  1
```

This kernel is clearly much easier to represent.

I'm very much not sure that this would help, though; I think it depends on the kinds of features you find in kernels.

(P.S. I tried it on a kernel like the circle bokeh kernel above. It made the approximation enormously worse.)

Topics

• Sparse filters (p. 3725) (11 notes)
• Linear algebra (p. 3551) (4 notes)
• Singular-value decomposition

Simple state machines

Kragen Javier Sitaker, 2016-09-19 (updated 2016-09-24) (8 minutes)

I was thinking about laser-cut mechanical computation systems. What are the simplest finite-state machines I could usefully implement?

Bit transformation pipelines

The simplest finite-state machines have two states and two inputs. There are 16 possible such machines, corresponding to the 16 fundamental logic gates. Calling the state Q and the input I, we can display all 16 state transition tables as one:

```
Q I   new state; each column defines a machine
0 0   0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 1   0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
1 0   0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1   0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

Many of these machines are not very interesting; you kind of need transitions in both directions between the states to be interesting. That reduces the table to the following:

```
Q I   new state; each column defines a machine
0 0   0 0 0 1 1 1 1 1
0 1   1 1 1 0 0 0 1 1
1 0   0 0 1 0 0 1 0 0
1 1   0 1 0 0 1 0 0 1
```

You also probably want to eliminate the state machines where the two possible inputs are equivalent. If the two states are equivalent, that might still be okay, because the state itself might be the machine's output. This eliminates one more machine, which alternated between states incessantly regardless of the input, reducing the table to the following:

```
Q I   new state; each column defines a machine
0 0   0 0 0 1 1 1 1
0 1   1 1 1 0 0 0 1
1 0   0 0 1 0 0 1 0
1 1   0 1 0 0 1 0 1
```

We can characterize these eight machines as follows:

- 0100: stable in state 0 when the input is 0, but when triggered by a 1 input, transitions to state 1 for one cycle. Alternatively you could call it a controllable oscillator: the input turns it on, and when the input is off, it stays in state 0.
- 0101: its state is the input from the previous cycle. This is basically a D flip-flop.
- 0110: stable in either state when the input is 0, but oscillates when the input is 1. This is basically a T flip-flop.
- 1000: the same as 0100 with the sense of the input inverted.

• 1001: the same as 0110 with the sense of the input inverted.
• 1010: its state is the inverted input from the previous cycle, like a D flip-flop's Q̄ inverted output.
• 1101: stable in state 1 when the input is 1, but when triggered by a 0 input, transitions to state 0 for one cycle. This is the same as 0100 but with the input and the state both inverted.
• 1110: same as 1101, but with the input inverted.

We can classify these "interesting" machines into three categories of behavior. Oscillators (0100, 1000, 1101, 1110) produce a Nyquist-frequency square wave when the input is in one state, or a constant history-independent output when it is in a different state. Delays (0101, 1010) delay the input by a clock cycle (and 1010 inverts it too). Bistable machines (0110, 1001) can stably remember a bit of information over time, but there is a given input that will cause them to switch (1 for 0110, 0 for 1001).

In a sense these machines all have "fan-in": even though they only have a single one-bit input, their output is a function of their previous input history (and initial state) as well their current input. But because they can only take a single input, the only topology in which you can hook them together is as a fanning-out tree, in which any given bit of output is produced from a linear pipeline of state machines from the input bitstream. (Alternatively, you could hook them up into a loop, taking no input from outside.)

This may be adequate to do things like produce a binary count of the number of 1 bits in a bitstream, although I'm not even sure of that. An interesting question is whether such a pipeline can in some sense emulate any arbitrary finite state machine (that doesn't get stuck), or what its expressive limits are, and how many of the above machines are needed to provide that expressiveness — presumably you need at least a bistable machine.

The lack of signal fan-in keeps the delay machines from being especially useful in this context, so we can consider just the bistable and oscillator machines.

If you feed a Nyquist-frequency square wave into a bistable machine, you get a half-Nyquist-frequency square wave. If you feed that into the appropriate kind of oscillator, it will reduce the duty cycle of the square wave; feeding that into another bistable machine gives you a quarter-Nyquist-frequency square wave. Feeding that into another bistable machine gives you the repeating sequence 10100000 or its complement. If you feed that into a bistable 0110 that's initially in the 0 state, it becomes 01000000, which when fed to another one becomes an eighth-Nyquist-frequency square wave.

This brings up the interesting fact that the initial states of the state machines can matter a lot. You could imagine a loop of such machines whose "input" is provided entirely by its initial state.

Time-delayed logic gates

If we consider state machines with two states and two binary inputs, one of them is a time-delayed NAND gate:

```
Q I J  new state
0 0 0  1
0 0 1  1
0 1 0  1
```

```
0 1 1   0
1 0 0   1
1 0 1   1
1 1 0   1
1 1 1   0
```

This is clearly sufficient for universal computation on its own, without any other kinds of machines.

Is there some kind of finite state machine that is in some sense simpler than this, but that still allows us to build universal computers by connecting networks of them together? I don't think there is.

J-K flip-flops

Above I mentioned that D and T flip-flops were two of the three kinds of interesting bit-transformation machines. J-K flip-flops are more widely used as a discrete component; their state transition table, assuming active-high inputs, looks like this:

```
Q J K  new state
0 0 0  0
0 0 1  0
0 1 0  1
0 1 1  1
1 0 0  1
1 0 1  0
1 1 0  1
1 1 1  0
```

This seems to me like it is certainly a universal gate in the same sense as the delayed NAND above, which is in some sense of equivalent complexity. You have inputs that will set the stored bit or clear it when they are high, and you can connect them to separate places. Additionally you have the magic power of making the output oscillate by putting them both high.

This extra magic power is pretty clearly not needed; you could use it as just an S-R flip-flop. But if you try to use this to simplify the machine, you are kind of forced to lump J and K together into one input:

```
Q I  new state
0 0  0
0 J  1
0 K  0
1 0  1
1 J  1
1 K  0
```

The problem with this is that you have lost not just fan-in but also compositionality; this machine's state can no longer be the input of another identical machine.

Topics

Hearing aids for disability compensation, protection, and augmentation

Kragen Javier Sitaker, 2019-09-08 (updated 2019-09-09) (4 minutes)

I want a hackable hearing aid for three reasons: disability compensation, protection, and augmentation. I was hoping to get one this year, but that seems unlikely now.

Disability compensation

This body is considerably more sensitive to strong stimuli than average human bodies, due to a syndrome known as autism. Sounds such as a pneumatic cylinder opening a bus door or a whistle during applause cause it acute pain, though that pain fades immediately — the main difference from the pain of, for example, barking one's shins on a pipe. Moreover, continued stimulation at a lower level, such as the traffic noise outside the window, cause continued low-level stress.

Earplugs mechanically block high-frequency sounds, reaching attenuations of some 30 dB above around 200–400 Hz, but they are helpless against sounds below 100 Hz. Complementarily, noise-canceling headphones like the Bose QuietComfort 25 can attenuate low-frequency sounds by some 30 dB by generating cancelation waves, but this benefit starts to roll off above some 100 Hz.

A hearing aid with noise-cancelation firmware offers both benefits at once, and has the major benefit that it is socially acceptable; the other humans will tolerate it considerably better than they will tolerate a human wearing earplugs or headphones during a conversation. Also, they can offer customizable nonlinear and time-variable attenuation curve, for example performing no attenuation while sound levels are under 60 dBa.

Hearing-aid earmolds are also safe and comfortable to wear for many hours a day, while disposable earplugs tend to cause irritation and even pain after more than an hour or three.

Protection

Aside from the pure pain, loud noises can also cause ear damage, and in the same way that earplugs and noise-cancelation headphones can prevent pain, they can also prevent damage.

Augmentation

A hearing aid with wireless communication can provide a private audio output channel from personal computers (pocket and otherwise), for example to provide alerts, reminders, or query results. Furthermore, it can also provide a private audio *input* channel to personal pocket computers, since it has a microphone; use as a substitute for Bell's Telephone is one possibility. Autonomous computing operation, including peer-to-peer Bluetooth communication, is another whole class of potential uses.

As discussed in Bokeh pointcasting (p. 92), a private bidirectional

communication channel to an external computing resource can, under some circumstances, save your life — the case considered there was sounding an alarm during armed home invasion.

Cocktail-party beamforming, perhaps using a chest-mounted microphone array, might enable this body to participate in social events it is currently excluded from, due to its inability to understand anything anyone is saying.

With ultrasound-capable microphones, simple heterodyning software could make bat calls audible. Other forms of "extrasensory perception" are implementable: detection of magnetic fields, electric fields, radio waves, carbon monoxide, anoxic atmospheres, objects such as motorcyclists rapidly approaching from behind, and so on.

Such intimate integration of a computer into a human life demands extreme caution about privacy and computer security problems, given the deplorable state of human computer security.

All of this is limited by battery considerations, though perhaps a flat-flex cable from the hearing aid taped to the back of the neck with flesh-colored tape could provide access to a much larger battery in, say, a pocket; energy-harvesting antennas may also be an option.

Topics

- Independence (p. 3520) (63 notes)
- Augmentation (p. 3333) (5 notes)
- Privacy (p. 3650) (2 notes)
- Autism (p. 3334) (2 notes)
- Hearing aid

Holographic archival

Kragen Javier Sitaker, 2014-04-24 (10 minutes)

I thought I'd written about this before, but I can't seem to find it in the archives.

The Rosetta Project wants to make an etched nickel-alloy disc with reduced-size images of some ten or twenty thousand pages, readable under a 650× optical microscope. They're using a process developed at Sandia based on microchip etching techniques — that is to say, microphotolithography.

This approach has a few significant drawbacks:

• It's extremely expensive. We're used to microchips being cheap, but in fact they cost several dollars each, and each of the Rosetta discs is like an entire wafer, not a single microchip. So we can expect the price per disc to stay in the hundreds to thousands of US dollars.
• You need a microscope to read it. It would be better if you could read it with no special equipment, or only very simple special equipment, such as could be easily improvised from rocks.
• The information capacity is limited by the wavelength of light. Pixels in the original image can't be more than about half a wavelength across for an optical microscope to resolve them: figure 250nm. That gives you a density of, at best, 16 terabits per square meter. With the 7.6-centimeter Rosetta disk, you only have 72 gigabits, not even enough for English Wikipedia in a readable font.

Suppose we could instead usefully encode archival information at a density substantially higher than a light wavelength, reproduce it cheaply, and read it without special equipment. That might produce a much more useful Rosetta disc.

I think that rainbow holography may provide this medium. Rainbow holograms have been inexpensively mass-produced since 1984; they can be viewed in sunlight; and their ability to encode many images in the same film is limited by the film resolution and the directionality of the light source.

Rainbow holograms can present a different image for each angle within an 180° viewing arc. The resolution of this image is degraded as more images from more angles are recorded, and the angle of the illumination and viewer are limiting factors in the number of angles. The degradation of the image is vaguely related in a way I don't understand to the square root of the number of images.

The illumination angle from the sun is the first problem: the sun subtends some 32 arcminutes, and without further work, this limits you to about 330 distinct images. This is about 1½ to 2 orders of magnitude worse than our objective. Fortunately, direct sunlight is excessive; it amounts to about 100klx, while people can read comfortably at 100lx, three orders of magnitude dimmer. If you go into a camera obscura with the holographic archival disc, illuminated by the sun shining through a vertical slit, you can obtain illumination that is much more directional and therefore allows you much more precision in imaging. You simply set the disc in the beam of light stretching across the floor from the slit in the wall.

The slit should, ideally, reduce the illumination by about three

orders of magnitude from direct sunlight; but the illumination will vary across the beam, since the sides of the beam can only see the edges of the sun's disc, and will therefore be dimmer. If we try for a factor of 2 between the brightest and dimmest parts of the disc, then we want the points on the edge of the sun's disc at 30° above and below the horizontal, since sin 30° = ½. cos 30° ≈ 0.87, so we only lose about 13% of the sun's width this way, leaving us with a usable sun-disk width of about 28 arcminutes horizontally.

For a beam spreading out from a thin slit with a divergence of 28 arcminutes to illuminate a whole 7.6-centimeter-wide disc, the disc needs to be some 9.4 meters away from the slit, which makes for a pretty big camera obscura. Now, how wide is the slit? We want to see about a thousandth of the roughly 28-arcminute-wide sun through it. To see the entire 28-arcminute-wide chunk of the sun through it, it would be about 7.6 centimeters across; instead, it needs to be about a thousandth of that, or about 76 microns. That is, it would need to be about the width of a human hair, and at least some 10 centimeters tall, although ideally many meters tall. It doesn't need to be perfectly straight, but it can't deviate from the vertical by more than its width within the 10-centimeter-tall chunk illuminating the disk at any moment.

So that's probably a bit too stringent; it might allow us to put 330 000 naked-eye-viewable images on the disk, but only at the cost of the solar illumination mechanism being unreasonably finicky.

Suppose instead that we allow the slit to be a millimeter wide, which will also boost our light level from 100 lux to 1300 lux, which is quite comfortable. Now, at 9.4 meters, it subtends a 9400th of a radian, which means that the light can illuminate the disc from about 30 000 distinct angles. By rotating the edge of the disc 4 microns to the left or right, we can switch to the "next" or "previous" "page".

How about our eyes? If our pupils are too big, we may need to wear pinhole goggles, or peek through a finger pinhole, to see a single page instead of a mixture of several. If we're trying to read at a distance of one meter, so that we can reach the disc with our hands, then a 9400th of a radian is about 0.1 millimeters. Pupils only shrink down to about 3 to 5 millimeters, so such dense encoding would indeed require something like pinhole goggles; but rather than pure pinhole goggles, they can be vertical-slit goggles, which can probably be improvised adequately from rocks, sticks, etc.

Can we really interfere 30 000 distinct wavefronts and get an image of reasonable resolution and contrast? Supposing the "grain size" of the "film" is 50 nanometers and we're shooting for 100 dpi resolution, each of our 0.01" pixels contains about 5000×5000 grains. I'm vague on exactly how holography scales, but I'm pretty sure it can manage to encode a pixel each of 30 000 different images in 2.5 million grains.

So suppose we can get 30 000 frames at 100dpi on a disc in the form of rainbow holograms. How many pixels is that in total? $30000 \times \pi \times (\frac{1}{2}7.6\text{cm})^2$ is an effective area of 136m², or 2 gigapixels. That works out to 465 gigabits per square meter. This is not nearly as good as the current Rosetta disc's microscopy approach, although it would still make the disc quite a bit denser than a book.

(Is there a limit in angular resolution resulting from the light itself? There must be: the light path length from the source to the hologram to the eye needs to differ by at least half a wavelength for a pixel to go

from light to dark, right? At (half of) 500nm over a hundredth of an inch, we have an angle of about 1 milliradian, which is actually a much more stringent limit than the slit width thing: it would limit you to about 3100 frames.)

Non-rainbow holograms could get some three orders of magnitude higher storage density by using both dimensions, but as far as I know, can only be read under laser illumination. Perhaps you could get 500 terabits per square meter that way.

As for manufacturing, the common approach to manufacturing rainbow holograms is embossing: you nickel-plate the master hologram, then squish molten plastic against it, then take the now-shaped plastic off and plate it with something — typically aluminum, but nickel would work too, and probably last longer. (Remember, we're talking about millennia here.) Ideally you'd use entirely amorphous materials (glasses) to avoid noise due to crystal grain boundaries, entirely covalently bonded materials like SiO_2 for chemical stability and hardness, and thoroughly oxidized materials like SiO_2 for chemical stability and resistance to combustion. But producing a sufficiently shiny surface on glass (whether pure fused silica, borosilicate, soda-lime glass, or something else) might prove too difficult.

If your manufacturing cost is low enough and your material flexible and robust enough, then instead of a single disc, you could produce a codex. If you're storing some 500 gigabits per square meter, a hardback book consisting of 400 A5-sized pages could store 6 terabits in human-readable form.

For the case of holographic archival discs, you probably want to compute the original image, if possible, rather than attempting to produce it with thousands of interfering optical wavefronts. Bill Beaty's scratch holograms provide a practical way of doing this that wouldn't require computing trillions of grains in order to store billions of pixels. They also, like rainbow holograms, limit you to one direction of parallax motion.

Topics

Single-point incremental forming of aluminum foil

Kragen Javier Sitaker, 2019-03-11 (updated 2019-06-10) (14 minutes)

I'm sitting in a café with a wall covered in embossed tiles, which I think are plastic painted to look like brass with a heavy patina. Some parts of the tiles have a sort of leather pattern, while other parts have raised floral designs and the like.

Despite the enthusiasm for 3-D printing via FDM, most of the things people make with it have very limited mechanical properties; instead they primarily are of interest because of their appearance — the infill is commonly a honeycomb or cross pattern with 60% to 90% empty space inside the outer shell. But appearance is mostly a function of the surface — entirely so, in the case of opaque objects. So perhaps a process that shapes only a surface may be of interest.

Single-point incremental forming, or "SPIF", is an emerging industrial process for rapid prototyping, which is to say it barely works at all and it's slow, but flexible. The idea is that you clamp a metal sheet around the edges and poke it with a stick; in cross section in ASCII art:

```
|clamp|     | |           |clamp|
-------      U            -------
---------------------------------
-------                   -------
|clamp|                   |clamp|
```

Where you poke it, you make a divot by stretching ("forming") the metal out the other side of the sheet, and if you drag the stick along the surface without letting up the pressure, you can make a groove. (Generally you use a rounded-end tool and you lubricate and/or rotate it to keep it from sticking.) You can see the same process in action if you try to write on paper with a ballpoint pen with no ink.

Where this gets interesting is that if you make the groove, say, circular, the metal inside the groove has nothing pulling it up, so it gets pulled down to the level of the bottom of the groove, making a flat circular depression. Then you can do the process again inside there to make the depression deeper, and so on. The same considerations of thinning, wrinkling, springback, and work-hardening that apply to deep drawing apply here, but since SPIF is mostly used for one-offs where you can't afford the investment to make stamping dies (though sometimes people use it with dies too) you need to use FEM software to simulate them.

(The low-tech approach to this whole thing is hammering sheet metal into shape over a form.)

I was thinking that maybe you could do this with aluminum foil with really minimal force. Aluminum foil also has the advantage that it's thin enough that you can easily cut it with a spark; and, if you need to anneal it, you can be sure of a uniform temperature through

the thickness of the material, and since aluminum doesn't need a soak time at high temperatures to anneal it, it can be done very quickly. (This requires heating to close to the melting point; people sometimes use burnoff of carbon black on aluminum to indicate that it's reached the right temperature.)

SPIF forming of a depression is normally done from the outside in, with an empty space under the workpiece, which is clamped only at the edges; this requires workpiece to transmit the load from the forming tool back all the way to the edges. A possible improvement may be to do an initial forming step on a resilient backing, such as a sheet of rubber, or a disposable one, such as a sheet of cardboard, creating many parallel grooves with a bit of separation between them; this produces an accordion-fashion section which can then be unfolded with much less force once the backing is removd.

This is not very far from what you might do with a beading machine to raise a rib to stiffen a sheet-metal surface, the difference being that you're raising a lot of parallel ribs next to each other, and with the objective of selectively increasing compliance rather than decreasing it. The work-hardening of the metal obviously works against you here.

Electrotyping and molding

Electrotyping may be a particularly appealing next process step to apply, allowing the soft, easily melted aluminum to give its precisely-dimensioned form to metals like copper, brass, bronze, nickel, chromium, gold, or silver; I'm not sure how well electrotyped coatings will adhere to the aluminum's passivating oxide layer, and I don't think it's likely for cathodic reduction to eliminate aluminum oxide in water, but if it doesn't adhere well, that merely facilitates the removal of the aluminum for disposal.

Electrotyping is difficult to apply to alloys (whichever metal is easier to reduce tends to crowd out the other metals), although there are some processes that can electrodeposit some brasses and bronzes. But, by the same token, the electrodeposited metal may work well as a shell to fill with a harder alloy in the molten state. The easiest combination is presumably a copper or nickel shell filled with type-metal, which (as discussed in Flux deposition for 3-D printing in glass and metals (p. 1366) and Hot oil cutter (p. 3287)) melts at 241°, does not shrink upon solidifying, and does not dissolve iron or steel; I am guessing that it will not dissolve nickel either. It probably dissolves copper pretty well, since lead–tin solder does, but probably not very deeply in the time before it cools at the surface of the mold.

Copper doesn't melt until 1084°, and nickel doesn't melt until 1455°, meaning that in theory you could fill shells of them with materials of much higher melting points than the aluminum itself can survive — including, in the case of nickel, cast iron (see Flux deposition for 3-D printing in glass and metals (p. 1366)), which will definitely dissolve it. Dimensional precision may suffer from thermal contraction, although I seem to recall that cast iron in particular shares type-metal's happy property of neither expanding nor contracting upon solidification.

Various pot-metal alloys (Zamak, etc.) are also an option; in theory Zamak 2 melts at only 379–390° but has a yield strength of 361 MPa, better than ASTM A36 steel's 290 MPa (according to

Heckballs: a laser-cuttable MDF set of building blocks (p. 2782)) or 250 MPa (according to Wikipedia's A36 steel article), though the steel beats it at ultimate tensile strength. I think Zamak is more expensive than cast iron, though.

Alternatively, you might be able to cast directly into the aluminum foil, as long as the pressure isn't too high. According to Filling hollow FDM things with other materials (p. 2119), pure aluminum doesn't melt until 660°, and the alloy used for aluminum foil isn't too much below that. Casting will clearly involve some pressure that can deform the aluminum, but this may be small enough not to matter. Solder, type metal, ABS, PLA, lead, hard candy (sugar syrup), paraffin wax, thermoset resins such as silicone or PMMA, and Zamak should all be moldable directly in aluminum foil; some of these will stick permanently to it, but probably all of them will stick to it well enough to require deforming the aluminum foil irreversibly (plastically) for demolding.

Casting crystalline materials involves substantial loss of surface detail from crystallization, both from crystal grain growth directly distorting the surface and from the contraction or expansion that usually accompanies the phase transition. Glasses such as hard candy avoid this problem.

Promising qualitative results from a simple manual casting experiment

I deformed some aluminum foil (thin, about 10 μm†, times or divided by 2) with a ballpoint pen on top of some paper by drawing a sort of face on it, washed it with 96% ethanol, laid it on a glazed porcelain floor, and then melted some 60–40 tin–lead solder (183°–190°) on top of it, using a shitty non-temperature-controlled soldering iron. The solder was able to pick up most of the contours of the drawing accurately, but something black matter was stuck to the bottom of the solder and screwed up the impression. I think it came from rosin contamination of the tool that carbonized instead of boiling off, but it could also be a chunk of metal oxide from the tool, either copper or iron. The foil peeled easily off the solidified solder. The solder surface showed solder's usual dull finish, despite the bright finish of the aluminum foil that had molded it; I think this is probably due to wrinkling as the surface cooled before the interior and would be eliminated by adding antimony, as in type metal.

An earlier attempt using the same blob of solder was much less successful, because the bottom of the solder blob was full of bubbles, obliterating most of the submillimeter-scale contours I was trying to pick up. These might have resulted from oil contamination of the aluminum foil (e.g., from my hands) or from the rosin in the solder, which does volatilize significantly at the temperatures needed to melt the solder, let alone the higher temperatures the tool was presumably reaching.

Specks of the black material were also evident on the upper surface of the solder, and upon breaking it with pliers, apparently also in its interior. I was hoping to draw some sort of conclusion about its density or solder-wettability, but I don't know what to make of this.

The solder blob was about 1.5 mm thick. If the solder's density is 9 g/cc, which I haven't measured but should be about right (see A

phase-change soldering iron (p. 2270)) this works out to about 130 Pa of pressure on the aluminum, or 0.02 psi in folk units, which should put fairly small forces on the aluminum foil, not deforming it much except in very flat areas; the total weight of the solder over a square centimeter would be about 1.4 grams. However, I melted the solder directly on the surface by resting the soldering iron on top of it, so in this case the aluminum foil experienced much greater forces than the weight of the solder. Pouring already molten solder would avoid this problem

† I folded a piece of it in half 8 times and measured it at almost 2 mm with my caliper. If I knew where to find the battery for the caliper I'd be able to give a more precise measurement, but I don't think it's that critical — I'd expect anything from 10 μm to 125 μm (a Coke can I had lying around, though that includes the epoxy and paint) to give roughly similar results.

Exponential typing and Gutenberg

Suppose you use SPIF to cast some simple stamps in, say, solder, Zamak, or type metal, some 100 times thicker than the aluminum foil, and thus 100 to 10'000 times as rigid, depending on the deformation mode you're considering. (Well, before you account for the difference in moduli of rigidity between the aluminum and the other materials, but those are small by comparison.) You can then use these stamps to stamp or incrementally form more foil, which you can then use to cast or electroform further shapes, either in positive or in negative.

To take a simple example, you could use the SPIF process to form stems, serifs, and bowls of letterforms in foil, which you can then cast into stamps, which you can then stamp into an aluminum foil matrix to form letters, which you can then cast or electrotype into a font; the additional step of making up a stereotype for a page of text allows you to reuse the same letters for each page without losing the ability to reprint previous pages. A recent analysis of Gutenberg's books suggests he probably used such a process, although of course he wasn't using aluminum foil for the expendable soft matrix. (Likely materials that occur to me include clay bodies, plaster of Paris, and gold foil; I think papier-mâché, as used for stereotypes, wouldn't have worked for reasons of temperature limitations. Later printers used a thick, durable, work-hardening copper matrix instead, cutting a single punch from steel for each letter.)

But other shapes might also work reasonably. For example, you could use a tiny spherical SPIF tool to make a mold for a larger spherical SPIF tool; you could quite reasonably make stamping dies (at least for soft metals and slow stamping) to put ribs in things, stamping the ribs themselves with a series of stamps from torus-shaped rib positive and negative dies; and even features like countersunk holes to cast into thin sheets might be feasible. To cast half of a screw thread, you could possibly use a stamp with the form of a single turn of the screw thread, repeatedly; two such molds clamped together could cast an entire screw.

In cases where the cast material releases easily from the aluminum-foil matrix, you may be able to cast both a positive and a negative die simultaneously from the same sheet, thus ensuring a fairly close fit even if the sheet deforms significantly during molding.

Topics

A simple content-addressable storage-server protocol

Kragen Javier Sitaker, 2015-09-03 (3 minutes)

Here's a simple network protocol that provides a reasonably secure and reasonably efficient content-addressable storage system.

Basic operations

The fundamental operations provided by the protocol are PUT and GET. PUT has a single argument, which is an arbitrary blob — although in this case I am limiting blob sizes to those that can be passed in a single internet datagram; GET also has a single argument, which is the cryptographic hash of a single blob.

(If you have several servers providing this service, you can arrange them into a DHT as you see fit; the servers don't have to know about it.)

Adding security

Both of these are, in some sense, unauthenticated operations: if your data isn't public, you can encrypt it with a secret key before you store it in the public content-addressable store, and so attackers will gain no useful information by retrieving it. However, they are subject to denial-of-service attacks. First, since the server presumably has limited storage, the amount of data that can be successfully PUT to it and GETted back later is finite; second, since the server has limited bandwidth, the amount of data that can be PUT or GETted from it in a second is finite. Finally, GET could potentially be a tool in a source-spoofed DoS amplification attack on someone else: if the GET request is 20 bytes, but the resulting blob is 20000 bytes, you can multiply your bandwidth by 1000 by sending a stream of source-IP-spoofed GET requests to an innocent server, which will faithfully direct a firehose of traffic at your chosen victim.

I think we can add DoS-resistance to the protocol with hashcash (or other forms of making requests expensive — reciprocity IOUs, Bitcoin payments, etc.), and amplification-resistance with mandatory request padding.

Protocol proposal

So I propose an additional NO answer, and reusing the PUT packet to send answers. A typical PUT dialog looks like this:

client: PUT nonce=b, blob=a
server: NO nonce=c, difficulty=n
client: PUT nonce=d, blob=a (such that HMAC(c, a || d)[0:n] == 0)
server: PUT nonce=d, blob=a

And a typical GET dialog:

client: GET hash=e, nonce=f
server: NO nonce=c, difficulty=n, length=m (m = length(a))
client: GET hash=e, nonce=g, pad=h (such that HMAC(c, e || g)[0:n] == 0, and length(h) >= m)
server: PUT nonce=g, blob=a

The server is free to share the nonce between clients, or not, and change it as often as it likes (potentially requiring slow clients to retry requests).

Topics

- Programming (p. 3658) (286 notes)
- Protocols (p. 3668) (21 notes)
- Security (p. 3701) (9 notes)
- REpresentational State Transfer (p. 3684) (8 notes)
- Content addressable (p. 3389) (8 notes)
- Proof of work

Distributed computing environment

Kragen Javier Sitaker, 2017-07-19 (8 minutes)

I want a new computing environment for reading writing, including reading, writing, and remixing computational models, that cleanly virtualizes the computer.

The ideal is something like this.

I sit down at a random untrusted computer and load up the virtual machine, as a web page or whatever, and type in an unspoofable identifier of the mutable information space I want to look at, maybe something like "lotus-marc-fix-kev-homes-vii-taps-car" or "dnbwalalhcsmgsnsspdyq". Immediately I see a display of that information space, loaded from a decentralized network of servers, and an unspoofable identifier for my session, which I write down on paper. I can start writing down my thoughts, using the keyboard or whatever, with an instantly accessible and reliably responsive user interface, taking advantage of the keyboard, mouse, tablet, webcam, or other peripherals available on the untrusted computer. As I assemble computational models, they are executed immediately on this local machine, but everything I create is snapshotted periodically and propagated out to the network within less than a minute, retrievable later by the session identifier. All of the information I see, too, is permanently snapshotted by the network, so that I can come back and refer to it later. At the end of my session, I generate an unspoofable identifier for the state at the end of the session, and I write it, too, down on paper.

Later, using a trusted but much less capable computer, such as my cellphone, I retrieve the data for the session identifier; I can see everything I created during the session, and it is certain to work compatibly there. I can look at the things I created, and if they are good, I can update my published information space with them.

Compatibility into the far future and past is guaranteed by the design of the system: if the needed data is preserved, a simple virtual machine suffices to bring it back to life. In addition to recording decoders in a format that can be run efficiently on a simple virtual machine, the original data itself is recorded in simple, general formats like ASCII text, STL files, CSV files, and uncompressed bitmaps, simplifying putting it to new uses in the future. Code and other original data have their integrity checked using a Merkle DAG; the hash of mutable data is authenticated by using a public-key signature.

Code is deterministic, so that executing the same code with the same data as input will always produce the same output data. Non-original data produced by executing code with data as input is thus cacheable in the same content-addressable store decentralized network; cache services store (code hash, input data hash, output data hash) tuples, so that if you trust everyone with write access to a cache, you can omit redundant computation. Trees of such tuples are published in the same network, its hash being authenticated by the public-key signature of the caching service itself.

Decoders and editors for crucial formats are accompanied by formal

proofs of correctness, of worst-case execution time, and of worst-case execution space. This is achieved by writing them originally in a formally tractable source language, making the proofs tractable at the source-code level, and then mechanically translating the proofs into proofs at the level of the virtual-machine code, which can be machine-checked without reference to the correctness of the compiler.

In addition to the content-hash-addressable immutable blob data store and the public-key-hash-addressable mutable data store, the system contains two additional communication services: one for reliably passing streams of authenticated messages between mutually consenting communicating correspondents identified by per-connection public keys, and an unauthenticated and unreliable publish-subscribe system for initially establishing contact between parties that don't have a pre-existing relationship.

Interactive user interfaces are specified by a state transition function, which takes as input the current user interface state and an event (which may be a keystroke, a multitouch notification, a webcam frame, a mouse motion, a network message, a timeout, a random number, or a few other things) and produces as output the new user interface state and a possibly empty set of actions to take, which may include updating a screen, playing a sound, scheduling a timeout, publishing a blob, publishing an update to mutable data, sending a message to a correspondent, publishing a message, subscribing to a topic, and requesting a random number. They are, it should be clear, composable from simpler user interfaces using the same interface or a simpler interface. For example, many games can perform adequately if given the controller state 30 or 60 times a second and producing a frame of output. XXX does the deterministic computation thing interfere with responsiveness? It would seem to make it impossible to update the rest of the screen while you're trying to deliver updates to a slow program.

Farming out computation to a cluster, if you trust it, is easily accomplished by sending the (code hash, input data hash) commands you want to execute to it, and arranging for the cluster coordinator to send you the final cacheable tuple when it finishes. For many uses, this also requires sharing a cache with the coordinator that you can trust not to leak secret data.

The virtual machine on which the code runs has this command-sending as a fundamental operation: any program running on it can submit (code, input) commands for execution, which may be transparently satisfied from cache, by local execution, or by farming out to a cluster. XXX Perhaps this shouldn't be a fundamental operation, both because you might want to change it and because it could make proofs of efficiency very difficult; but that just means that the computation producing an output needs to be able to delegate that responsibility to a future computation composed from commands it spawns, which might come to the same thing in the end.

Programs manipulate the hashes through handles that are opaque to them; rather than being contained in the code itself, they are listed in a manifest associated with the code. Similarly, plain data containing a reference to other plain data lists its hash in an associated manifest. The hashes in the manifest have associated types which restrict what

kinds of references can be made to them: an image transcluded in a hypermedia document has the type "dependency" associated with its hash in the manifest, while a hypertext link has the type "reference" associated with it.

These link types allow archive-auditing systems to verify that all of the dependencies of a document are included on an archival medium or stored in the network with an acceptable degree of redundancy, without having to understand not-yet-invented hypermedia data formats.

Computation-cache services can also publish the estimated cost to recompute derived data, based on how long it took last time. This aids cooperating data-cache services in choosing which data to throw away.

Topics

Interactive bandwidth

Kragen Javier Sitaker, 2017-08-03 (2 minutes)

The internet connection here is broken. I just used Android tethering to log in, read my mail, check a couple of chat channels, and answer some chat messages, for about ten minutes. At the end, ssh reported:

```
Transferred: sent 23048, received 60304 bytes, in 618.1 seconds
Bytes per second: sent 37.3, received 97.6
```

That is, on average over those 10 minutes, I was typing and otherwise sending data at "373 baud" and reading at "976 baud". The incoming number was, I think, reduced by about a factor of 2 by gzip compression.

This 500-bit-per-second range is interesting, because it's a fairly easy bandwidth to achieve under many circumstances. I think you should be able to do better than this with subliminal audio coding layered on top of a voice channel, for example, and even simple binary FSK coding at audio frequencies can achieve it pretty easily. It's about a factor of 20 less bandwidth than the "2G" GSM GPRS data connection I often used to read my email in 2001.

At times, today's connection was hard to use because of unpredictable, high latency. But high latency is not inherent to the low-bandwidth domain — indeed, often you can trade bandwidth off against latency, for example with data compression or more aggressive error-correction coding, but also in a more general way by having a wider choice of acceptable communication media.

Topics

• Communication (p. 3382) (19 notes)
• Networking (p. 3594) (7 notes)

Separating implementation, optimization, and proofs

Kragen Javier Sitaker, 2019-06-26 (updated 2019-07-22) (41 minutes)

One of the approaches the STEPS program wanted to try was to "separate implementation from optimization". I don't know that they ever got anywhere with that (certainly I never saw anything in their public reports that could be described that way) but I think it's a really interesting idea.

Conventional programming practice unashamedly mixes together the specification of what your program does with a lot of implementation choices about how to do it efficiently. Of the languages I've used, the one that is purest at describing just the semantics of the desired computation is perhaps Scheme, though it's in no sense the theoretical limit in that sense. What are the ways Scheme and other programming languages mix these things together?

(See also Generic programming with proofs, specification, refinement, and specialization (p. 958).)

Scheme and its discontents

Scheme is pretty abstract.

In Scheme you don't explicitly specify what type your variables are, when to deallocate things, how to represent enumerated choices in machine registers, whether you're calling a function or invoking a macro. You don't specify whether a call to a function will return to the callsite, through a saved continuation somewhere up the stack from the callsite, or through a saved continuation to somewhere on another stack. Iteration uses the same syntax as recursion. If you use lists or vectors for your records, you never have to declare record types, and you never have to say which record type a list or vector represents, although the built-in field accessors have names like `cadddr`.

But Scheme is relentlessly monomorphic, at least outside the numerical tower — the standard procedures for lists, for example, work only on lists, not arbitrary sequences. To index into a string or to index into a list, you use different procedures, so you can't write a procedure that works implicitly on either one. It's unapologetically eager (although continuations provide some escape from this, in that they allow a procedure to suspend evaluation until a result is ready; because it has the implicit variable capture semantics later adopted by nearly all modern programming languages, its closures also provide an escape valve, though also a rather syntactically clumsy one. Your record types must be explicitly defined (if your system implements SRFI 9 so that you have record types). And if you need both a function and its inverse, you must define them separately — more generally, given an n-ary relation, there exist 2^n functions from subsets of its columns to sets of values for the other columns, and in Scheme each of these must generally be reified as a separate procedure.

Schemaless records

By contrast, in Lua or JS, you can implicitly ("schemalessly") define a record type, with reasonable accessor syntax, just by using it; in Lua syntax:

```
dragStart = {x=ev.p.x, y=ev.p.y}
```

The schemaless Scheme equivalent is something like

```
(let ((drag-start (cons (x (p ev)) (y (p ev)))))
    …)
```

but then you have to access the `x` and `y` fields of `drag-start` with `car` and `cdr`, or define functions wrapping them (which might have to have longer names than `x` and `y`).

If you use a SRFI-9 declaration like this one in Scheme:

```
(define-record-type :point (make-point x y) point? (x get-x) (y get-y))
```

then you can instead write

```
(let ((drag-start (make-point (x (p ev)) (y (p ev)))))
    …)
```

and then you have `(get-x drag-start)` and `(get-y drag-start)` as you would want. But you still can't use the same accessors `x` and `y` for your new record type that were used for the fields of `(p ev)`; this is a specific case of a larger issue, which is that Scheme makes polymorphism cumbersome and often impossible.

Object-orientation or polymorphism

In Python, Smalltalk, or Ruby, you can generally define a new type that can be used in place of an existing type as an argument to a function or method, simply by implementing the interface that the function or method expects of that argument. Consider this very simple Python function from Dercuano:

```
def pluralize(noun, number):
    return noun if number == 1 else noun + 's'
```

This is intended for use with a string and an integer, returning a string, as you'd expect. A Scheme equivalent:

```
(define (pluralize noun number)
  (if (= number 1)
      noun
      (string-concatenate (list noun "s"))))
```

But all the Python version demands of the "integer" is that it be usefully comparable to 1, and all it demands of the "string" is that you can concatenate a string to it and get a value of the same type. So, for example, you can partially evaluate `pluralize` (and a class of similar functions) with respect to the `number` argument by passing the following `Template` object as an argument instead of the string:

```
class Template:
    def __radd__(self, other):
```

```
        return Template_cat(templatify(other), self)

    def __add__(self, other):
        return Template_cat(self, templatify(other))

    def of(self, value):
        return value

class Template_cat(Template):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def of(self, value):
        return self.a.of(value) + self.b.of(value)

class Template_k(Template):
    def __init__(self, value):
        self.value = value

    def of(self, value):
        return self.value

def templatify(val):
    return val if isinstance(val, Template) else Template_k(val)
```

This amounts to partial evaluation by way of abstract interpretation with an non-standard semantics, but it's done entirely at the level of ordinary code by passing an argument that implements an interface. To do something like this with Scheme, you have to step up to a metalevel and write a sort of compiler rather than just implementing metamethods.

So you could reasonably argue that the Python `pluralize` is a more abstract specification of a computation than the Scheme version.

Laziness

As many of us learned from SICP, although Scheme is eager, you can define lazy streams in Scheme by delaying evaluation with `lambda`:

```
(define (ints-from n) (lambda () (cons n (ints-from (+ n 1)))))
```

Or by using `delay`:

```
(define (ints-from-delay n) (delay (cons n (ints-from-delay (+ n 1)))))
```

But — related to the above issue about polymorphism — in standard Scheme, you cannot pass these lazy streams to things that expect regular lists. (A Scheme that provides implicit `force` would allow this; as I understand it, supporting this possibility is the reason `delay` and `force` are in the standard, but neither of the Schemes I tried it on has implicit `force`.) So a caller that expects a regular list as a return value is constraining the callee to produce it entirely before returning. And a callee that expects a regular list as an argument is constraining the caller to produce it entirely before the callee begins execution.

In mathematics, it's entirely commonplace to deal with infinite

sequences on equal terms (heh) with finite sequences, and even infinite sequences of infinite sequences, etc. The decision to compute only a finite subset of an infinite sequence is, in many cases, a sort of optimization decision — computing too many terms is relatively harmless, consuming only extra time and memory, and computing all the terms would take infinite time and therefore require some kind of hypercomputation.

This kind of thing frequently results in dual implementations: an eager implementation for when all the results are needed (more convenient to handle in Scheme and in most languages) and a lazy one for when it's important to produce results lazily. Usually these are provided through separate operations (functions, methods, or operators), though in some cases, it's just different options to the same operation; for example, in the perlop documentation for the pattern-match operator `m//`:

> The "/g" modifier specifies global pattern matching — that is, matching as many times as possible within the string. How it behaves depends on the context. In list context, it returns a list of the substrings matched by any capturing parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.
>
> In scalar context, each execution of "m//g" finds the next match, returning true if it matches, and false if there is no further match. The position after the last match can be read or set using the "pos()" function; see "pos" in perlfunc.

By contrast, in Python, you can write a generator function, which creates a generator object when initially invoked; both the generator object and the builtin list object implement the builtin iterator protocol, so things that just use the iterator protocol (by, for example, iterating or converting to a tuple) will not notice the difference between a list and a generator.

In Dercuano, for example, this method is a generator; it generates a lazy sequence of (subject, verb, object) triples that form the main data store of the Dercuano application:

```
def load_triples(filename):
    with open(filename) as f:
        last_subj = None
        for line in f:
            if not line.strip():
                continue
            fields = tuple(unquote(field.replace('+', '%20'))
                           for field in line.split())
            if line[0] in string.whitespace:
                fields = (last_subj,) + fields
            for fn in fields[2:]:
                yield (fields[0], fields[1], fn)
            last_subj = fields[0]
```

This parses some vaguely RDF-N3-like text saying things like this:

```
7805-6-volts written 2019-06-08
    updated 2019-06-10
    concerns electronics pricing

array-intervals written 2019-06-23
    concerns math intervals arrays apl python c programming algorithms numpy
```

Its return value is passed to this function, which generates some in-memory hash tables for quick access:

```
def as_relations(triples):
    relations = {}
    for subj, verb, obj in triples:
        if verb not in relations:
            relations[verb] = relation.Relation()
        relations[verb].put(subj, obj)

    # Precompute inverse relations eagerly
    for verb in relations:
        ~relations[verb]

    return relations
```

Right now, the execution of `as_relations` is interleaved with that of `load_triples`. If `load_triples` is changed to build up a list of 3-tuples and return it, or if `load_triples`'s caller does so by invoking `list()` on its return value before passing it to `as_relations`, the code of `as_relations` continues to produce the same results, but the file access is no longer interleaved with the hashing, and an additional big temporary data list is built up in memory. As it turns out, this makes it run slightly faster. (So I made the change.)

The ability to hide optimization decisions behind interfaces like this is powerful, but I'm kind of overselling Python's implementation of the iterator interface here. If you iterate over a list a second time, you get the same items, unless you've changed it; but iterators in Python are single-pass, and so you cannot iterate over a generator a second time. Worse, if you attempt to do so, you silently get no items, rather than an error message. And, because Python's semantics have pervasive mutability, it's easy to write a generator that unintentionally produces a different sequence of values depending on exactly when the iteration happens. (In the above example, this could happen if the contents of the file changed, which is a problem that goes beyond Python; see Immutability-based filesystems: interfaces, problems, and benefits (p. 1672).)

Laziness is useful for a few different reasons: to avoid building up big in-memory temporary data structures that will just be thrown away again; to do computations in finite time involving infinite sequences like the continued-fraction expansion of φ or π; and to provide partial results sooner from things that can take a long time to finish, including jobs that use heavy computation, I/O, and remote procedure calls, which are a sort of I/O.

(Haskell is the champion of laziness, avoiding both Python's repeated-iteration problem and Python's implicit-mutation-dependency problem, but I don't know Haskell, so I don't have a good example to hand.)

N-ary relations and relational programming

Consider this Prolog "predicate", which is to say, n-ary relation:

```
x(7805-6-volts, written, 2019-06-08).
x(7805-6-volts, updated, 2019-06-10).
x(7805-6-volts, concerns, electronics).
```

```
x(7805-6-volts, concerns, pricing).
x(array-intervals, written, 2019-06-23).
x(array-intervals, updated, 2019-06-10).
x(array-intervals, concerns, math).
x(array-intervals, concerns, electronics).
```

We can interactively query the topics covered by How to get 6 volts out of a 7805, and why you shouldn't (p. 537) as follows:

```
?- x(7805-6-volts, concerns, Topic).
Topic = electronics ;
Topic = pricing
```

So we can think of `x` as a function which, when executed, gives the topics for How to get 6 volts out of a 7805, and why you shouldn't (p. 537). But we could make the note name an input variable and think of it as a function from note names to (sets of) topics:

```
?- Note = 7805-6-volts, x(Note, concerns, Topic).
Note = 7805-6-volts,
Topic = electronics ;
Note = 7805-6-volts,
Topic = pricing ;
false.
```

(Calling back to the previous topic, this sequence is lazy; you can say `Note = 7805-6-volts, findall(Topic, x(Note, concerns, Topic), L).` to get the whole list at once.)

Also, though, `x` is the inverse function, from topics to note names:

```
?- Topic = electronics, x(Note, concerns, Topic).
Topic = electronics,
Note = 7805-6-volts ;
Topic = electronics,
Note = array-intervals.
```

But we can also invoke `x` as a function to return a list of all note–topic pairs:

```
?- x(Note, concerns, Topic).
Note = 7805-6-volts,
Topic = electronics ;
Note = 7805-6-volts,
Topic = pricing ;
Note = array-intervals,
Topic = math ;
Note = array-intervals,
Topic = electronics.
```

Or a function from note names to last-updated dates:

```
?- Note = array-intervals, x(Note, updated, Date).
Note = array-intervals,
Date = 2019-6-10.
```

If we compose `x` with itself, we can even, for example, find all the pairs of notes last updated on the same date:

```
?- x(Note1, updated, Date), x(Note2, updated, Date).
Note1 = Note2, Note2 = 7805-6-volts,
Date = 2019-6-10 ;
Note1 = 7805-6-volts,
Date = 2019-6-10,
Note2 = array-intervals ;
Note1 = array-intervals,
Date = 2019-6-10,
Note2 = 7805-6-volts ;
Note1 = Note2, Note2 = array-intervals,
Date = 2019-6-10.
```

But even without composing `x` with itself or selecting from it, we can construct eight different functions from it by declaring some of its columns to be inputs and others to be outputs; by currying or partially evaluating those functions with respect to some subset of their input arguments (in the relational view, selecting a subset of rows from the relation), we can get a much larger set of functions out of it. As written, there are four functions with the first column fixed to How to get 6 volts out of a 7805, and why you shouldn't (p. 537) and four more with it fixed to Reducing the cost of self-verifying arithmetic with array operations (p. 2205); four each with the second column fixed to each of its three values; and four each (much less interesting) functions with the third column fixed to each of its six values. This is 44 more functions, plus the 8 uncurried ones, for a total of 52, plus the functions with more than one curried argument (which are less interesting).

Defining 52+ functions is not too bad for 8 lines of code, and you won't have to go searching through a many-page API document to figure out how to use these 52+ functions, either.

As I said before, though, Prolog hardwires a depth-first search strategy, which fails to terminate for many searches, and is exponentially inefficient for many others. So the extent to which you can do this kind of thing in Prolog directly is fairly limited. The usual example is `append`, which is a built-in Prolog function; we can ask SWI-Prolog for a listing of it as follows:

```
?- listing(append/3).
lists:append([], A, A).
lists:append([A|B], C, [A|D]) :-
        append(B, C, D).

true.
```

These two or three lines of code define the list-append relation among lists; thinking of it in the usual way as a way to append two lists, we can call it to append two lists:

```
?- append([h,e,l,l,o], [v,e,n,u,s], L).
L = [h, e, l, l, o, v, e, n, u|...].
```

But these two lines of code define eight functions, just like the `x/3`

predicate defined above. We can ask for all pairs of lists that append to another list:

```
?- append(A, B, C).
A = [],
B = C ;
A = [_G249],
C = [_G249|B] ;
A = [_G249, _G255],
C = [_G249, _G255|B]
```

We can ask for every pair of lists that can be appended to form a given list:

```
?- append(A, B, [v,e,n,u,s]).
A = [],
B = [v, e, n, u, s] ;
A = [v],
B = [e, n, u, s] ;
A = [v, e],
B = [n, u, s] ;
A = [v, e, n],
B = [u, s] ;
A = [v, e, n, u],
B = [s] ;
A = [v, e, n, u, s],
B = [] ;
false.
```

We can ask if one list is a prefix of another list (and, implicitly, what's left over):

```
?- append([h,e,l,l], X, [h,e,l,l,o]).
X = [o].

?- append([h,e,l,l], X, [v,e,n,u,s]).
false.
```

Or, more interestingly, a suffix:

```
?- append(Hello, [n,u,s], [v,e,n,u,s]).
Hello = [v, e] ;
false.
```

That's a lot of functionality for two lines of code!

You can define Scheme's standard `append` function in a precisely analogous way:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b))))
```

But you can't run a Scheme procedure "backwards" in the way the above Prolog predicates are being used, so if you want starts-with and

ends-with predicates, much less lists of sublists, you need to program those separately.

This kind of thing is a really tempting pointer to what Will Byrd calls "relational programming", where instead of programming *procedures* or *functions* (or, as *Total Functional Programming* points out, partial functions) we program *relations*. Modern Prolog systems are doing some work with tabling and integer programming to expand the scope of this kind of thing, but they're incurably handicapped by Prolog's depth-first evaluation model. Byrd and Kiselyov's system miniKANREN is, like the database query mini-language in SICP, an exploration of making this kind of thing much more general than it is in Prolog. Its second most impressive feat is that, given a relational specification of an interpreter, it was able to deduce a self-reproducing program, a quine, for that interpreter. (Its most impressive feat is that Chung-Chieh Shan and Oleg Kiselyov extended it to support probabilistic programming five years before anyone else realized probabilistic programming was a useful thing to do.)

As I understand it — and I may have this part wrong — miniKANREN uses a sort of breadth-first search strategy which is guaranteed to find a solution if one exists, but, like Prolog's strategy, is not guaranteed to terminate. My intuition is that this means that for many problems — in some sense, most problems — it will take exponentially longer than a program that always knows the right choice to make, or even a version of miniKANREN that knows the right choice to make more often, or gets to it sooner. This suggests that some kind of optimization-hint information could yield an exponential speedup, so adding some kind of optimization to your miniKANREN program (whether separated or, as is traditional, mixed into it) could yield big wins.

Naturally, Byrd and Kiselyov implemented miniKANREN in Scheme, although Clojure also includes a version of it.

From a certain point of view, the Scheme `append` procedure is an optimized version of the Prolog `append/3` predicate: it's a version of the predicate specialized to the case where the first two arguments are ground and the third argument is a free variable.

Irreversible computation

Consider this Python implementation of the method of secants (see Using the method of secants for general optimization (p. 1773), and note that these implementations tend to report success by dividing by zero):

```
def sec(f, x0, x1, e):
    y = f(x0), f(x1)

    while abs(y[1]) > e:
        x0, x1 = x1, x1 - y[1]*(x1 - x0)/(y[1] - y[0])
        y = y[1], f(x1)

    return x1
```

Here is a Scheme equivalent, although it may not be the best way to express the algorithm in Scheme:

```
(define (sec f x0 x1 e)
  (letrec ((loop
            (lambda (y0 y1 x0 x1)
              (if (> (abs y1) e)
                  (let ((xn (- x1 (/ (* y1 (- x1 x0))
                                     (- y1 y0)))))
                    (loop y1 (f xn) x1 xn))
                  x1))))
    (loop (f x0) (f x1) x0 x1)))
```

If it terminates, this approximates a root of the function `f` by finding the intersection point of the X-axis and the secant line through (x_{i-1}, f(x_{i-1})) and (x_i, f(x_i)); in the usual cases, this is a slightly more efficient approximation of Newton–Raphson iteration, since it converges with degree φ to Newton's 2, but only requires a single computation of $y(x)$ per iteration, rather than separate computations of $y(x)$ and $\dot{y}(x)$. (In some cases, not needing to compute the derivative is also useful, although if the derivative doesn't actually exist, this method may not converge.)

The thing I want to point out here is that, assuming floating-point, this procedure computes in constant space, but by virtue of doing so, it erases all the intermediate values of `x1` and `y`. If you just want the root, that may be okay, but if you want to analyze the performance of the algorithm, it may be useful to save those intermediate results.

Related to the previous item about laziness — in this form, the algorithm fails to terminate at times — notably when you give it real starting points for a function whose roots are complex, such as `sec(lambda x: x**2 + 2, 0, 1.0, .0000001)`, although it works fine with other starting points such as `sec(lambda x: x**2 + 2, 0, 1.0 + 1.0j, .0000001)`. Even if it does manage to terminate, it can run for many iterations, and in some applications it would be best to terminate its execution early, for example if a concurrent search from a different starting point finds a solution.

If you sometimes want the constant-space behavior and sometimes want the whole sequence of values or guaranteed termination, most programming languages require you to restructure the procedure as a generator of a lazy sequence, so that a concurrent consumer has the opportunity to discard its intermediate values or abort the iteration. For example:

```
def sec_seq(f, x0, x1):
    y = f(x0), f(x1)

    while True:
        yield x1
        x0, x1 = x1, x1 - y[1]*(x1 - x0)/(y[1] - y[0])
        y = y[1], f(x1)
```

This allows you to, for example, take the 10th value, discarding all previous values, to compute the first 10 values, or to interleave the iterative computation in other ways:

```
>>> list(itertools.islice(sec_seq(lambda x: x**2 + 2, 0, 1.0 + 1.0j), 9, 10))
[(-2.2572481470524732e-21+1.4142135623730951j)]
>>> list(itertools.islice(sec_seq(lambda x: x**2 + 2, 0, 1.0 + 1.0j), 10))
```

```
[(1+1j),
 (-1+1j),
 2j,
 (-0.2+1.4j),
 (-0.03448275862068967+1.4137931034482758j),
 (1.3877787807814457e-17+1.411764705882353j),
 (2.9885538387977862e-05+1.4142135620573204j),
 (-2.5897366406179418e-08+1.4142135623730947j),
 (-8.007108329513308e-22+1.4142135623733687j),
 (-2.257248147052473e-21+1.4142135623730951j)]
```

The original Python and Scheme code is seen to have mixed together the space optimization of discarding the intermediate values with the implementation of the underlying algorithm. Scheme is in some sense often more abstract than Python with regard to this sort of irreversibility, in that you write this loop with a call whose tail position the Scheme system is supposed to recognize, rather than explicitly overwriting variables with new values; but, on a standard Scheme system, you can't construct some kind of context where executing the original Scheme example above gives you access to all of the intermediate values. And in fact the surgery needed to convert the Scheme version of the algorithm to optionally produce the whole sequence of values, interleaved with the execution of the consumer, is more extensive than on the Python version.

(The situation is somewhat worse even than this suggests: in both Python and Scheme, if the arguments are arbitrary-precision rational objects (or, in some Schemes, integers), the algorithm is not constant-space, and the Scheme standard more or less explicitly permits garbage collection to be broken, and Scheme space behavior depends in most cases on garbage collection.)

But we see that Python and Scheme *permit* us to write this algorithm in such a way that the invoking context can preserve the intermediate values or discard them, as it wishes. (Doing this in general requires using FP-persistent data structures and avoiding mutation entirely, which is totally impractical in Python, and pretty hard to do without sacrificing efficiency in Scheme.) But what would an algorithm specification language look like that entirely omitted erasure?

Euclid's gcd algorithm

The first algorithm presented in an algorithms course is often Euclid's algorithm for computing greatest common divisors. Consider this Python implementation:

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

Or its Scheme equivalent, although the Scheme standard already provides this function:

```
(define (gcd a b)
  (if (= b 0)
      a
```

```
    (gcd b (remainder a b))))
```

Like the method-of-secants code, this computes a sequence of intermediate values before arriving at its final answer, and these intermediate values are lost. (However, at least for the usual data types, in this case termination is guaranteed.) It turns out that, as explained in Using Aryabhata's pulverizer algorithm to calculate multiplicative inverses in prime Galois fields and other multiplicative groups (p. 2255), the lost values are actually very useful; you can use them to efficiently compute the inverses of the elements of a multiplicative group or find that no such inverses exist, because they allow you to compute not only $g = \gcd(a, b)$ but also coefficients s and t such that $sa + tb = g$. The easiest way to do this in Python is, first, to restructure the algorithm as a recursive algorithm like the Scheme implementation:

```
def gcd(a, b):
    return a if b == 0 else gcd(b, a % b)
```

Second, to push the termination test down the stack one level:

```
def gcd(a, b):
    r = a % b
    if r == 0:
        return b

    return gcd(b, r)
```

Finally, to augment the return value with the additional information:

```
def egcd(a, b):
    q, r = divmod(a, b)      # r = a - qb
    if r == 0:
        return 0, 1, b       # 0a + 1b = b

    s, t, g = egcd(b, r)     # sb + tr = g
    assert s*b + t*r == g
    return t, s - t*q, g     # sb + t(a - qb) = g = ta + (s - tq)b
```

For example, if invoked as `egcd(81, 18)`, this returns (1, -4, 9), because $1 \cdot 81 - 4 \cdot 18 = 9$, their greatest common divisor;, so in $\mathbb{Z}/81\mathbb{Z}$, 18 has no multiplicative inverse. But, if invoked as `egcd(93, 26)`, it returns (7, -25, 1), because 93 and 26 are relatively prime, and $7 \cdot 93 - 25 \cdot 26 = 1$, so in $\mathbb{Z}/93\mathbb{Z}$, the multiplicative inverse of 26 is -25, which is to say, 68.

In a sense, the simpler functions above are optimizations of this one: they are iterative and they avoid doing work and allocating memory to calculate s and t when not needed. You can of course write the extended version and hope that your compiler is smart enough to detect this situation, but you will probably be disappointed, and moreover you will probably have no way of knowing. (Even if you grovel over assembly listings to verify it once, you likely won't notice if future change to the code break the optimization.)

Wouldn't it be better if you could write the extended version and

assert that, at a given callsite, the algorithm used should be iterative? If the compilation failed because that assertion couldn't be satisfied, wouldn't it be good to be able to explain to the compiler how to make the optimization and what assumptions justified the optimization, and have the compiler search for a counterexample for the compilation error message if your assumptions were wrong, or at least suggest a lemma you could try to give it a proof for?

Mixing together optimization, implementation, and proof

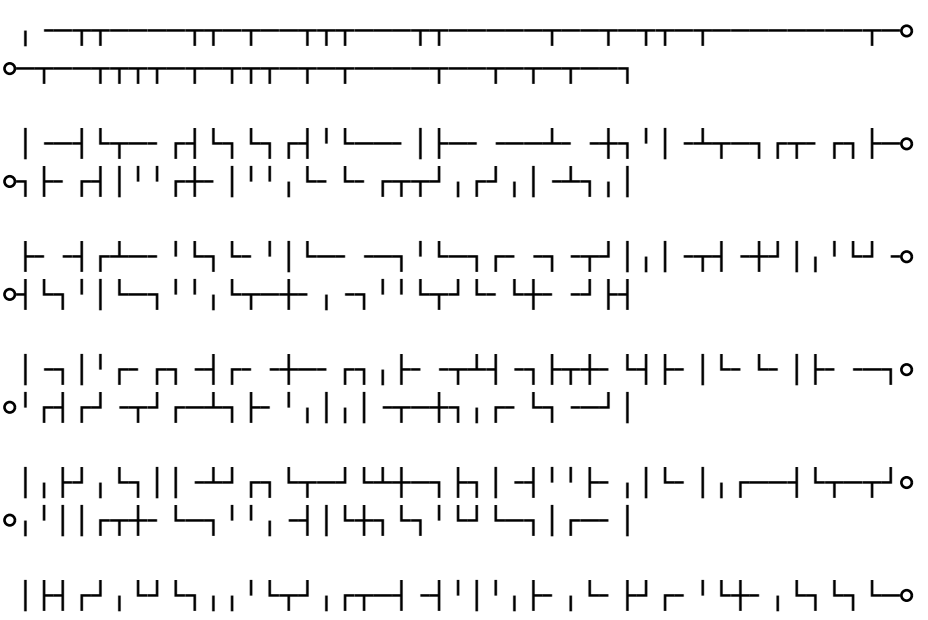
So we've seen particular ways in which even in Scheme we must entangle optimization with implementation, but in more mainstream programming languages, the situation is even more extreme.

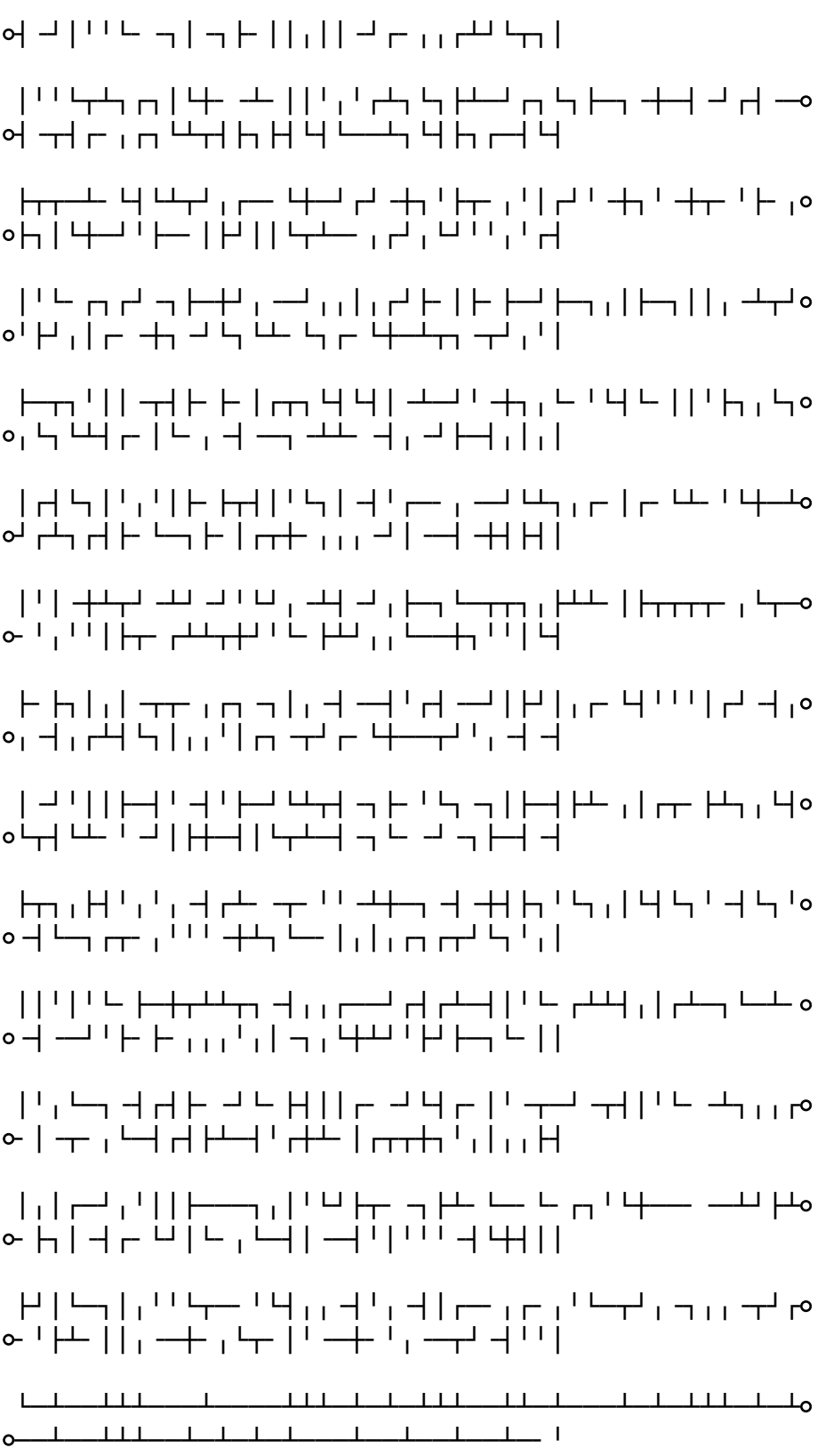
Consider this Golang code from my maze program:

```
func (m Maze) Show(output io.Writer) {
	h, v := m.Horizontal, m.Vertical

	for r := 0; r < len(h); r++ {
		s := []byte{}
		for c := 0; c < len(v[0]); c++ {
			k := context{
				up:    v[r][c],
				down:  v[r+1][c],
				left:  h[r][c],
				right: h[r][c+1],
			}
			s = append(s, glyphs[k]...)
		}
		output.Write(append(s, '\n'))
	}
}
```

Using a maze previously generated by the rest of the program with Kruskal's algorithm, this produces output like the following maze, which has one path through it:





If you translate this function directly to Python, it's about half as much code:

```
def show(self, output):
    h, v = self.horizontal, self.vertical

    for r in range(len(h)):
        s = []
        for c in range(len(v[0])):
            k = (v[r][c],
                 v[r+1][c],
```

```
                h[r][c],
                h[r][c+1])
            s.append(glyphs[k])

        output.write(''.join(s) + '\n')
```

What was the other half? It's not, mostly, that Python uses shorter tokens. As far as I can tell, the only really missing part is the types (and struct fields), although those are only about 62 of the extra 300 bytes of code. In Python, we don't have to declare that our hash key is a `context` or name its fields, declare the type of the receiver, declare the type of the output stream, or declare the output buffer as a byte slice.

A (non-modern) C++ or Java version would be far worse — it would have type annotations on almost every line of the code.

A non-word-for-word translation

The Golang function above was actually from a translation from a Python program which didn't use a hash table to produce the characters but rather a search over parallel arrays:

```
def render(h, v):
    for hline, upline, downline in zip(h, v[:-1], v[1:]):
        row = []
        for r, l, u, d in zip(hline[1:], hline[:-1], upline, downline):
            row.append(random.choice([c for i, c in enumerate(chars)
                                      if  l == left[i]
                                      and r == right[i]
                                      and u == top[i]
                                      and d == bottom[i]]))
        yield ''.join(row)
```

This ended up being almost as long as the Golang version, a somewhat unfair comparison since it leaves out the Golang initialization code for the `glyphs` hash table:

```
        for i, glyph := range []rune(chars) {
                k := context{
                        up:    topbs[i] == '1',
                        down:  bottm[i] == '1',
                        left:  lefts[i] == '1',
                        right: right[i] == '1',
                }
                glyphs[k] = []byte(string(glyph))
        }
```

Optimizations and proofs

The Python version will detect even method name misspellings at runtime; if you said `output.wriet` instead of `output.write`, it won't complain. Similarly, it detects type errors, where you're calling it with no argument or a non-output-stream argument, at runtime. And if one of the things in `glyphs` isn't actually a string, or one of the keys isn't a 4-tuple as it should be, or `h` or `v` isn't a two-dimensional array of strings of the right size, that's also a runtime error. Moreover, if you got the keys in the 4-tuple in the wrong order, it won't be obvious and Python won't complain.

By contrast, most of these are checked at compile-time by the Golang compiler. It constructs a simple formal proof that the `output` argument has a `Write` method: it's an `io.Writer` interface value, and those have `Write` methods, QED. The only one it doesn't check is the array size, a check it does do at runtime. But the Golang compiler catches a fairly large fraction of bugs by using simple theorem proving at compile time, bugs that the Python program would need a test suite to catch. (I didn't write the test suite.)

The Golang program also runs a lot faster. It generates a 301×301 maze (comparable to the Python program's 300×300 maze) in 400 milliseconds, while the Python program takes 4.9 seconds to do the same task with the same algorithm. Both are single-threaded, so, crudely, the Python program is spending 92% of its time doing computations the Golang program doesn't need to do. (The Golang program could in theory be benefiting from concurrent garbage collection, but its user+system time never exceeds the wallclock time by more than about 3%, so it can't be benefiting much.)

A lot of those computations are the ones the compiler did at compile-time in order to find bugs, using the type annotations. Other optimizations, though, were choices of how to represent the program's state in memory.

Declaring the output buffer as a byte slice was an optimization; originally it was a string appended to with `+=`. Indeed, having an explicit output buffer at all was an optimization; previously the code used `fmt.Fprintf(output, "%c", glyphs[k])`. And the `context` type, with its four boolean fields, is presumably both considerably faster to copy around and gentler on the garbage collector than Python's tuple of reference-counted pointers.

Separating optimization, implementation, and proof

It's tempting to think that proofs and type specifications are at a *higher* level of abstraction than a straightforward implementation, while an optimized implementation is working at a *lower* level of abstraction than that straightforward implementation. And you could imagine a way for that to be true, where you have a sort of cascade from high-level specifications ("the program must not crash", "there must be only a single path through the maze") down to lower-level more detailed specifications ("use Kruskal's algorithm") which fill out the high-level specifications into something closer to the metal, down to low-level micro-optimizations ("represent a cell context as a tuple of four booleans, represent those booleans as single bits in a byte, and index the glyphs numerically by context rather than by hashing"). But the above example shows that this is not really the way things work today. Perhaps it could be?

Ideally, you could separate the decisions about how to represent state (use a byte buffer, use a hash, use an array, use a four-byte struct) from the higher-level specification of the algorithm so that you could separately prove that the algorithm, abstractly, solves the problem it's needed to solve, and that the state representation is a valid state representation for that algorithm (that is, its semantics faithfully implements the semantics the algorithm needs). Given a system that could construct such proofs, you could automatically conduct a

heuristic search for state representations that provide the required properties.

If an algorithm is written in a sufficiently abstract way, it can be automatically specialized to different contexts (given, for example, context-specific implementations of the operations it needs), while proofs of its properties can be written in terms of the abstract algorithm, perhaps with reference to properties of the concrete representation it's using. The C++ STL is one example of this approach, but also, for example, you can use Newton's divided-differences polynomial-interpolation algorithm to perform polynomial interpolation in different fields — including, for example, Galois fields, where it provides not numerical approximation but Shamir secret sharing. But this can only be done automatically if the original specification of polynomial interpolation is written in such a way as to not be limited to floating-point.

Arithmetic approximation

Two very common optimizations: representing integers as fixed-length n-bit bitstrings and approximating arithmetic on them with the cyclic and multiplicative groups of $\mathbb{Z}/2^n\mathbb{Z}$ ("integer arithmetic", in the computer sense); and representing members of $\mathbb{Q}$ or $\mathbb{R}$ as floating-point numbers and approximating arithmetic on them with floating-point arithmetic. The first is precisely correct in the cases where we have no overflow or underflow, and where the overflows and underflows cancel out. (In ANSI C, undefined behavior deprives us of the second possibility unless we use `unsigned` values.) The second is approximately correct when the rounding errors are not too large, which can be detected dynamically using interval arithmetic, affine arithmetic, and friends, or statically using numerical analysis (see Reducing the cost of self-verifying arithmetic with array operations (p. 2205) for more on this relationship.) Also, sometimes, we use fixed-length bitstrings with "integer division" to approximate the richer arithmetic of $\mathbb{Q}$ or $\mathbb{R}$, with rounding-error considerations similar to those of floating-point.

These are not minor or even constant-factor optimizations. Consider the recurrence that gives rise to the Mandelbrot set† — in $\mathbb{C}$ this is $z_i = z_{i-1}^2 + z_0$, but for calculation we normally translate it into $\mathbb{R}^2$:

$$x_i = x_{i-1}^2 - y_{i-1}^2 + x_0$$
$$y_i = 2x_{i-1}y_{i-1} + y_0$$

If $(x_0, y_0) \in \mathbb{Q}^2$, then all of the pairs in the recurrence will be in $\mathbb{Q}^2$, so we can straightforwardly compute them precisely on a computer using arbitrary-precision arithmetic. However, if we do this, in most cases, the computation takes time exponential in i, because the numerators and denominators of the x_i and y_i double in length every iteration! The first time I implemented the Mandelbrot set in Squeak Smalltalk, this is what happened to me, because in Squeak, integer division produces arbitrary-precision rationals by default, not floating-point numbers.

So optimizing this computation by approximating the results with floating-point arithmetic makes it possible to calculate new items of the recurrence in constant time, and without the possibility of a memory-exhaustion exception, while the precise computation takes exponential time and also exponential memory.

Being able to express an algorithm like the computation of the Mandelbrot recurrence *separately from* a specification of the concrete realization of its operations in a computer could allow us to prove results about the algorithm in the abstract, then automatically specialize it for a particular concrete domain (such as 32-bit floating-point numbers) and determine which of these proofs still holds for that concrete approximation. For example, a proof that it computes terms in linear time would fail to hold for an arbitrary-precision realization.

As I mentioned above, the Python and Scheme implementations above of the method of secants have the same feature — if applied to arbitrary-precision rational objects, they can consume amounts of space and time that increase exponentially with the number of iterations, even when they erase previous results, precisely because the expression of the algorithm in those languages can be abstract over numeric types. So, although we could imagine a system which would allow us to mechanically verify a constant-space guarantee for that algorithm in the case where we erase intermediate values and use constant-space numeric types, neither Python nor Scheme is that system; and, although we could imagine a system which would allow us to use the intermediate values computed by that algorithm without explicitly exporting them with a `yield`, neither Python nor Scheme is that system either. We enjoy the worst of both worlds.

† I think Gaston Julia, not Mandelbrot, first called attention to the astonishing features of this simple recurrence, but like nearly everyone else alive, I owe my introduction to it to Mandelbrot.

I don't know what such separation would look like for real

All of the above is a sort of laundry list of ways that these different optimizations get mixed into our source code daily, leading to duplication of code, poor performance (because the cost of an optimization is so high), and lack of adaptability to new situations. But it doesn't go very far toward exploring how we could really separate them in practice. Suppose you write the Mandelbrot recurrence the way we write recurrence relations in math books, as in my example above.

Topics

You can stuff a UHMWPE hammock in your wallet

Kragen Javier Sitaker, 2018-05-15 (updated 2018-10-28) (11 minutes)

How much would a minimal UHMWPE hammock (i.e. a net woven from ultra-high-molecular-weight polyethylene) weigh?

A standard hammock setup reaches 4–6 meters from anchor to anchor, with a hang angle 30° below the horizontal, so the bottom is 1–1.5 m below the anchor points when you're sitting at a point in the middle, and the total length of the hammock then is about 4.5–6.7 m.

UHMWPE is springy enough that you don't need any safety factor for dynamic loads. Suppose the hammock needs to be able to support 200 kg, i.e. 2.0 kN, vertically, or 1.0 kN on each support. This requires the tension in the cord to be 2.0 kN diagonally to have a 1.0 kN vertical component, so the support cord needs to resist 2.0 kN. I don't have stats on UHMWPE handy at the moment, but suppose its tensile strength is 500 MPa, which is probably in about the right range — that's the strength of poorly-heat-treated ordinary steel bolts. Then you need 4 mm² of cross-sectional area to support you.

If your hammock were just a cord, that would be all — you need 6.7 m of 4 mm² cord, 2.3 mm in diameter if it's circular, and you can sit on it. And it would weigh 26 grams at 0.96 g/mℓ.

However, the hammock net spreads out over an area, and you need to not have it break when you're sitting on just part of it. It probably needs to be able to spread out to at least 700 mm wide, and any, say, 100 mm of the width needs to not break when you sit on it. That means the net needs to be about 7 times stronger: 14.0 kN, and thus 28 mm², which will form a sort of rope about 6.0 mm in diameter when you twist it up. Also, the spacing of the lines in the net needs to be close enough that any particular 100 mm of its width has about the same strength, so you need at least 4 lines in that width — and that seems challenging to fabricate, because it means you need 28 knots and at times 56 lines across the whole width, so let's stick with 4. Then you need knots along the length at somewhat regular intervals — let's say, every 50 mm.

This sort of implies that each line only has a strength of 14/56 = 250 N, or 25 kg. This may be a little low — you might be able to break it by hand if you have some kind of tool to keep the line from cutting your hand, like a stick or something — but maybe it's okay.

If the net part of the hammock is 2 m long, then it contains 2 m · 28 mm² = 56 mℓ or 54 g of line, which is divided among 56 lines of 0.80 mm diameter and 0.5 mm² cross-sectional area. There are about 28 · 2000 mm / 50 mm = 1120 knots in the net, so tying it by hand is probably a few hours of work. The remaining 4.7 m of support are still only 4 mm² and so are only 19 mℓ = 18 g of line, giving a total of 72 grams.

The circumference of each net hole there is 100 mm, which is hopefully small enough to prevent injury.

This doesn't include anything to connect the support to, such as a tree-hugging loop or a carabiner or a hook.

If you wanted to include an airproof barrier, 100 microns of boPET

under the hammock would probably be adequate, although you could make reasonable arguments for 200μm LDPE or HDPE — they would be quieter and less likely to rip. 2 m · 700 mm · 100 μm, disregarding the narrowing at the ends, works out to 140 mℓ, which is almost twice the volume of the net hammock itself; it's also about than twice the mass, as PET is almost the same density but slightly denser. Sheets and blankets are heavier still. So we can see that the structural support is a minority of the mass of the overall hammock.

Update: UHMWPE fiber normally has, apparently, 2.4 GPa of tensile strength. This means the 4.7 m of 2kN support line need only be 0.83 mm², 1.3 mm in diameter, and the 250 N net lines need only be 0.104 mm² individually (360 μm in diameter), 5.8 mm² collectively — 2.7 mm in diameter. The overall hammock would occupy 2 m · 5.8 mm² + 4.7 m · 0.83 mm² = 15.5 mℓ, 14.9 g.

If it's possible to actually make this work, it would be pretty stunning: a hammock you can sleep in that's roughly one tablespoon in volume when rolled up.

I tried a minimal version of this: six strands of 500μm UHMWPE stretched between two metal wall hooks, a total of about 24 m. I was able to sit on it for a while, but then it broke in two places. The snapped cord showed a bit of curling, suggesting either that the shock of breaking deformed it plastically, or that it had deformed plastically ahead of time. It's possible I might have mistreated the cords while letting them out, running them around the metal hooks — I might have generated too much heat, melting the cord.

I don't have my scale handy, but this amount of cord ought to occupy 4.7 mℓ and thus weigh about 5 g. If it were a single cord, it should be 1.2 mm in diameter (500 μm √6). This is close to the 1.3 mm I calculated for the 2kN support line, but the fact that it broke without me even jumping around on it makes me think that I should probably double or triple it. Also, though, it's possible that my weight distribution might have been unequal across the lines.

Knots are consistently weak points, not just in the usual way where they put extra stress on rope fibers, but also because UHMWPE is so slippery that it tends to slide out of knots. My first attempt to reconstitute my backpack buckles using UHMWPE and steel rings failed when I pulled on it, not because the thread broke, but because the sheet bend came untied.

A bit of analysis suggests that stretching 1 m of 1mm² UHMWPE cord to its 2.4 GPa limit requires 2.4 kN; if this elongates it by 2% = 20 mm, implying a 120 GPa Young's modulus (a bit over half of steel's) and close to the 66–124 (Spectra) and 115 (Dyneema) given in http://www.mse.mtu.edu/~drjohn/my4150/props.html. Assuming linearity, that's 24 J of energy stored in 1 mℓ (≈ 1 g) of fiber. The 5 g I broke should then be able to store about 120 J as spring energy before breaking. Unfortunately, that's my body moving at a bit under 2 m/s.

(This is similar to calculations I did in Spring energy density (p. 1010) for the specific energy of different spring materials.)

This gets worse if, as suggested above, the main body of the hammock has a cross-sectional area much larger than that of the support line, while being made of the same material, because the main body of the hammock then won't stretch much at all when you sit in it; all the stretch will be taken up by the thinner support lines.

I was thinking that sticking it in series with some kind of spring might help to protect it against shocks, but I don't know what kind of spring. Nylon, from the same page, has 1.36 g/cc, 2.5 GPa Young's modulus, and 100 MPa tensile yield strength, but supposedly also has like 90% elongation at break rather than the 4% you'd calculate from those figures. (https://www.azom.com/article.aspx?ArticleID=477 explains: 6,6 nylon does indeed have 4.5% strain at yield, but also 60% elongation at break.) If we take the 60% figure, but figure that the engineering stress is limited to 100 MPa over nearly all of the distance, we get the remarkable result that our 1m × 1mm² fiber can elongate to 1.6 m under a force of 100 N, absorbing 60 J of energy in the process, more than twice what the UHMWPE fiber can absorb. This is an improvement, but not good enough to justify it.

Maybe a hyperelastic material like latex rubber would be a better choice. Latex has roughly 1500% elongation at break. Some loops cut from a bicycle inner tube might be most practical, despite their suboptimal properties. The 2012 "Characterization of Natural Rubber Latex Film Containing Various Enhancers" https://core.ac.uk/download/pdf/82660375.pdf got an ultimate tensile strength of 0.34 MPa and 1400% elongation at break, but unfortunately they did not plot the stress-strain curve. Surely the ultimate tensile strength here is simply incorrect; other sources give tensile strengths in the neighborhood of 10–20 MPa, such as https://vtechworks.lib.vt.edu/bitstream/handle/10919/26306/1JTS_ooETD.pdf.

If we assume that it's linear (probably conservative for hyperelasticity) and that in practice we don't want to exceed 500% elongation, then we can put it under, say, 5 MPa of stress.

If we were to do the same experiment with our hypothetical 1 m × 1 mm² shape, but this time made from rubber, we would stretch it out to 6 m while averaging 2.5 MPa and 2.5 N. This is a rather pathetic 12 J.

Rhett Allain did some simple experiments https://www.wired.com/story/how-much-energy-can-you-store-ino-a-rubber-band/ to derive a specific energy of 1.7 kJ/kg in tension for the rubber bands he had lying around the office. That means that absorbing an impact of 250 J would require 150 g of rubber.

So rubber is good for limiting forces given a fixed acceleration, like when you have something mounted on a vibrating chassis, but it isn't particularly good at absorbing a fixed impact energy. UHMWPE is many times better than rubber at that, and nylon is twice as good as UHMWPE, albeit dissipatively. (And presumably plastics like LDPE (7 MPa, 400%), PET (100 MPa, 300%), polycarbonate (100 MPa, 200%), and polycaprolactone (10 MPa, 300%, though I think I've seen more like 1000%) are better still (see https://www.smithersrapra.com/SmithersRapra/media/Sample-Chapters/Physical-Testing-of-Plastics.pdf and https://www.makeitfrom.com/material-properties/Polycaprolactone-PCL), if you're willing to allow plastic deformation, making the shock absorber consumable.)

Topics

Anytime realtime

Kragen Javier Sitaker, 2016-04-22 (4 minutes)

Every program is a real-time program; late answers are wrong answers, because every system has a user, and users do not live forever, and they have a discount rate.

A traditional way to ensure that your programs don't miss their hard real-time deadline is to calculate a worst-case run time for your algorithms, then ensure that your input doesn't get big enough to make your run time miss the deadline. This requires analyzing the compiled program.

A different way is to use anytime algorithms; these are algorithms that you can keep running until you are about to run out of time, then get an answer. In general, if you don't run them long enough, the answer you get isn't very good, but in many cases it's still better than no answer. And it's a *lot* easier to verify that an anytime algorithm hits its deadline.

There are several different classes of anytime algorithms; two of the common ones are Monte Carlo algorithms and numerical optimization algorithms.

Monte Carlo algorithms, such as particle filters, work by doing a large number of random trials and producing some sort of aggregate answer from them. For example, if you're ray-tracing, you can shoot rays at random into your scene, ideally several per pixel. This is, as far as I can tell, how Blender ensures that its visual feedback on rotation of complex objects will will always be instantaneous. It's also the way people basically always do radiosity rendering, as far as I know.

Numerical optimization algorithms seek to find an answer that minimizes some "error" or "loss" function by manipulating some "design variables" within a "feasible region", and they find better and better answers when you run them longer. Large systems of linear equations these days are solved by successive over-relaxation, which is an example of such an algorithm, but there are a whole family of fairly generally applicable optimization algorithms:

• Random sampling: generate random sets of design variables, remembering the best one.
• Hill-climbing: incrementally mutate your best set of design variables at random, undoing each mutation that worsens the loss function. Random restarts make this relatively robust against local minima. If you decrease the magnitude of the mutations over time this is "simulated annealing". It can be slow in a many-dimensional design space.
• Genetic algorithms: just like hill-climbing, but you maintain a bunch of different sets of design variables instead of just one at a time, and you cross them with each other, and you make more mutated versions of the ones that are doing best.
• Gradient descent: just like hill-climbing, except that you calculate the gradient of the loss function with respect to your design variables, so that you can incrementally mutate all of your design variables in the direction that decreases the loss function the most, instead of at random like in hill-climbing. If your design space is

many-dimensional, this is a lot faster than hill-climbing, but it requires you to be able to calculate the derivatives. This is why automatic differentiation is so hot in the last few years. It also benefits from random restarts, a lot of the time.

• Newton-Raphson iteration: vaguely related to gradient descent in that your next guess is where you would linearly extrapolate that the loss function hits zero according to the gradient. This is *much* faster than gradient descent for some things.

Obviously all of these can benefit from massive databases of existing designs and clever neural network stuff to speed them up.

Topics

Rich programmers

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

a comment on http://coliveira.net/2008/12/why-great-programmers-dont-make-aso-much-money-as-soccer-stars/

You write:

"Even if the best programmer in the world is in a team with other 10 average developers, he won't be able to cope...

"That is the (sad) tale of software development in more than 90% of programming shops. It is not a mystery, then, that being a software manager, especially a good one, pays much more on average than being a good developer."

Let me suggest another hypothesis. The second richest man in the world, much richer than any soccer player, is a programmer. I'm acquainted with a half-dozen or so programmers who have made hundreds of millions of dollars by programming — some have made more than a billion. I've met one guy who worked on his own for about 20 years, on a team by himself, and retired with substantial savings in one of the most expensive cities in the world, Menlo Park. (Peter Deutsch; none of this is a secret. I have no idea how much money he actually has.)

You meet these guys *all the time* in Silicon Valley, Seattle (where they work for Microsoft), and New York (where they work for investment banks, or did until a few weeks ago). Generally they aren't into flashy cars, big mansions, wild cocaine-filled parties, and trophy wives, so it usually isn't obvious that they have a lot of money.

So I suggest that perhaps your premise is wrong. Programmers *do* make as much money as soccer stars and rock stars. You just haven't noticed because you live in Brazil, which is a wonderful country with fantastic soccer teams, intelligent and friendly people, and enormous intellectual resources that unaccountably hasn't had very many superstar programmers yet. Maybe they moved to Silicon Valley to work with the other superstar programmers from around the world, and in the 1970s and 1980s it was really hard to get good hardware in Brazil, so you kind of lost a generation of programmers. (Still, how much do you think Red Hat is paying Marcelo Tosatti and Alexandre Oliva? How much do you think they have turned down from IBM and Google?)

It's true that nobody is going to make a billion dollars programming on a team full of bad programmers, but nobody is going to make a million dollars a year playing soccer in the Premier Development League either. In both cases the most talented are going to move on to someplace better, not just so they can get paid more (although they do, because the companies they go to work for make a lot more money per employee), but so they will have a chance to work with better collaborators.

Which means that in the 90% of programming shops you talk about above, you will never meet them.

But they're making a lot more than your manager is. One acquaintance of mine turned down job offers to go work for Google

(as a programmer) because their offer of US$120 000 a year was a 40% pay cut from what he was getting paid at an investment bank. He made several million dollars the next year, selling a site he had built in his spare time, over the previous five years or so, to another company.

Topics

- Politics (p. 3639) (39 notes)
- Economics (p. 3424) (33 notes)

Fixed point

Kragen Javier Sitaker, 2014-04-24 (1 minute)

Fixed-point math is a case where you'd really benefit a lot from a rich type system. Instead of "This is an integer," or "This is a floating-point number," you'd like to be able to say, "This is a 32-bit fixed-point number with 2 decimal digits of fraction," or perhaps more commonly, "This is a 16-bit fixed-point number with 8 bits of fraction". This allows you to do something like floating-point math, but with all the manipulation of the exponent done at compile time, leaving only mantissas to run-time. Especially for addition, subtraction, and comparisons, this is a huge savings on processors without floating-point hardware, such a huge savings that it's often worth it to do it by hand, without compiler support.

Type systems like C's can't cope with types like that. C++'s template system can do surprising things sometimes, but I don't think it can do it either. But you could certainly imagine a language whose type system was rich enough to let you do it.

Topics

Text editor slow keys

Kragen Javier Sitaker, 2017-02-07 (2 minutes)

It is often noted that users continue to use inefficient ways of doing things (for example, repeating cursor-key movements) even when better ways are available (for example, mouse clicks or incremental search (LEAP, as Raskin called it)). A simple way to change users' behavior might be to add a certain amount of extra frustration to the inefficient way of doing things.

For example, perhaps the first cursor-key movement takes 10ms, and each successive one takes 20% longer, so the first few delays would be [10, 12, 14, 17, 21, 25, 30, 36, 43, 52, 62, 74, 89, 107, 128, 154, 185, 222, 266, 319, 383, 460, 552, 662, 795, 954, 1145, 1374, 1648, 1978] milliseconds. For moving three or four characters, the cursor movement would be instant; even at 14 characters it would be barely perceptible; at 19 it would start to feel slow; and by 28 it would be really annoying. Even without any further features, this would start to be uncomfortable enough to encourage users to practice other methods of movement, if they knew them.

This is just one example of a larger shift in user interface design that I think is overdue. The job of the user interface is not just to provide access to functionality and reduce the user's cognitive load; it should also progressively enhance the user to higher and higher levels of awesomeness by shaping their behavior. It's unavoidable that the results produced by the user interacting with the UI will shape the user's behavior through classical and operant conditioning, but so far user interface and user experience designers have not risen to meet the responsibility that comes with that fact; if anything, the result has been the opposite, where user experience designers at companies like Zynga and Facebook work to shape users' behavior to click more ads and use their apps more.

A good user interface, like a game, is its own tutorial.

Topics

Buenos Aires seen from behind the mirror

Kragen Javier Sitaker, 2014-09-02 (7 minutes)

The cracks crazing the boiling blue sky run together into rivulets, brooks, mighty raging rivers flowing into a sunlit yellow tree trunk standing brilliant against the cold, tranquil winter sky, riddled with the bullet holes of its shadowed fruits. The numinous rising wingbeats of a doomed pigeon, one foot long ago eaten by lye or some leprosy, carry it across the frozen static of crossed branches with another tree that still retains its worthless, dried leaves, the poisoned excrement of a summer it cannot remember. Below, there is singing: the radiator fan of a car lamenting its overwork, the baritone thunder of a bus diesel. The fragmentary sidewalk, bearing the scars of too many careless attempts to repair the electrical and communication lines beneath bears the weight of dozen fear-filled West African Plains Apes, each containing a universe of wonders no one else will ever glimpse, each a lonely prisoner of her own distrust and hate, their lives piously dedicated to serving hungry ghosts that never lived upon this earth, but who nevertheless poured these sidewalks, planted these trees, and built these engines that moan with the cries of fish and krill dead since before our foremothers began to lay eggs on land, ghosts called states and companies and races and churches and schools.

In the bus, a skeletal man in a suit fidgets, alternately slumping and suddenly sitting erect, mumbling to himself as he wipes his eyes and scratches his chin. A young mother braids the hair of a daughter on her lap. When her daughter was born in a heroic ordeal of agony and blood, she came to the city, seeking solace in its vastness, yearning to become part of something bigger than she is, leaving behind the memories of her lover who died bisected on the train tracks, stinking of Fernet Branca, a memory that hurt her more than the birth of his child. Still sometimes she awakes at night to his kisses, only to find herself more alone than ever, alone and aging toward her inevitable death.

Her great-grandfather died by a bullet from a man from this city, who came to fight for his rstate and his ace, the race of the city, and erase hers from the Earth; to this race existing only in his imagination, he dedicated his demonic human sacrifice, lawlessly murdering in the name of Law. Her line did not die, but now she sings of her love to her daughter with the words of the man who slaughtered her tribe, and in this city her accent and her skin mark her forever as an outsider, a subjected people. Some days she dreams of a world where her people remain free and her love never tasted Fernet, never called her a slut, where he stood beside her still, tall and strong as he was when she first kissed him, not bloody and cold like when she identified him in the morgue. She dreams that her great-grandfather never beat his wife.

The bus is the larger body of its driver, and in his halting and hurried caresses we feel his anger, his impatience, his desperation to compete for position with the taxis and motorcycles he swims through. The girl in her mother's lap feels it and scowls; the skeletal

man in the suit fidgets a little faster and rubs his face, a face nobody else caresses. The engine growls the impotent rage of its driver, oblivious to the beatific smile on the plastic face of Jesus beside the steering wheel.

The air cylinder to open the doors cries out like a gunshot in a tunnel, and the bus leaves me behind, its superchargers screaming in fury. I walk through the darkening silence it left behind, its vitriol stinging my nostrils like the legacy of decades of insufficiently strict pollution standards, as if a law were as real as the bloated corpses of idealists who hung themselves or took fifty times the lethal dose of Ecstasy, trapped in their own unyielding prisons of logic and delusion and burgeoning, cancerous timidity.

Later, the planet has turned and shadowed me, wrapping me in a darkness that reaches only to behind the nearest streetlight, an assembly of 98 5-watt white LEDs. These false stars of civilization, infecting the wound of our fear of our fellow humans, cannot hold back the unstoppable blackness of the night, but they do shield us from its wonders with ignorance. Children growing up here like weeds in the cracks of the concrete never know the sight of the Milky Way or the Pleiades, except on the hood of a Subaru. At night the woman's grandfather in her village had used stories rather than streetlights to swaddle her in addictive, comforting, toxic ignorance, telling her the Milky Way was spilled milk in the sky.

I emerge from an artificial mountain where I have been reflecting and weeping, a cube of false stone honeycombed with geometrically formed corridors, with trees and meticulously mowed grass on top, and in a few pits in the middle. It has stood in the middle of this city for four generations, and thousands of people earned their livings spinning and weaving in it; unknown dozens of them have lost fingers or hands or died, mangled by the machinery, within its echoing cavernous depths. Walking around it takes almost ten minutes. Our concrete is lashed together with tendons of steel, but our pozzolanic additives are inferior to the Romans', so such buildings decay without constant attention, and a generation ago the roof swimming pool had cracked and would not hold water, and the palm tree next to it had fallen over, unable to shatter our conglomerated stones with its roots. Now I make my office within.

On a black roundrect of mostly stone, bearing an ideogram of a fruit on the back to represent the imaginary person who made it and still owns it in a way that I never will, I inscribe my thoughts in letters of fire, letters that will leap invisibly through subtle vibrations behind the air when I arrive

Topics

• Psychology (p. 3669) (18 notes)
• Fiction (p. 3454) (7 notes)

Querying a pile of free-text strings with quasi-Prolog

Kragen Javier Sitaker, 2017-11-17 (6 minutes)

Literal facts

So, I've been thinking about a sort of free-text Datalog or Prolog system that uses string matching. At the most basic level, you can put literal facts into it (what Datalog would call ground facts) in free text:

```
Romina Vasconcelos lives at Colón 2014 #8.
Valeria Lemmi lives at Suipacha 825.
Valeria Lemmi dances contact improv.
Glenda Quesada dances contact improv.
Glenda Quesada's phone number is +54 11 4777 4909.
Glenda Quesada lives at Hidalgo 979.
Lunch with Glenda Quesada on 2017-11-22 at O'Higgins 1255.
Lunch with Romina Vasconcelos on 2017-11-23 at Billinghurst 3236.
Valeria Lemmi's phone number is +54 11 2560 6898.
Glenda Quesada and Valeria Lemmi are friends.
Valeria Lemmi and Juan Pablo Suracco are friends.
```

Simple searches and joins

And you can search it, at the simplest level, with just a pattern like '%Valeria Lemmi%' or '% dances contact improv.' That is already pretty useful, though not very high tech; it's basically `grep`.

The next level, though, is to relate two or more such facts with patterns containing a common variable. For example:

```
Lunch with @who on @when at @where.
@who's phone number is @phone.
```

This finds a single viable assignment of variables out of the above facts, still based entirely on substring matching:

```
who             when        where           phone
--------------  ----------  --------------  ----------------
Glenda Quesada  2017-11-22  O'Higgins 1255  +54 11 4777 4909
```

Deduction or inference rules

At a third level, we can add deduction rules. For example, with the premises above the line and the conclusions below:

```
@a and @b are friends.
----------------------
@a and @b are connected.
```

If we add a separate rule which can justify the same conclusion, then we can get transitive closure:

```
@a and @b are friends.
@b and @c are connected.
-----------------------
@a and @c are connected.
```

From this and the above facts we can justify, for example,

```
Glenda Quesada and Juan Pablo Suracco are connected.
```

This much is sufficient to implement basic SQL pointwise queries, but it doesn't yet touch issues of quantification, aggregation, negation, and ordering. You can get existential quantification simply enough:

```
@someone's phone number is @number.
-----------------------------------
@someone has a phone.
```

Note that this immediately gives you both AND and OR (intersection ∩ and union ∪), though not negation or even abjunction. You can get AND with two clauses:

```
@someone dances contact improv.
@someone lives at Hidalgo @address.
-----------------------------------
@someone is acrobatic.
```

And you can get OR with two inference rules that can produce the same result:

```
@someone and Juan Pablo Suracco are friends.
--------------------------------------------
@someone is cool.

@someone dances contact improv.
------------------------------
@someone is cool.
```

Problems of infinite regress

In some sense, this string-based formulation is strictly more powerful than Datalog, in dangerous ways. For example, given the rule:

```
@P.
---
It is the case that @P.
```

we can derive:

```
It is the case that Valeria Lemmi lives at Suipacha 825.
It is the case that It is the case that Valeria Lemmi lives at Suipacha 825.

It is the case that It is the case that It is the case that Valeria Lemmi lives a
ot Suipacha 825.
```

and so on.

User interface affordances

An obvious problem with this kind of naïve text matching is that it's very easy to miss records because of very slight textual mismatches. For example, consider the following pattern:

```
@who's phone number is @phone.
```

It won't match the following fact:

```
Valeria Lemmi's phone number is +54 11 2560 6898.
```

That's because it uses a different apostrophe, an ASCII one rather than the Unicode one used in the pattern. It won't match this one either:

```
Valeria Lemmi's cellphone number is +54 11 2560 6898.
```

Once you become aware of these, you can bridge them with inference rules:

```
@who's cellphone number is @phone.
---------------------------------
@who's phone number is @phone.
```

But in many cases it's better to avoid them in the first place, which is best done with user interface affordances.

If you have a set of inference rules like the examples in the earlier section, or even queries that you are likely to evaluate again, they can provide a certain amount of guidance to the system about what kind of data you might want to put into it. A query result, without any further inference rules, is a table that you can type more values into, which immediately provides a quick data-entry user interface. But also, they provide some amount of hints as to the type structure of your data.

```
Lunch with @who on @when at @where.
@who's phone number is @phone.
```

Then, what can we tentatively infer, given the following ground fact?

```
Valeria Lemmi's phone number is +54 11 2560 6898.
```

We can infer that "Valeria Lemmi" is some kind of meaningful entity, and it might be worthwhile to make her name a link to a view of all facts that mention her; and we can also infer that a possible thing that the user might want to do in that context would be to add a ground fact of the form "Lunch with Valeria Lemmi on ... at ...." to the store. There might even be places where you've planned lunch before that could be suggested for the location "field".

Topics

Learning low level stuff is not just fun, but also useful

Kragen Javier Sitaker, 2007 to 2009 (5 minutes)

This post argues that learning low-level stuff is a waste of time: http://www.fluidinfo.com/terry/2008/06/18/embracing-encapsulation/

But I don't think that the low-level stuff is frozen in time; it changes too. There was a time when a better C compiler would only help the few people working on Unix at Bell Labs, but today it can improve performance (or security, or debuggability, or whatever) for all of that high-level stuff built on top of C. So the people working on high-level stuff kind of act as a multiplier for the people working on low-level stuff. Which is why Intel and AMD and ARM etc. can spend as much effort as they do designing CPUs.

More and More People Work at the Low Level

I wonder how the amount of effort that goes into making CPUs and other electronics for computers has changed over the years. Intel alone has something like 100 000 employees, and essentially all of them is focused on the lowest levels of computers: processors, motherboard chipsets, flash memory, and so on. AMD has another 16 500 employees, and other top semiconductor companies (Toshiba, STMicroelectonics, TI, Samsung Electronics, Freescale (24000), Infineon, Sanmina-SCI, and so on) add several hundred thousand more people.

Back in 2000, the EE Times reported that the "embedded software tools market" was US$670 million (estimated at US$500 per seat) and the electronic design automation market was four times that, or US$2.7 billion (estimated at US$100 000 per seat). More recently CIOL reported almost twice that size, and EE Times Asia reported 192 000 total semiconductor design seats in 2004, although the article was published on April Fool's Day, 2005, plus 537 000 FPGA and PCB design seats:

> A "seat count" report issued by Gartner Dataquest in January showed that there were only about 56,000 ASIC design seats in 2004, a number that's expected to decline to 39,000 by 2008. The total number of semiconductor design seats was pegged at 192,000 for 2004, with a CAGR of just 1.3 percent until 2008.
>
> In contrast, the report cited 537,000 FPGA and PCB design seats in 2004, growing at a 2.9 percent CAGR. Granted FPGA and PCB designers don't spend as much money as IC designers, but there are a lot more of them. And there are tough problems that could prove very lucrative for vendors that have solutions. For example, FPGA designers need to look at power and signal integrity, and PCB designers are having a hard time getting IC packages to work on boards.

So it seems that the number of people working at the very lowest levels --- the hardware levels --- has been gradually increasing over the last several years, and has already reached a really remarkable number.

Changing the Low Level is Getting Easier

To write software for CP/M, you would write it in 8080 assembly, or maybe Z80 or 8085 assembly if you were willing to limit your userbase, and used the BIOS and BDOS functions for I/O and file management. You could use C or FORTH, but they had big drawbacks. Porting software in 8080 assembly even to MS-DOS on an 8086 was a pain, and porting it to a 68000 basically required rewriting it, or buying or writing a slow 8080 emulator on the 68000.

If you write software in C, it's only a bit more effort to make it portable and compile it for many different CPU architectures. Alll of the tens of thousands of packages in Debian Linux are routinely built for eleven different CPUs: the Alpha, the Opteron, the ARM (big and little-endian), the HP PA-RISC, the Intel 386, the Itanic, the 68000, the MIPS (big and little-endian), the PowerPC, the IBM System/390, and the SPARC. All of the 6400 packages in NetBSD run on all of its supported 16 CPU architectures. Some individual software packages written in C have been ported to even more; GCC, for example, runs on something like 30 CPU architectures.

This kind of portability has the advantage that you can switch CPU architectures and keep using the same software --- and the hardware exists only to support the software, anyway, just as the software generally exists only to support human activities carried out with it.

This still leaves the problem of

Topics

• Programming (p. 3658) (286 notes)
• History (p. 3500) (71 notes)

Jellybean ICs 2016

Kragen Javier Sitaker, 2016-07-14 (updated 2019-05-05) (17 minutes)

I thought I'd check out some of the cheapest and most popular items in the most diverse IC categories on Digi-Key.

(Related to My attempt to learn about jellybean electronic components (p. 1974), from 2017 and 2018, which isn't specifically focused on chips, and Transistors vs. Microcontrollers (p. 2918), which explores the implications of microcontrollers now being as cheap as individual discrete transistors nowadays in many cases.)

Microcontrollers: 61814 items

• 72000 available of the US$2.86 http://www.digikey.com/product-detail/en/texas-instruments/MSP430G2755IDA38R/MSP430G2755IDA38R-ND/4090934 which is a TI MSP430, "16BIT 32KB FLASH 38TSSOP". 16MHz, 32K flash, 4K RAM, probably the usual super low power, a 12-channel 10-bit ADC, 32 GPIOs, internal oscillator, and so on. Other MSP430 parts with 55601 available are #3.

• 67594 available of the US$2.69 http://www.digikey.com/product-detail/en/freescale-semiconductor-nxp/S9S08SG16E1CTLR/S9S08SG16E1CTLRDKR-ND/2252301 Freescale S9S08SG16E1CTLR, "8BIT 16KB FLASH 28TSSOP," from their S08 series, which I've never heard of before. It's the MC9S08SG32 in the datasheet, in the HCS08 category, originally the 68HC08 or HC08, the successor to the 68HC05, a successor to the 6800. Also it's sometimes called the 9S08. So despite all the obfuscation this is a 6800. But a 40MHz 6800 with memory paging and on-chip flash, 16K of it I guess. Von Neumann, 22 GPIOs (with two selectable output drive strengths plus an optional pullup on input), 8 interrupt pins with selectable priority, a 16-channel 10-bit ADC, SPI, I²C, wakeup timer, PWM generation, 1024 bytes of RAM. Its big brother, the ...SG32, has 32K of flash and operates up to 150°. Runs on 5V, normally uses an external crystal, though it has a 40MHz internal clock source too. It has a "standard 6-pin header" for a documented "background debug mode" that's driven through a single-wire serial protocol, typically bitbanged from a PC. The BDM documentation says there are other secret test modes too. It makes it sound like chips of this family found in the field might have the Flash-reprogramming functionality available through the debug port disabled.

• 45696 available of the US$3.20 http://www.digikey.com/product-detail/en/atmel/ATMEGA48-20AU/ATMEGA48-20AU-ND/739775 Atmel ATMega48, "8BIT 4KB FLASH 32TQFP". 20MHz, 8-bit, 23 GPIO, 4K of flash and half a K of RAM, 8 10-bit ADCs.

• 9000 available of the US$0.42 http://www.digikey.com/product-detail/en/microchip-technology/PIC10F200T-I-OT/PIC10F200T-I-OTTR-ND/665882 PIC10F200T-I/OT, a 4MHz Microchip PIC with 384 bytes (!!) of flash, 16 bytes (!!!) of RAM, and 3 GPIOs, in a six-pin SOT-23

package, 3.1mm × 1.75 mm × 1.3 mm.

• 4500 available of the US$1.01 http://www.digikey.com/product-detail/en/atmel/ATTINY4-TSHR/ATTINY4-TSHRTR-ND/2238292 ATTINY4-TSHR, a 12MHz Atmel AVR ATtiny4 with 512 bytes of flash and 32 bytes of RAM, in the same package as the PIC.

• 127 available of the US$1.11 http://www.digikey.com/product-detail/en/silicon-labs/EFM8SB20F16G-A-QFN24/336-3175-5-ND/5115732 Silicon Labs EFM8SB20F16G-A-QFN24, a 25MHz 8051, "8BIT 16KB FLASH 24QFN", with 4¼ kibibytes of RAM and 16 kibibytes of flash. This is the cheapest microcontroller with over 4K of RAM. It runs off 1.8 to 3.6 volts, with 16 GPIOs, a 15-channel 10-bit 300ksps ADC, a thermometer good to about half a degree, and a PWM generator; of the "Sleepy Bee" line, "the world's most energy friendly 8-bit microcontrollers". 50 nA sleep current (or 300 nA with the internal RTC running), 170 µA / MHz; most instructions are 1- or 2-cycle, so if we assume an average of 1½ and 1.8V, that's about 200 pJ/insn. The RAM is "256 bytes standard 8051 RAM and 4096 bytes on-chip XRAM"; the GPIOs are 5 mA source, 12.5 mA sink. And it has an internal 20MHz ±10% clock, an internal 16 (or 32?) kHz clock, and a 24.5MHz ±2% one, so you don't need an external crystal, and the RTC optionally uses a watch crystal. Four timers. It ships with a UART bootloader. Its "on-chip debugging interface" is a thing called "C2 Silicon Labs 2-Wire", which is apparently public. 4mm × 4mm.

• 2,570 available of the US$3.49 http://www.digikey.com/product-detail/en/atmel/AT80C51RD2-3CSUM/AT80C51RD2-3CSUM-ND/1026863 Atmel AT80C51RD2-3CSUM, which is a 5V 40MHz 8051 in a 40-pin DIP with no ROM; you hold its "external enable" pin low so that it executes code from an external ROM.

• only 22 available of the US$2.06 http://www.digikey.com/product-detail/en/intel/EE80C51FA24SF88/864420-ND/1464725 Intel EE80C51FA24SF88. That's an actual 8051! With no ROM, using an external ROM.

Linear regulators: 58468 items

• 163000 available of the US$0.71 http://www.digikey.com/product-detail/en/texas-instruments/LM317LIPK/296-21742-6-ND/1944499 LM317LIPK, an adjustable ungrounded linear voltage regulator that can output from 1.2 to 32 V at up to 100 mA, in a tiny SOT-89-3 case. This LM317L family is the low-current version of the regular 1500mA LM317. You adjust it with an external voltage divider providing negative feedback from its output; I guess you could probably use a FET in its linear region to make it a more custom voltage regulator. And you can also use it as a precision current source! Note that this part is more popular than any of the microcontrollers. Its datasheet includes instructions for how to make a slow-turn-on regulator circuit with it, I guess to ensure that your brownout detection circuits have time to do a power-on reset.

• 121799 available of the US$0.39 http://www.digikey.com/product-detail/en/toshiba-semiconductor-

°and-storage/TCR2EF12,LM%28CT/TCR2EF12LM%28CTDKR-N°
°D/4503462 Toshiba TCR2EF12,LM(CT), “LDO 1.2V 0.2A SMV”. This is a 1.2-volt output voltage regulator with an 0.38V dropout and up to 200mA of output. I am guessing that this is such an important voltage because of NiMH battery chargers, but I don't know.

• 14254 available of the US$0.12 http://www.digikey.com/product-detail/en/microchip-technology/°
°MIC5365-3.0YC5-TR/576-3191-1-ND/1868228 Microchip (ex-Micrel) MIC5365-3.0YC5-TR, a tiny (1mm×1mm!) 3V 150mA linear regulator. I think this may be the cheapest linear regulator of all.

Memory: 40599 items

• 168177 available of the US$0.23 http://www.digikey.com/product-detail/en/stmicroelectronics/M20°
°4C32-WMN6TP/497-5027-6-ND/1007946 STM M24C32-WMN6TP, a 400kHz 32-kibibit serial I²C EEPROM in an 8-pin package. STMicroelectronics will even sell you an unsawn wafer! It supports random address reads, and also erases of individual 32-byte pages.

• 130237 available of the US$0.36 http://www.digikey.com/product-detail/en/winbond-electronics/°
°W25X40CLSNIG/W25X40CLSNIG-ND/3008652 Winbond W25X40CLSNIG, a 104MHz (!) 4-mebibit (!!) dual SPI EEPROM in an 8-pin package. It's insane that this is literally 256 times faster than and 128 times bigger than, but only 57% more expensive than, the #1 memory above.

• 5357 available of the US$0.13 http://www.digikey.com/product-detail/en/stmicroelectronics/M20°
°4C02-FDW6TP/497-15746-1-ND/5283304 STMicroelectronics M24C02-FDW6TP, another 400kHz serial I²C EEPROM, this time 2 kibibits. This one claims “More than 200-year data retention,” which is a surprising claim I haven't seen other Flash chips make, but now that I notice, the 32-kibibit one above makes it too!

• 27500 available of the US$2.78 http://www.digikey.com/product-detail/en/stmicroelectronics/M95°
°M02-DRMN6TP/497-11405-2-ND/2679404 STMicroelectronics M95M02-DRMN6TP, another serial EEPROM with “200-year data retention”, but this time SPI, two megabits, and 5 MHz! It's in the same 8-SOIC as the others above.

Op-amps, etc.: 36285 items

• 136806 available of the US$0.57 http://www.digikey.com/product-detail/en/stmicroelectronics/TS°
°V321RILT/497-8164-6-ND/1884598 STMicroelectronics TSV321RILT “Op Amp GP 1.4MHz RRO SOT23-5”, “General purpose input/output rail-to-rail low-power operational amplifiers”, going 200mV past the power rails, good up to 125°, now replaced by the LMV321L. These are lower-voltage versions of the LMV324 for the 2.5–6V range, have a 1.3MHz GBP, and can source or sink up to 80mA and is stable driving up to 500pF of capacitive load.

• 99104 available of the US$1.00

http://www.digikey.com/product-detail/en/texas-instruments/LMo
oC6032IMX-NOPB/LMC6032IMX-NOPBCT-ND/3440137 TI LMC6032IMX/NOPB, which is also a 1.4MHz GBP op-amp, but there are two op-amps on the chip, so it has 8 pins. This one only sources or sinks up to 18mA, it's not quite rail-to-rail, runs on 4.75–15.5V, and is stable driving up to 100pF of capacitive load.

• 83402 available of the US$0.77 http://www.digikey.com/product-detail/en/rohm-semiconductor/o
oBU7411SG-TR/BU7411SGDKR-ND/2791706 Rohm BU7411SG-TR, an "Op Amp GP 4KHZ 5SSOP", which makes it sound like it's worse than the TSV321$ in every way, orders of magnitude worse in the GBP. But it only need 0.35µA of supply current, and needs even lower voltages: 1.6–5.5V. It can only source or sink a couple milliamps on its output.

• 53751 available of the US$0.24 http://www.digikey.com/product-detail/en/microchip-technology/o
oMCP6001T-I-OT/MCP6001T-I-OTCT-ND/697158 Microchip MCP6001T-I/OT, the world's cheapest op amp. This is a rail-to-rail 1MHz GBP op-amp using 100µA and sourcing/sinking up to 23mA, running on 1.8 to 6 V, in a SOT-23-5. It says it's "low power" but it consumes 300 times what the BU7411SG above does.

• 2148 available of the US$0.31 http://www.digikey.com/product-detail/en/microchip-technology/o
oMIC912YM5-TR/576-3691-1-ND/2339688 Microchip MIC912YM5-TR "Op Amp VFB 200MHz SOT23-5", which is the cheapest 10MHz-or-over op amp; this was a Micrel part before Microchip bought them. This uses 2.4mA of supply current, is stable with unlimited capacitive loads, and runs on 2.5–9V. The datasheet PDF is corrupted, so I can't find out more.

• 51585 available of the US$2.77 http://www.digikey.com/product-detail/en/analog-devices-inc/AD o
o8606ACBZ-REEL7/AD8606ACBZ-REEL7DKR-ND/2468795 Analog Devices. AD8606ACBZ-REEL7 "Op Amp GP 10MHZ RRO 8WLCSP". It's a dual-op-amp chip, and the most popular (as measured by Digi-Key stock) RF op amp. It's immune to oscillation and phase reversal even when driving large (1000 pF) capacitive loads, and its picoamp input offset current allows you to use large feedback resistors.

Power management supervisor ICs: 35843 items

• 67032 available of the US$0.54 http://www.digikey.com/product-detail/en/on-semiconductor/NC o
oP302LSN20T1G/NCP302LSN20T1GOSCT-ND/2121405 ON Semiconductor NCP302LSN20T1G two-volt detector in a 5-TSOP (a thin SOT-23 with five pins). This is a power-on reset circuit. These suck 500 nA from 0.8 V to 10 V and produce an active-low reset signal when the voltage is below 2 V with hysteresis and a delay programmable with a capacitor on one of its pins. "This device contains 28 active transistors."

• 56,660 available of the US$0.36 http://www.digikey.com/product-detail/en/diodes-incorporated/A o

oPX809-31SAG-7/APX809-31SAGDICT-ND/1844722 Diodes Incorporated APX809-31SAG-7, which is basically the same thing, but with a 3.08V threshold voltage, a fixed delay of 240ms instead of a programmable delay, and only 3 pins. It sucks 30000 nA though and only works in the 1.1 to 5.5 volt range.

• 38041 available of the US$0.64 http://www.digikey.com/product-detail/en/torex-semiconductor-lto
od/XC61CC2502MR-G/893-1027-6-ND/2138365 Torex XC61CC2502MR-G, which is a ±2% low voltage detector in a SOT-23 with a 2.5V threshold that sucks 700 nA.

DC-DC switching regulators: 25051 items

• 79,855 available of the US$1.16 http://www.digikey.com/product-detail/en/texas-instruments/TPSo
o61221DCKR/296-41854-6-ND/5224781 TI TPS61221DCKR, a TPS61221DCKR, described as "Boost Switching Regulator IC Positive Fixed 3.3V 1 Output 200mA (Switch) 6-TSSOP, SC-88, SOT-363". This takes an input of anywhere from 0.7 to 5.5 volts and turns it into a 3.3-volt ±3% output at up to 200mA. It needs an external inductor (4.7µH suggested, higher inductances give higher efficiency) and a couple of 10µF capacitors; the adjustable-voltage versions also use a couple of resistors for feedback. At over 1V input and from 0.1 to 100 mA output, it's over 80% efficient; above 2.3V input and 0.3 mA output, it's over 90% efficient. This is basically intended for running 3.3V circuits off rechargeable batteries.

• 73,400 available of the US$0.70 http://www.digikey.com/product-detail/en/alpha-omega-semicondo
ouctor-inc/AOZ1280CI/785-1277-1-ND/2769845 Alpha Omega AOZ1280CI, described as "Buck Switching Regulator IC Positive Adjustable 0.8V 1 Output 1.2A SOT-23-6". This is a step-down switching regulator that takes 3–26V in and produces a 1.5MHz PWM output signal which you filter with a 2.2µH output inductor and a 10µF output bypass cap to get whatever voltage you want (programmed with a 800mV voltage divider), down to 0.8V, at up to 1.2 amps. It's "up to 95% efficient", but typically more like 80–90%. It sucks a whole milliamp itself, though, so it's not suitable for super low-power circuits.

• 58,254 available of the US$1.36 http://www.digikey.com/product-detail/en/vishay-siliconix/SIP121o
o07DMP-T1-GE3/SIP12107DMP-T1-GE3CT-ND/3309123 Vishay Siliconix SIP12107DMP-T1-GE3 "Buck Switching Regulator IC Positive Adjustable 0.6V 1 Output 3A 16-SMD", which honestly sounds like more of the same, but higher power and less versatile. But it says it's "current-mode constant on-time", and its switching frequency is 4MHz, and it has 16 pins.

Programmable timers and oscillators: 23241 items

• 56,647 available of the US$0.44 http://www.digikey.com/product-detail/en/texas-instruments/NE5o
o55P/296-1411-5-ND/277057 TI 555 (NE555P), the chip that gave

birth to this product category in the 1970s. The datasheet says, "September 1973—revised September 2014", which I guess means TI bought Signetics at some point. Astable or monostable operation. Sinks or sources up to 200mA, runs on 4.5 to 16 volts, at 1 millihertz to 100 kHz. This is an 8-pin DIP.

• 34,355 available of the US$0.73 http://www.digikey.com/product-detail/en/intersil/ICM7555IBAZ-T/ICM7555IBAZ-TDKR-ND/2529181 Intersil ICM7555IBAZ-T, which is a CMOS version of the 555, but runs at up to 1 MHz, with a wider 2–18 V power supply, and it runs on 60 μA and comes in an 8-SOIC rather than a DIP.

• 2,717 available of the US$3.41 http://www.digikey.com/product-detail/en/texas-instruments/SN74LS628DR/296-37424-1-ND/4758875 TI SN74LS628DR, aka 74628, a 20MHz VCO, an improved version of the 74LS324, using an external timing resistor to improve temperature compensation. Datasheet is from 1980, revised 1988. It can oscillate at roughly 2MHz to 20MHz as the frequency-control input voltage ranges from 1V to 5V.

• the great majority of these 23000 items are very slightly different VCXOs whose prices start at US$30.

FPGAs: 19724 items

• 3,482 available of the US$32.49 http://www.digikey.com/product-detail/en/atmel/AT40K20AL-1BQU/AT40K20AL-1BQU-ND/1914271 Atmel AT40K20AL-1BQU, a 1024-cell 3.3V FPGA ("30000 gates") with 8192 bits of 10ns RAM and 114 5V-tolerant GPIOs in an LQFP. Supposedly good up to 100MHz, with 50MHz multipliers. Its cells have propagation delays of about 2 ns, are capable of implementing full-adders or even multiplier bits, and have diagonal interconnections to make multipliers out of. Bitstream format and debugging features are apparently not documented.

• 3,127 available of the US$6.75 http://www.digikey.com/product-detail/en/microsemi-corporation/A3P030-QNG48/1100-1012-ND/2744959 Microsemi A3P030-QNG48, a ProASIC3/E FPGA with 34 3.6V-tolerant GPIO and "30000 gates". "Ideal for CPLD replacement." "1.5 V single voltage operation." "350 MHz system performance." 1024 bits of flash that it can't write to (and also its programming is stored in flash; it doesn't have to load a bitstream to boot). Consists of 768 "VersaTiles" which are three-input arbitrary functions registered with D flip-flops, but no RAM. No AES protection on its JTAG ISP. Quiescent supply current is 2 mA, enormous compared to the microcontrollers. Maximum JTAG TCK clock is 19MHz. Bitstream format and debugging features are apparently not documented.

• 1,953 available of the US$2.96 http://www.digikey.com/product-detail/en/lattice-semiconductor-corporation/LCMXO256C-4TN100C/220-1048-ND/2641849 Lattice LCMXO256C-4TN100C, marketed as a CPLD, from the "MachXO" family. 256 LUT4s, no external configuration memory (and "instant-on"), fast SRAM reconfigurability, normally programmed via JTAG.

Tantalum capacitors: 54077 items

• 1,534,001 available of the US$0.84 http://www.digikey.com/product-detail/en/avx-corporation/TAJB226M010RNJ/478-3040-6-ND/1717036 AVX TAJB226M010RNJ 22µF ±20% 10V 2.4Ω capacitor in a 3.5 mm × 2.8 mm package. This is not the highest-energy-capacity capacitor in the series (that would be the 25V 150µF type, holding 47 mJ, like the TAJV157M025#NJ in a 7.3 mm × 6.1 mm case, though Digi-Key doesn't carry it; they'll sell you the lower-voltage 20V type TAJV157M020RNJ at US$2.17 but only in lots of 400) but it still has a pretty impressive energy density. These are advertised as "standard tantalum" capacitors; I think that means they don't have wet electrolyte. AVX's wet tantalum page explains, "AVX's wet tantalum capacitors offer higher capacitance and voltage capability than solid tantalum capacitors." On another page, they say, "We are the global leader in MnO_2 solid tantalum technologies such as smallest case size MnO_2, highest temperature capabilities up to 230°C and lowest DCL product offering," and the TAJ series is indeed on that page.
• 792,881 available of the US$0.38 http://www.digikey.com/product-detail/en/kemet/T491A106K010AT/399-3684-1-ND/819009 Kemet T491A106K010AT 10µF 10V 3.8Ω 3.2mm x 1.6mm tantalum capacitor. This is also a MnO_2 capacitor.
• 761,191 available of the US$1.49 http://www.digikey.com/product-detail/en/kemet/T491D107K016AT/399-3770-1-ND/819095 Kemet T491D107K016AT 100µF 16V 0.7Ω 7.3mm × 4.3mm tantalum capacitor. This one is rated for 2000 hours, even though it's I think also a MnO_2 capacitor. This is really cheap for such a high-energy capacitor.
• 4169 available of the US$8.91 http://www.digikey.com/product-detail/en/vishay-sprague/597D157X9025F2T/718-1632-1-ND/1973910 Vishay 597D157X9025F2T, the cheapest 150 µF 25 V capacitor I could find.
• 2278 available of the US$9.55 http://www.digikey.com/product-detail/en/vishay-sprague/597D227X0025M2T/718-1945-1-ND/3985794 597D227X0025M2T, which is the 220µF version, with 37% more energy capacity per dollar.

Some interesting notes from the above list

• High-speed serial interfaces usually are SPI rather than I²C.
• Although sub-picojoule-per-instruction CPUs exist in research labs, the ones you can actually buy are up in the hundreds-of-pJ range.

Topics

Laser cut next step

Kragen Javier Sitaker, 2018-04-27 (updated 2018-04-30) (7 minutes)

I find myself in need of objects that can be easily laser-cut. The deskbox for the office is an example, but there are others: chairs, storage boxes, etc.

I've been sitting on this issue for more than a month since I moved. In a way, over a year, kind of due to lack of money. But now I have money, and I need to make my things manageable. And I need to return to the learning cycle.

So far most of the pieces I've cut have been 0.01 m², with the exception of that one that was about 0.1 m² and didn't turn out that well. The deskbox is about 1 m²; the bedbox is about 10 m². So I should make some smaller things first.

Draining rack

At the 0.02 m² level, I could benefit immediately from a draining rack for cutlery, maybe with a 100 mm x 200 mm bottom and 150 mm walls, and bottom hole diameter of no more than 8 mm so the steak-knife tips can't poke through very far. Also, for cleanability, it would be nice for the bottom to be as thin as possible and have relatively round holes, and it would be nice for the joints to have some kind of inside radius or at least be relatively rounded for the same reason, and to be connected in ways that don't leave cracks for mold to grow in.

This couldn't be made of MDF; acrylic and HIPS are options. (Max58 has clear, black, and white acrylic in 2.4, 3, 4, 5, and 8 mm, and HIPS in 1 mm.) Both acrylic and HIPS can be solvent-welded with acetone or MEK (and maybe ethyl acetate, I'd have to try it) but resist alcohol. Heat welding and caulking with silicone are plausible alternatives.

There is a good introduction to solvent welding on the NerfHaven forum. It has a short list of acetone-resistant plastics: Delrin, UHMWPE, nylon, Teflon. Nylon and Teflon resist every solvent on the list, and I suppose epoxy would too. Paint thinner may be my best bet in Argentina.

The bottom holes can be relatively infrequent; they don't have to fill most of the bottom. The bottom will bow downward with the weight of the cutlery, so I need a bottom hole in the middle, at least.

HIPS supposedly has 40% elongation at break, so I should be able to curve it into really ridiculously curvy shapes, to the point of origami. (The ratio of its 32 MPa UTS to its 1.9 GPa tensile modulus, however, suggests more like 1.7% elongation.) Maybe actually the best approach would be to cut some darts into the edge of a single piece and then weld them shut to make a curvy shape.

PMMA, by contrast, only has 4% elongation at break, which beats the hell out of MDF but isn't very much. And I'm more sanguine about its food safety. So the 2.4 mm PMMA could presumably be compressed 4% on its inner face and stretched 4% on its outer face at 1.2 mm, giving a curve radius of 1.2 mm/4% = 30 mm. This rather implausible result suggests that I should be able to bend it into 600mℓ bottles. I think they would be fairly prone to exploding, but maybe

they could work.

PET panes of 0.3mm are available from Maderera Gascón at a lower price than acrylic. These are AR$115 for 1×2 m.

The whole cutlery drainer would be (+ (* 2 200 150) (* 2 100 150) (* 200 100)) = 110000 mm² or 0.11 m². If I make it as a box from separate pieces, the biggest separate pieces are the 200×150 sides, which are 0.03 m², which is the right order of magnitude for this project.

The simplest possibly usable design would be a box with some hooks on the side, with a few tabs to keep the pieces aligned while I glue them: (+ (* 2 2 (+ 200 150 100 150)) (* 2 (+ 200 100))) = 3000 mm of box edges to cut, plus the tabs, the hooks, and the drainage hole. Maybe 2 minutes of cutting.

Or maybe two 100mm-radius approximate semicircles (.0314 m² each) plus a 100mm×314mm rectangle (also 0.0314 m²) to bend into a semicircle, with a drainage hole in the middle, a couple of tabs to hold it together for gluing, and a couple of hooks on the side. That's (+ (* 2 (+ 100 314 200 314))) = 1856 mm of edges to cut, plus the drainage hole and the hooks.

Or maybe both. Or more possibilities. If I use 3mm acrylic, it should be dimensionally compatible with MDF Heckballs, but awesomely transparent and stuff. If the acrylic sheet is 600×400 (.24 m²) I have room for about 8 pieces of this size, so basically both cutlery drainers and not much else.

A 600×400 mm sheet of acrylic might cost AR$300 or AR$400. If cutting still costs AR$0.40 per second and cutting is 24 mm per second plus 60 ms per vertex, we have only about 90 seconds of cutting for the semicircle thing and another 200 seconds of cutting for the box (99% edge, 1% vertex), totalling like 300 seconds and AR$120. This suggests that the quality of the product could be improved dramatically at minimal extra cost; doubling the cut time by using 100× as many vertices would be a minimal problem, for example, and there's plenty of time to engrave surfaces and scallop edges and stuff.

Bowls

While I'm at it maybe I should make some bowls or something. PMMA is about 1.2 g/cc, so 8mm PMMA should weigh almost 1g/cm². So an 0.03 m² bowl (crudely: 0.01 m² of base, plus four sides of 0.005 m² each — 100 mm × 100 mm × 50 mm) would be 300 g.

Other stuff

I've moved again! Now I have no housewares.

I need fans, and I have motors and power supplies. A pinwheel-like structure should be easy to cut out of thin HIPS; also I should be able to assemble something out of sheets of acrylic or even MDF.

I now need an entire draining *rack*.

I'd like to make some IQLights. These can be laser-cut out of HIPS as well, I think.

I'd like a dehydrator, an apparatus that runs heated air through a serpentine airflow with food or other things on shelves. I'd like a dehydrator for food, a dehydrator for garbage (so the garbage doesn't become food), and a dehydrator for laundry, although that may be aiming a bit high.

I need chairs. A relatively small amount of PMMA should suffice to make a sittable chair, but it will be somewhat fragile, and potentially cut you if it breaks.

Sheets of PMMA should work adequately as cutting boards.

Topics

- Materials (p. 3560) (112 notes)
- Digital fabrication (p. 3411) (42 notes)
- Sheet cutting (p. 3710) (10 notes)
- Laser cutters (p. 3540) (10 notes)

On the method of finite differences used in Babbage's Difference Engine

Kragen Javier Sitaker, 2019-05-31 (6 minutes)

The "method of finite differences" as used in the Difference Engine is closely related to, but slightly different from, Newton's method of divided differences (used, for example, for polynomial interpolation or for boundary-value problems in ordinary differential equations) and the finite difference method used in the solution of ODEs and PDEs.

The Wikipedia page on the Difference Engine explains how to calculate the initial values, but I am skeptical of its explanation. However, it cites Nathan Myhrvold's instructions for setting up the machine, which we can suppose have been tested on the one he had built.

Reading F. Baily's 1823 comments on the Difference Engine we find:

> 2°. Tables of Square Numbers. The squares of all numbers, as far as 1000, were a long time ago published … In computing a table of this kind by the machine, even if extended to the most remote point that could be desired, the whole of the mental labour would be saved: and when the numbers 1, 1, 2 are once placed in it, it will continue to produce all the square numbers in succession without interruption. This is, in fact, one of those tables which the engine already made is capable of computing, as far as its limited number of wheels will admit.
>
> 3°. Tables of Cube Numbers. Tables of this kind have likewise been already computed… In computing such a table by the machine, the whole of the mental labour would be in this case also saved: since it would be merely necessary to place in the machine the numbers 1, 7, 6, 6; and it would then produce in succession all the cube numbers.

The table of cubes begins 1, 8, 27, 64, 125; its first differences are 7, 19, 37, 61…, and its second differences are thus 12, 18, 24…, its third differences 6, 6…, and its fourth differences merely a sequence of zeroes, since the third-order approximation is actually precisely correct. The first item of each of these sequences gives the (1, 7, 6, 6) setup described in Baily's article.

XXX no it doesn't; 6 is the difference between 1 and 7, so I think Baily actually took into account the half-cycle offset I claim below that he didn't take into account; his numbers give the correct answer as shown at the end.

However, to update this state (1, 7, 6, 6) to its successor state (8, 13, 12, 6) without any extra storage, we must work strictly from left to right, because the previous value of each number (except the 1, in the lowest-order register, T) is needed to update the next-lower-order term, before being itself updated (except the second 6, in the highest-order register, Δ^3). So we must proceed as follows:

T	Δ^1	Δ^2	Δ^3
1	7	6	6
8	7	6	6
8	13	6	6
8	13	12	6

XXX note that 13 is wrong because it should be 19

However, at least in the "Difference Engine No. 2" design Babbage developed between 1846 and 1849 (23–26 years after Baily's letter), the calculations are not performed in this serial fashion. As explained in Swade's analysis, instead a set of four parallel additions is carried out, then a set of three parallel additions:

However, the sequence of additions as executed by the engine does not proceed in a stepwise way from right to left as one might expect from the manual method. One complete calculating cycle consists of two symmetrical half-cycles. During the first half-cycle the number values on the odd-numbered axes are simultaneously added to those of the even-numbered axes to the immediate left i.e. axes 1 [Δ^7], 3 [Δ^5], 5 [Δ^3], 7 [Δ^1] are added to 2 [Δ^6], 4 [Δ^4], 6 [Δ^2],and 8 [T] respectively. Similarly, during the second half-cycle all the even-numbered axes are simultaneously added to the odd-numbered axes again to the immediate left. Provided this is taken account of when setting up at the start of a run by offsetting the initial values on alternate axes by a half-cycle, the end result is the same i.e. each full calculating cycle results in a new tabular value which is the cumulative sum of the number of active differences.

So, in fact, if we want to compute a table of cubes, to get 8 as our second cube, we do indeed need to set Δ^1 (column 7 on the machine) to 7, as Baily says. And then, having been used, it is incremented by Δ^2 (column 6 on the machine) in the second half-cycle — but that Δ^2 had previously been incremented by Δ^3 in the first half-cycle. So the correct initial state we would need is not Baily's (1, 7, 6, 6) but (1, 7, 0, 6), proceeding as follows:

T	Δ^1	Δ^2	Δ^3
1	7	0	6
8	7	6	6
8	13	6	6

XXX this is the wrong answer. If we start from Baily's setup we get the right answer:

T	Δ^1	Δ^2	Δ^3
1	7	6	6
8	7	12	6
8	19	12	6
27	19	18	6
27	37	18	6
64	37	24	6
64	61	24	6
125	61	30	6

Topics

- Math (p. 3564) (78 notes)
- History (p. 3500) (71 notes)

Jim Weirich's death and my daily life

Kragen Javier Sitaker, 2014-04-24 (5 minutes)

Today, 2014-02-20, I learned that my friend Jim Weirich died yesterday. No word yet on the cause of death, except for a pseudonymous rumor on Hacker News that he died of a heart attack, which is quite plausible. If I remember correctly, Jim was the guy who guided me through understanding the applicative-order Y combinator during the previous millennium, and who has also given me one of the nicest compliments I've ever received. I suppose this is an example of that saying: they won't remember what you said, and they won't remember what you did, but they will remember how you made them feel; because I'm not even sure about the Y combinator thing.

I don't know how to describe the loss to the world that is Jim's death. He worked on code that was useful to millions of people, shared his knowledge, and was always willing to mentor people; but, also, and maybe more importantly, he was one of the least assholish people I know. A militant atheist mutual friend of ours complained to me once that Jim was a fundamentalist Christian, which could be true for all I know; I never asked him, and he never brought it up. In many ways, he was a paragon of loving your neighbor as yourself.

I don't remember when the last time I talked to Jim was. It's probably public on Twitter.

Today I didn't get much done at work. A little bit, but not enough. Tomorrow I go in early, which might help or might hurt.

Jim was 57, about 20 years older than I am. Every year now for several years, some of my friends have died. They say that your risk of death per year increases by about tenfold every ten years, and if that's true, it's also true of your group of friends, if they continue to have the same age distribution relative to you. So in my 20s this rarely happened, and now it's happening once or twice a year. I guess that means that when I'm 47, I can expect one of my friends to die every month, unless I start hanging out with a younger crowd.

I've just arrived home from work at about 21:30. Stace had messaged me a few hours earlier to let me know power was out at the house. As I approached home on my bicycle, I noticed that several blocks of buildings and streetlights were dark.

I had been hoping power would be back on, because I was looking forward to answering a sweet email someone had sent me earlier today, and I didn't bring a copy of it with me. And I was hoping perhaps to work on a search-engine project for my chapter in an upcoming book, and it turns out I don't have the current version of the code with me on this netbook.

The upstairs neighbor is peeing, a sound that always makes me nervous when I hear it in my bedroom.

My bedroom is beginning to smell of exhaust from poorly maintained gasoline engines, probably from neighbors who have turned on their generators.

I went to work today by bicycle for the first time in a while. My

rear tire had gone flat, so I pumped it up; but after a few blocks, the seam on the inner tube split and it went totally flat again. After I wandered around for a while (forgetting the bike map I have in my backpack, which showed two bike shops within a few blocks) I found a bike shop and got a new inner tube for $70 (US$6.50). I'd've bought an extra one to carry as a spare, but I didn't have the cash on me.

On the way home, it went flat again, but more slowly, so I was able to make it home by pumping it up. I guess I need to see if I have a puncture and patch it by this weekend.

Last night, Stace, her dad, and I went over to a friend's house, who showed us his new video game. Although he told me the name several times, I can't remember it now. It's a guitar trainer that analyzes the analog signal from your electric guitar to teach you to play the chords and riffs of popular rock songs, including tricks like pitch bending. "Look, Stace!" I said. "This is the future of education!" She was unimpressed. (My track record at predicting the futures of things is not that great.)

Oh! Power's back on!

Topics

- Pricing (p. 3646) (89 notes)
- Argentina (p. 3325) (12 notes)
- Journal (p. 3532) (11 notes)
- Death (p. 3403) (2 notes)
- Bicycle

Hadamard rhythms

Kragen Javier Sitaker, 2019-11-01 (6 minutes)

I was thinking about synthesizing drum loops and it occurred to me that Hadamard bases might be an interesting way to do it.

A simple approach to synthesizing a drum loop with a single instrument and up to 8 beats is to just have 16 variables for the volume and timing offset of each of the 8 beats. Some of the volumes might be 0, silencing those beats; some of the offsets might be nonzero, swinging those beats a bit.

A problem with this representation is that usually you don't want to adjust just one beat at a time; you might want to do something to all the odd beats, for example, or all the even beats. A possible solution to this is to transform 8 volume variables and 8 timing-offset variables each through a 8x8 Walsh matrix, here given in "sequency order" rather than as a Hadamard matrix:

```
 1  1  1  1  1  1  1  1
 1  1  1  1 -1 -1 -1 -1
 1  1 -1 -1 -1 -1  1  1
 1  1 -1 -1  1  1 -1 -1
 1 -1 -1  1  1 -1 -1  1
 1 -1 -1  1 -1  1  1 -1
 1 -1  1 -1 -1  1 -1  1
 1 -1  1 -1  1 -1  1 -1
```

Given an appropriate constant scaling factor, this is an orthonormal basis for 8-vectors.

A more convenient form might be the ReLU of the same matrix, which you will note is still symmetric though no longer normalized:

```
1  1  1  1  1  1  1  1
1  1  1  1  0  0  0  0
1  1  0  0  0  0  1  1
1  1  0  0  1  1  0  0
1  0  0  1  1  0  0  1
1  0  0  1  0  1  1  0
1  0  1  0  0  1  0  1
1  0  1  0  1  0  1  0
```

So, for example with volumes, the first variable is a master volume control; the last variable is a master volume control for the major beats (assuming we're in 4/4 time); the second variable controls the volume on the first half of the measure; the fourth variable controls the first and third beats (the "on" beats, giving you a backbeat if you turn them down); the fifth variable controls the first halves of the first and third beats, and the second halves of the second and fourth beats; and so on. It might make the most sense to have scale values that extend from zero amplitude to somewhere above 1.0, but do the matrix-multiply in decibels, so any single slider is capable of silencing either half or all the beats, and can either attenuate or amplify those beats from the overall reference volume level.

In this way you can adjust the volumes of the 8 beats to any desired

combination of volumes, but every slider affects half the beats in what I speculate will be a more musically appealing way.

The same scheme can be used for adjusting timing, though perhaps there is no need for the first variable in that case. If the last timing variable (10101010) is pushed to an extreme, it could perhaps push the eighth notes to coincide with the quarter notes.

Here's the corresponding 4x4 binary matrix:

```
1 1 1 1
1 1 0 0
1 0 0 1
1 0 1 0
```

And the corresponding 16x16 binary matrix:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0
1 1 0 0 1 1 0 0 0 0 1 1 0 0 1 1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1
1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 0
1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1
1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0
1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1
1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 0
1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
```

This was derived as follows:

```
def hadamard(n):
    return ([[1]] if n < 1 else
            [xs + xs               for xs in hadamard(n-1)] +
            [xs + [-x for x in xs] for xs in hadamard(n-1)])

def walsh(n):
    return sorted(hadamard(n),
        key=lambda xs: sum(1 for i in range(1, len(xs))
                           if xs[i] != xs[i-1]))

def walsh_order(n):  # bit-reversed reflected binary Gray code
    return ([0] if n < 1 else
            [x * 2     for x in          walsh_order(n-1) ] +
            [x * 2 + 1 for x in reversed(walsh_order(n-1))])

def binrow(row):
    return ('1' if x > 0 else '0' for x in row)

if __name__ == '__main__':
    for row in walsh(4):
        print(' '.join(binrow(row)))
```

It is possible to use the fast Hadamard transform to efficiently transform between the two representations — the set of user parameters and the set of music parameters — but that isn't really important in this case.

Topics

Sparse filters

Kragen Javier Sitaker, 2018-12-02 (4 minutes)

I'm interested in sparse filters, in the sense that you can realize them with only a small number of taps to reduce the number of multiplications, and multiplication-free or multiplication-light filters, in the sense that the nonzero tap coefficients are numbers like 1, 2, 3, 4, or maybe 6 or 8, but not things like 1.03594513, except perhaps in a very few cases.

The Hogenauer cascaded integrator-comb filter is a well-known filter of this class, commonly used for sample-rate conversion. But here are a few other related ideas.

The context for this is that a lot of our filter design lore comes from the world of analog electronics, where multiplication is trivial and memory is hard. This means it is not a good fit for digital computation, where memory is trivial and multiplication is hard, although immense hardware effort has been devoted to papering over this for the benefit of DSP designers.

A CIC filter is low-pass but linear-phase, so you can invert it — subtracting the appropriately scaled low-pass signal from the input sample in the middle of the kernel — to get a high-pass filter. Or you can subtract the outputs of two such filters to get a bandpass filter. This may be particularly useful in combination with undersampling — decimating the bandpass-filtered signal to alias the band of interest down to IF or baseband, thus allowing you to detect a high-frequency signal without doing anything high-frequency except for running some integrators.

A unity-gain negative-feedback comb filter $y(n) = x(n) - y(n-k)$ is an oscillator, and indeed it's very close to the Karplus-Strong oscillator (which is, in its original form, $y(n) = x(n) - ½y(n-k) - ½y(n-k-1)$, to gradually attenuate higher frequencies). If you compose it with a unity-gain feedforward comb filter in very much the same way you do in a CIC filter, its impulse response is a finite-length alternating impulse train, so it's a bandpass filter, though it also detects harmonics of the target frequency. A cascade of a few of these approximates a Gaussian window; if you add an actual CIC filter, which you can tune to have nulls at the harmonics of the target frequency, you can get a very inexpensive high-Q filter. As one example, I got a bandpass Q of 17.8 and 38dB stopband attenuation to generate I and Q signals for oscillations with a period of 60 samples using 2.64 additions and subtractions per sample: a two-stage CIC filter with lags of 36 and 40 samples on the front end, three 30-sample feedback combs, and further feedforward combs of 300, 480, and 780 samples. This gives a kernel with a temporal response of about 1000 samples (FWHM) which is a reasonably good approximation to a Gabor filter with $Q \approx 17$.

(You'll note that this is very similar to the previous technique, and the two may be alternatives; with the previous technique, for example, it may be useful to set up the low-pass filter you're inverting to have precise nulls at the harmonics of the signal you're trying to detect.)

More generally, if you can construct a filter whose impulse

response is half a wave or more of some waveform you want, you can cascade it with the unity-gain negative-feedback comb filter and get an oscillator for that waveform, or more or less equivalently, a filter that matches it. And you can use the same trick described above to construct a filter whose impulse response is a specific number of oscillations of the waveform.

Anything you can construct by convolving, adding, and subtracting box filters and comb filters can be computed as a sparse filter. So, for example, you can get a difference of gaussians fairly easily.

Once we get into nonlinear filter territory, things get more interesting still. You can do a PLL as a pretty sparse filter, for example.

Topics

Solving initial-value problems faster and with guaranteed error bounds with affine arithmetic

Kragen Javier Sitaker, 2019-04-02 (5 minutes)

Can we improve the solutions of initial-value problems using affine arithmetic?

An IVP consists of some set of differential equations, with all the derivatives given with respect to a single "time" independent variable, and a set of initial conditions that hold at some single point in time. Standard ways to solve them numerically include Runge-Kutta integration. I wonder if we can make some improvements by using affine arithmetic.

Affine arithmetic

Affine arithmetic is a generalization of interval arithmetic which can give tighter bounds under many circumstances, and additionally can capture some of the dependency of outputs on inputs. Instead of representing each quantity as $a \pm b$, as interval arithmetic does, affine arithmetic represents each quantity as an affine combination of variables $c_0v_0 + c_1v_1 + c_2v_2 + \ldots\ c_nv_n + k$. Most of the v_i are assumed to range over some range like [-1, 1] or [0, 1], and initially they each represent the uncertainty associated with a given input.

In ordinary affine arithmetic, a new v_i is introduced at every nonlinear or rounded calculation to represent the uncertainty about the result of that operation. This allows the uncertainty due to a given input, calculation, or rounding to cancel: $(c_{00}v_0 + c_1v_1 + k_0) - (c_{01}v_0 - c_1v_1 + k_1) = (c_{00} - c_{01})v_0 + k_0 - k_1$, canceling the uncertainty due to v_1 (and some of the uncertainty due to v_0 as well, if c_{00} and c_{01} have the same sign). Depending on the situation, this subtraction may introduce an additional variable to account for the rounding error of this subtraction; the number of such variables can increase without limit, making successive operations successively more costly.

By contrast, in *reduced* affine arithmetic ("RAA"), all the extra uncertainty introduced during the calculation (due to rounding or nonlinear operations) is dumped into a special c_nv_n which doesn't cancel in this way, instead growing like ordinary interval arithmetic. This caps the computational cost of reduced affine arithmetic at the cost of providing more pessimistic error bounds — but maybe only very slightly more pessimistic, if most of the uncertainty of the result is due to the initially-present input uncertainty, not introduced during the calculation.

Other variants of RAA may introduce new variables during the calculation at some times, but not others.

IVPs with affine arithmetic

So I have some vague thoughts about how affine arithmetic might help with IVPs. You can recursively divide the state space of your system into boxes and compute an affine approximation of the derivatives in question inside each box, rather than just

approximations around some points. Each box is a paraxial parallelepiped some of whose axes are dependent variables, but one of whose axes is the time variable.

This allows you to compute trajectories of the system through this multidimensional state space; the arc through a given box is in general some kind of exponential, but you can of course approximate it with an affine form, subdividing the trajectory more finely than the derivatives if necessary. Then, once you have an estimated result with its error bounds, you can consult its coefficients to see where the most uncertainty comes from — is it the value of x_3 in box 5, or the value of x_5 in box 8? This allows you to intelligently choose a box to subdivide to improve the approximation. Also, it allows you to intelligently choose a dimension along which to subdivide it to improve the approximation, which becomes progressively more important with larger dimensionality. (Even if you have only a single independent variable, you might have a large state vector evolving along it.)

This also allows uncertainty to be associated with the initial values — in effect you can compute the trajectory of not just a single point but an entire neighborhood of points. This neighborhood, too, might need to be subdivided if the system distorts it into a shape whose nonlinearity exceeds your desired precision.

Extension to BVPs

Boundary-value problems specify known values at not just a single point but potentially many points, and possibly have many dimensions; liquid flow, for example, or heat flow might have boundaries where the flow is zero or the temperature or heat flow is constant. You could similarly imagine recursively subdividing the state space, again partitioning it using both state variables and (now plural) independent variables, computing an affine approximation to the derivatives of the state variables for each box. Perhaps the final result is a piecewise-linear approximation to each state variable, with the pieces being boxes defined over the independent variables, with error bounds on it.

Topics

IRC bots with object-oriented equational rewrite rules

Kragen Javier Sitaker, 2007 to 2009 (6 minutes)

(Previously published on krageon-tol.)

Thinking about my "object-oriented equational rewrite rules" and IRC bots.

Suppose we think of definitions like

```
c = (f - 32) * 5 / 9
```

as defining properties that exist on every object that meets their qualifications (and isn't shadowed by some previously-existing c). Then we could think of something like this:

```
todayweather = { f: 95, windspeed: 37 }
```

as defining a property 'todayweather' that exists on every object. And then you could plausibly ask, on an object that has only the properties that all objects have (the Ur-Object), what are the properties that have a c property?

```
c?
        { todayweather: 36.1, normalweather: 25 }
```

Namespaces

If you're doing this on an IRC bot on a casual channel, you could have a namespace per nick, and only enumerate properties in your own namespace. So there might be [kragen]c and [Isomer]c; but I could refer to properties from other namespaces explicitly if I wanted to; simply importing:

```
f = [isomer]f
```

or using without importing:

```
c = ([isomer]f - 32) * 5 / 9
```

Actual Programs

Let's hold off for now on any way to make these properties (as opposed to their value on a particular object) first-class in the language. But let's have self-quoting keywords, like in Prolog or Erlang (lower-case), or Common Lisp or Ruby (with colons). I'm pretty sure that if I use lower-case for auto-quoting, I'll end up with Very Important-Looking Programs, so I'm thinking I'll use backquote (`) instead.

Let's start with a little syntactic sugar for lists. Let's say that if we have some semicolon-terminated expressions inside parentheses, with the last semicolon optional, it is syntactic sugar for a list made of conses:

```
( a; ) => { x: a, y: `nil }
( a; b; c ) => { x: a, y: { x: b, y: {x: c, y: `nil } } }
() => `nil
```

Even without first-class properties, we can still write:

```
(`nplusone; n).apply = n + 1
```

such that we can write

```
(`nplusone; 3).apply
```

and get 4.
Given that, we can write map:

```
(fn; ()).map = ()
(fn; {x, y}).map = {x: (fn; x).apply, y: (fn; y).map}
```

And filter:

```
(fn; ()).filter = ()
(fn; {x, y}).filter = (fn; x; (fn; x).apply; y).filter2
(fn; x; `t; y).filter2 = { x: x, y: (fn; y).filter }
(fn; x; `f; y).filter2 = (fn; y).filter
```

It's pretty brutally obvious that we need some special syntax for cons here. Let's try infix @.

```
(fn; ()).map = ()
(fn; x @ y).map = (fn; x).apply @ (fn; y).map
(fn; ()).filter = ()
(fn; x @ y).filter = (fn; x; (fn; x).apply; (fn; y).filter).filter2
(fn; x; `t; y).filter2 = x @ y
(fn; x; `f; y).filter2 = y
().length = 0
length = 1 + y.length
x.reverse = ((); x).reverse2
(a; ()).reverse2 = a
(a; x @ y).reverse2 = (x @ a; y).reverse2
```

That works OK, and I could imagine people typing it in IRC. It would probably be good to eliminate the argument-order dependencies as much as possible, and reduce the number of properties defined on all lists of length two or three. In the below, {foo, bar} is short for {foo: foo, bar: bar}, and any free variables are required properties of the object the property is defined on.

```
{list: ()}.map = ()
map = (fn; list.x).apply @ {fn, list: list.y}.map
{list: ()}.filter = ()
{list: x @ y}.filter = {fn, x, include: (fn; x).apply,
                        y: {fn, list: y}.filter}.filter2
{include: `t}.filter2 = x @ y
{include: `f}.filter2 = y
().length = 0
length = 1 + y.length
```

```
x.reverse = { reversed: (), left: x }.reverse2
{left: ()}.reverse2 = reversed
{left: x @ y}.reverse2 = { reversed: x @ reversed, left: y }.reverse2
```

Those aren't quite so brief, but they're still within the length where people could plausibly type them in a conversation.

So how about strings? Let's suppose that strings support the interface of lists of one-character strings, work properly in pattern-matching, and that one-character strings additionally have a .ord property that tells you their ASCII code. Can we split words on spaces?

```
" ".space? = `t
{ord}.space? = `f
({space?: `t} @ y).words = y.words
({space?: `f, x} @ y).words = {wletters: (x;), left: y}.word
{left: {space?: `t} @ rest}.word = wletters.reverse @ rest.words
{left: x @ rest}.word = {wletters: x @ wletters, left: rest}.word
{left: ()}.word = (wletters.reverse;)
```

That's not too bad. It compares favorably to Scheme in total code volume:

```
(define (words string) (words-of-list (string->list string)))
(define (words-of-list clst) (map list->string (words-list clst)))
(define (words-list clst)
  (let ((x (car clst)) (y (cdr clst)))
    (if (char-whitespace? x) (words-list y) (word (list x) y))))
(define (word wletters left)
  (cond ((null? left) (list (reverse wletters)))
        ((char-whitespace? (car left))
         (cons (reverse wletters) (words-list (cdr left))))
        (else (word (cons (car left) wletters) (cdr left)))))
```

The Scheme is 10 lines, 59 words, 498 characters, in place of 7 lines, 47 words, 299 characters.

So from there I think you could quite reasonably, e.g. implement ternary trees and compute a time-efficient inverted index of a big bag of strings. Say, old chat lines, or a small number of HTML documents.

Precedence

When you have more than one rule that's applicable to finding a property value, you have to decide what to do. Should you use the first rule, use the second rule, or combine them somehow?

Aardappel used specificity ordering. I think you should do the same here, given that the "source code" is a bunch of people chatting. But it would probably be helpful if the specificity ordering were partial, so that the users would have to resolve potential conflicts manually.

Syntax

The question of syntax is a little ugly but probably soluble. The normal namespace syntaxes are as follows:

Syntax	Precedents	Why Not
isomer/f	Unix	used for division
isomer\f	MS-DOS	painful memories
isomer.f	C++, Java, Python	used for property access
isomer:f	MacOS, XML	used for property definition
isomer::f	C++, Perl	too verbose
isomer f	Smalltalk	not verbose enough
[ISOMER]F	VMS	forgotten
isomer'f	Ada, Perl4	painful memories, unbalanced '
isomer$f	R, DCL	painful memories, ugly

There are some other alternatives, especially if we involve Latin-1:

```
isomer|f isomer»f isomer«f isomer§f isomerºf isomer->f
isomer>f isomer¦f isomer×f isomer÷f isomer¬f isomer®f isomer©f
isomer±f isomer£f isomer·f isomer=>f isomer=-f isomer:-f
isomer--f isomer-)f isomer-»f isomer°f isomer*f isomer~f
isomer!f isomer[]f isomer()f <isomer f> <isomer>f isomer<f>
isomer_f @isomer.f
```

Topics

OMeta contains Wadler's "Views"

Kragen Javier Sitaker, 2007 to 2009 (updated 2019-05-20)
(13 minutes)

I think.

Views

Philip Wadler proposed "views" in a 1986 paper (see References.) This section just explains pattern-matching, views, and their relationship to various programming languages.

ML, Haskell, Aardappel, and many other functional programming languages (plus Prolog) support discriminated unions ("sum types" or I think "algebraic data types") that can be used in "pattern matching" that combines case-analysis control flow with data access. Here's an example program in OCaml that defines a single type called `tree` and uses it to sort a list of strings.

```
(* a simple binary tree sort program *)

(* an object of type "foo tree" can be either a Leaf, containing
   no data, or a Fork, containing another "foo tree", a "foo", and
   another "foo tree". *)
type 'foo tree = Leaf | Fork of 'foo tree * 'foo * 'foo tree ;;

(* Here we define a function whose unnamed second argument is a tree.
   If it's a Fork, we bind the names "left", "key", and "right" to its
   data contents, and which of the two branches we take is determined
   by whether it's a Leaf or a Fork. *)
let rec tree_insert datum = function
    Leaf                   -> Fork (Leaf, datum, Leaf)
  | Fork(left, key, right) -> if datum < key
      then Fork(tree_insert datum left, key,                   right)
      else Fork(                left, key, tree_insert datum right) ;;

(* Here we pattern-match on a list, which can be either the empty
   list, or a list consisting of a first item "h" followed by a list
   of zero or more other items "t". *)
let rec make_tree old_tree = function
    [] -> old_tree
  | h :: t -> make_tree (tree_insert h old_tree) t ;;

let sorted_tree = make_tree Leaf ["t"; "r"; "e"; "e"; "t"; "o"; "p"; "!"] ;;

(* Here we use pattern-matching to iterate over the data of the tree, in
   what will be sorted order if the tree was built with the tree_insert
   function above. *)
let rec tree_iter f = function
    Leaf -> ()
  | Fork(left, key, right) -> tree_iter f left; f key; tree_iter f right ;;

tree_iter print_endline sorted_tree ;;
```

The trouble with the pattern-matching is that it breaks data

abstraction. The interface to the data structure is the same as its implementation, at least for those who want to pattern-match on it instead of just passing it to other functions. Changing the implementation implies changing all of the code that pattern-matches on it, and possibly changing that code to not use pattern-matching.

Wadler gives several examples:

• small integers are most efficiently represented as machine words containing binary numbers, but some kinds of functions clearer when written with pattern-matching on Peano notation (`Succ(Succ(Zero))`) or explicit binary (`Twice(Twiceplusone(Twiceplusone(Zero)))` for 6).
• complex numbers have imaginary and real parts, or a magnitude and an angle, depending on how you look at them; in languages like Haskell, only one of these can get the brevity benefit of pattern-matching.
• lists are conventionally viewed, as above, as being either empty or consisting of a first item and a list of other items; some algorithms are more conveniently expressed if the second alternative is instead a last item and a list of other items. A "rope" representation can provide reasonable performance for both of these "views" of the list.

Wadler's suggested explicit-binary view looks like this:

```
view int ::= Zero | Even int | Odd int
  in n = Zero,              if n = 0
       = Even (n div 2),    if n > 0 && n mod 2 = 0
       = Odd  ((n-1) div 2), if n > 0 && n mod 2 = 1
  out Zero     = 0
  out (Even n) = 2 * n,     if 2 * n > 0
  out (Odd n)  = 2 * n + 1, if 2 * n + 1 > 0
```

Here `Even` means "twice", and "Odd" means "twice, plus one". His `out` "function" works by pattern-matching, just like the OCaml functions above; and the `in` "function" can do the same. The `in` and `out` clauses explain how to translate between the `Zero`/`Even`/`Odd` view of integers and the language's native representation of integers.

In Wouter van Oortmerssen's Aardappel, the problem is diminished; if you define a new representation of some abstract data type, then in a single place, you can provide new clauses for all the functions that pattern-match on the old representation. However, the problem still exists, since you have to do an amount of work that's proportional to the number of places that pattern-match on the old concrete data type.

In 2007, I wrote about object-oriented equational rewrite rules which are basically nothing but views. I still haven't implemented the system I described in that post.

OMeta

OMeta is a system intended to dramatically simplify compiler development. Its authors write:

> Several popular programming languages --- ML, for instance --- include support for pattern matching. Unfortunately, while ML-style pattern matching is a great tool for processing structured data, it is not expressive enough on its own to support more complex pattern matching tasks such as lexical analysis and parsing.

But what is weak about ML-style pattern-matching? Why can't it

do lexical analysis and parsing, if you have (perhaps lazy) lists of characters or tokens?

OMeta in terms of Views

In fact, I suspect that views, plus non-backtracking pattern matching, plus some syntactic sugar, more or less equals OMeta. Here's the first sample grammar from the OMeta paper, modified to remove left recursion; incidentally, this is roughly Figure 3-1 from Bryan Ford's PEG parsing thesis:

```
meta E {
  dig ::=  '0' | ... | '9';
  num ::=  <dig>+;
  fac ::=  <num> '*' <fac>
         | <num>;
  exp ::=  <fac> '+' <exp>
         | <fac>;
}
```

We can think of this as a "view" on a list of characters, with the proviso that a particular list of characters may fail to parse with some or all of the nonterminals. Let's augment the grammar with some semantic actions:

```
meta E {
  dig ::=  '0' | ... | '9';
  num ::=  <dig>+:ds           => number ds;
  fac ::=  <num>:x '*' <fac>:y => x * y
         | <num>;
  exp ::=  <fac>:x '+' <exp>:y => x + y
         | <fac>;
}
```

(I'm assuming a `number` function from a list of characters to a number.)

So when we ask for the `fac` "view" of the character stream, we're hoping to get a number --- and also, implicitly, the rest of the stream after the parsed-out factor. In Wadler's notation, borrowing from OCaml where necessary:

```
view fac ::= Fac int * char list
  in Num (x, '*' :: Fac(y, rest)) = Fac(x * y, rest)
  in Num (x, rest)                = Fac(x, rest)
```

(I've omitted the `out` clause because it's not relevant to the problem of parsing.) If the grammar were more OCaml-like, I think it would look like this. (Incomplete matches are considered bad style in OCaml, and the compiler warns about them.)

```
view fac = Fac of int * char list
  in Num (x, '*' :: Fac(y, rest)) -> Fac(x * y, rest)
   | Num (x, rest)                -> Fac(x, rest) ;;
```

The theory here is that if neither `in` clause matches, then the attempt to view a list of characters as a `fac` will fail to pattern-match,

which may cause backtracking of outer parses. In this syntax, it is only two and a half times as long as the rule in the OMeta-style grammar above.

The + notation causes a little bit of trouble; it's syntactic sugar. We can write out a desugared form as follows:

```
num ::= <digplus>:ds -> number ds;
digplus ::= <dig>:x <digplus>:y => x :: y
          | <dig>:x             => [x];
```

Here are the other nonterminals in this grammar, in the OCamly notation:

```
view dig = Dig of char * char list
  in c :: t when '0' <= c && c <= '9' -> Dig (c, t) ;;
view num = Num of int * char list
  in Digplus (ds, rest)               -> Num (number ds, rest) ;;
view digplus = Digplus of char list * char list
  in Dig (x, Digplus (y, rest))       -> Digplus (x :: y, rest)
   | Dig (x, rest)                    -> Digplus ([x], rest) ;;
view exp = Exp of int * char list
  in Fac(x, '+' :: Exp(y, rest))      -> Exp(x + y, rest)
   | Fac(x, rest)                     -> Exp(x, rest) ;;
```

And I think there you have the parser and expression evaluator, in a hypothetical extension of OCaml. Although this is several times wordier than the OMeta version above, it's several times more concise than the Haskell version in Ford's thesis. It's missing a bit of syntactic sugar, resulting in nested parentheses, unnecessary type definitions, redundant default cases, and explicit passing-around of the `rest` parameter, but it's much of the way there. ...

You might think that all of these nonterminals should be part of a single view `E`, but the normal case in OCaml and the other languages Wadler was thinking about that support pattern-matching is that the variants of a sum type are disjoint, which generally allows the compiler to tell you if you've missed a case in your case analysis. So here each one is a separate "type".

OMeta introduces four major extensions over normal PEGs:

- you're not limited to pattern-matching on streams of bytes or characters;
- there are ways to pass more arguments that weren't in the original input to nonterminals;
- guard clauses on patterns;
- you can subclass grammars.

Of course, if you have a functional programming language with pattern-matching, you have the first three of these four already.

Views in terms of OMeta

So if OMeta "contains views", can we write Wadler's other examples of uses of "views" in a reasonably homeomorphic way in OMeta?

PEG Parsing and Object-Oriented

Equational Rewrite Rules

My unimplemented object-oriented equational rewrite rules language supports views quite straightforwardly and concisely. ...

PEG parsing and Bicicleta

Bicicleta, although it doesn't have pattern-matching, does have "views" in the sense that accessing data that's stored inside a data structure looks the same as accessing data that's computed on the fly. This is also a feature of Self and Forth....

, and also contains a sort of notion of failure, which I believe can be implemented entirely outside of the core language itself --- it merely requires overloading the infix !! operator differently on non-error objects and error objects.

So I suspect that OMeta's simplistic notion of failure (it doesn't do general backtracking) should be fairly straightforward to implement in Bicicleta with an overloaded '/' operator, something like this:

```
parsed = {self:
    '/' = {op: '()' = self}
}
parse_failure = ...
```

References

Wadler's paper, "Views: a way for pattern matching to cohabit with data abstraction", was written in 1986 and published in 1987 in some ACM journal, pp.307--313.

Aardappel is an experimental visual programming language designed by Wouter van Oortmerssen, with a novel concurrency primitive and a very simple side-effect-free subset used to describe sequential computation.

Object-oriented equational rewrite rules is a kragen-tol post I wrote in 2007 about a slightly novel design for a pattern-matching programming language that natively incorporates views. I wrote further on it in 2008 in IRC bots with object-oriented equational rewrite rules, which is IRC bots with object-oriented equational rewrite rules (p. 838).

OMeta, by Alessandro Warth and Ian Piumarta, is a PEG parsing system generalized so that it can be used for a wider range of tasks. To date it has been used for scanners, parsers, optimizers, code generators, network protocol implementations, and automated code inspections, among other things. They published a paper about it in the Dynamic Language Symposium 2007.

PEG parsing is a new parsing method invented by Bryan Ford in 2002, based on work from the late 1960s on Bob McClure's TMG "TransMoGrifier" program. It can parse a large set of languages, including some non-context-free languages but probably not including all context-free languages, supports scannerless one-level parsing well, and has a provably linear-time parsing algorithm called "packrat parsing", which uses a lot of memory. It has its problems; the non-memory-hungry implementation can require exponential time, the absence of backtracking makes some languages more difficult to parse and presumably makes some languages impossible to parse, and the lack of support for left recursion in the basic algorithm

is annoying (although Warth and buddies have a solution to this problem that I haven't read yet.)

Topics

Berlinite gel

Kragen Javier Sitaker, 2019-12-14 (updated 2019-12-15) (10 minutes)

As I was adding materials data to Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) I happened across a material I'd never heard of before that seemed like a better fit for what I was looking for than any of the materials I *had* heard of: aluminum orthophosphate, the mineral berlinite.

I looked into its synthesis, and what I found was absolutely astounding; I must be missing something important, because it seems like a fairly revolutionary new material, but no revolution has resulted in the last 140 years since its discovery.

Berlinite

Berlinite is a rare quartz-like crystal of aluminum orthophosphate with a Mohs hardness of 6.5 and a melting point of 1800°. It's nontoxic enough to be used as an antacid, although there are several other phosphates of aluminum, which are caustic, and a highly toxic phosphide of aluminum, and sometimes these chemicals are confused with aluminum orthophosphate; every MSDS I can find states that aluminum orthophosphate causes severe skin burns, presumably as a result of this confusion; although I don't have any here to put on my skin, I think this is very unlikely to be correct. Its dihydrate is the uncommon turquoise-like mineral variscite, which has a Mohs hardness of only 4.5. It dunts like quartz, but at 583°. It wasn't discovered until 1868.

The really interesting thing about Berlinite is that it's almost as hard as quartz, but several investigators report easy ways to synthesize it at much lower temperatures.

Grover *et al.*'s low-temperature synthesis

In 1999 some researchers at Argonne National Lab and Purdue investigating the problem of safe nuclear waste storage published a paper on a berlinite-bonded alumina ceramic, entitled "Low-temperature synthesis of berlinite-bonded alumina ceramics". Their stunning claim was that they had synthesized berlinite *at 150°* by heating a hydrated aluminum phosphate they made by partly dissolving alumina in aqueous phosphoric acid at 130°.

In detail, they heated a slurry of alumina (micron-sized mixed with sand) with 50 wt.% phosphoric acid with an Al_2O_3:H_3PO_4 weight ratio of 5:1, detected an endothermic reaction at 118° with differential thermal analysis, held it at 130° for 1, 2, or 4 days to allow the reaction to complete, identified with X-ray diffraction that it produced an intermediate phase of another aluminum phosphate $AlH_3(PO_4)_2.H_2O$ mixed with the remaining alumina; they described the mixture as "a thick puttylike...gel", which they left at ambient temperature for another week "so that some crystalline growth would occur". They dried the samples into "hard monoliths", which "disintegrated when placed in water".

All of the above had been done by previous researchers. But then they baked the "monoliths" at 150° for one, two, and three days: "Significant porosity developed in the monoliths as bound water

escaped through an increasingly viscous slurry." They infer that the hot monoaluminum phosphate reacted with *more* of the remaining alumina to produce berlinite over the course of these days, and explain, "the resultant ceramic appeared to be a very hard monolith with dense phases separated by large pores," with some solid phosphoric acid on top; they measured a compressive strength of 6824 psi (47.05 MPa in modern units) on materials with "20.9 vol.% open porosity". This is a bit stronger than ordinary concrete but not remarkably so, and much lower than high-performance concrete.

Mysteriously, this paper has been almost entirely ignored. It's been cited in 2019 by US patent 10,233,803B2 on exhaust-gas filters with a catalyst film on a porous ceramic support (and its related 2017 applications in .de, .br, .uk, .cn, .kr, the WIPO PCT ("WO") and the EPO ("EP")); in 2006 by US patent 6,858,174B2 on gel-casting ceramic slurries; by Stefania CASSIANO GASPAR's dissertation in 2013 at INSA-Lyon on extruding porous berlinite-bonded alumina ceramic supports for catalyst films (in French) for exhaust-gas filtering; by her 2012 patent (US 9,227,873B2) with four other co-authors on the same subject; and by a 2015 Russian paper.

Cassiano Gaspar's dissertation also cites some 2010 work by Lee *et al.* with zeolites in aluminum phosphate, which she reports they got to be nonporous.

So, why do I think this is such an interesting result? It gives a simple, low-temperature recipe for a moldable porous material that costs about US$3/kg and produce a result as strong as ordinary concrete, which costs a tenth of that. What's so special about that? After all, if you want concrete with alumina abrasives in it, you can put some alumina in your concrete.

Well, a couple of things. First, this material is highly refractory, despite its low-temperature preparation; it will not spall from heat shocks. Second, I think it's nontoxic and noncaustic, unlike portland cement. It might even be biocompatible, which conventional phyllosilicate ceramics are not; alumina is well-known to be biocompatible, phosphate as such is nontoxic, and so the potential biocompatibility concern would mostly be with whether it releases aluminum into the body, catalyzes harmful reactions, or provokes inflammation.

In addition to its use for making massive objects, the resulting berlinite may work well as a binder for connecting aluminum parts together or for a mineral paint similar to those made from waterglass.

Apparently without citing Grover *et al.*, monoaluminum phosphate is sold as a castable refractory, sometimes using the heinous abbreviation "MALP" (to distinguish it from "MAP", monoaluminum phosphate). Fosbind is one brand. Luz, Oliveira, Gomes, and Pandolfelli reviewed its properties, calling it MAP, in a 2016 paper, using 200-micron dead-burnt magnesia and a sooper seekrit 31337 boron source to get it to set in 90–120 minutes, finding that they would work well up to 1400°–1500°; they cite a book (*Technology of Monolithic Refractories*, Nishikawa) and two papers on the subject in 1982–1989, and point out that it's easier to get aluminum phosphates if your aluminum source is aluminum hydroxide instead of alumina.

Interestingly, Luz *et al.* give a different formula for "monoaluminum phosphate": $Al(H_2PO_4)_3$, which is presumably

dramatically more acidic than the $AlH_3(PO_4)_2$ Grover *et al.* observed, since its phosphate groups still have their second hydrogen. Also Luz *et al.* experimented with binder systems including Al_2O_3/H_3PO_4 in 2015, and found that the most effective recipe was 48 wt.% phosphoric acid with the hydroxide. They claim that their X-ray diffraction results show that the aluminum orthophosphate was hydrated (i.e., variscite, not berlinite). For different compositions, they report *flexural* strengths in the 10 to 30 MPa range; presumably the compressive strength is much greater.

Silicon phosphate

It would be interesting if silica can somehow be phosphated the same way; normally the answer is no, and phosphoric-acid etching is a normal way to get silicon nitride off of silicon dioxide. But silicon orthophosphate, Si_3PO_4, does exist. Apparently it's caustic, which suggests that it isn't waterproof, which makes it immediately much less interesting.

Often the reason for wanting to use silica rather than alumina is that alumina costs 300 times as much as silica (see Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) for materials pricing). But in this case that reason is not nearly as overriding, because phosphoric acid is nearly as expensive as alumina.

Extrusion 3-D printing

The inexpensive 3-D printers that have become popular recently, descended from the RepRap project, are "fused deposition modeling" printers: they feed a filament into a melting chamber to produce pressure to force it out a small heated opening (the chamber and the opening are collectively the "hotend"), moving the opening and the workpiece relative to each other to deposit material in controlled positions on the object produced. Typically they use low-melting polylactide plastic at temperatures of 185° to 225°. Although higher temperatures are sometimes used, most organic chemicals start to break down into simpler substances in the 230° to 270° range. Other popular plastics include PETG, ABS, and nylon (PA6 I think, maybe 6,6.)

There are variants of this process that work well for clay bodies, clay slips, and adobe, largely relying on thixotropic or plastic yielding of the material with some kind of extrusion screw rather than melting. Instead of stressing the "hotend" with heat, these variants instead abrade it.

This approach should be applicable to the pasty aluminum phosphate Grover described, but it is somewhat complicated. If you use alumina as the aluminum source, the alumina particles are highly abrasive; only a very small number of coatings (diamond, CBN, and maybe zirconia, carborundum, and tungsten carbide) can resist being ground away by them, and frequent replacement of consumable liners and nozzles may be the only viable alternative. Also, the aluminum phosphates other than the orthophosphate are highly acidic and consequently corrosive.

Topics

A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins

Kragen Javier Sitaker, 2013-05-17 (updated 2013-05-20) (17 minutes)

(I think I published this on kragen-tol at some point.)

(Not yet tested.)

It occurred to me that you can probably make a resistive trackpad out of pencil lead on paper, sensing the position of a conductive probe in the conductive patch by measuring the resistance to various points around the edge many times per second. This should be doable with a few tri-stateable digital GPIO pins and a capacitor whose discharge time we measure, or a few analog input pins.

Perhaps you could even make this work with human fingers, even providing multitouch, even capacitively through a protective surface layer (say, a plastic shopping bag) to keep the graphite off your fingers, protect the microcontroller from static electricity, and maybe keep you from getting shocked by power surges.

It turns out that, in theory, the resistance in a uniform flat sheet is low and highly dependent on probe contact area when your probes are close together, but then approaches some limit, determined by the resistivity and thickness, as your probes get further and further apart. This might mean you need to have quite a number of probes.

Normal trackpads work differently

Normal resistive trackpads have a grid of wires, with the X and Y wires running in two separate layers and a resistive layer between them, which becomes less resistive under pressure. Normal capacitive trackpads have a grid of wires, too, but use capacitive coupling to your finger instead of sensing pressure.

This trackpad design doesn't have a grid of wires at all, just a resistive plane, which is much easier to fabricate.

Resistance tests with pencil lead

In my first test, I blackened a patch on a post-it note (Stick'N brand) rather thoroughly with an HB pencil, moving the post-it around a bit to blacken in different directions to avoid voids in the graphite caused by roughness of the desk surface, and measured with an ohmmeter resistances of some 2–5kΩ over distances of 1–4 cm, and as low as 1kΩ with probes close together (≈1mm).

(As predicted by theory, the resistance depends strongly on probe contact area. Using a couple of big coins, I was able to get measurements down to 300Ω, and the limit at larger distances was about 2kΩ instead of 5kΩ.)

In a second test, instead, I drew a grid which was about 90% empty space and 10% fairly dark lines with the same pencil. With this, I was able to get readings of a few hundred kΩ on my ohmmeter over distances of a centimeter or so, but they weren't very consistent.

In a third test, I gradually darkened a patch with the pencil until I started to get consistent continuity between different parts of the

patch, at which point I was getting readings of 200–500kΩ over 1–4 cm, although I don't totally trust my analog ohmmeter at such large resistances. Even with the probes separated by only 1mm or so, I was still getting 100kΩ or so. This patch was much less dark than the other one, but the other one certainly did not contain 100 times as much graphite; so I assume that the majority of the graphite particles in this one are not participating in the conductivity.

I hypothesize that you might be able to reliably get an even higher sheet resistance this way, but it might help if you have really smooth paper to start with — glossy magazine or inkjet-photo stock, say.

Measuring distance to the capacitive patch where your finger is

If you press your finger against a layer of polyethylene on top of the pencil lead, your finger and the pencil lead form a capacitor. (I thought about using paper instead of polyethylene, but it's six times as thick.)

Your finger contact area might be 1.5cm², and a plastic shopping bag might be 13 μm thick, according to Multi-Pak's "Thick and Thin of Plastic Bags". $C = \varepsilon A/d$, and the permittivity ε of polyethylene is 2.25 times that of free space ε_0, which is about 8.9 pF/m. So we have 8.9 pF/m * 2.25 * 1.5cm² / 13μm ≈ 220pF.

Now, we'd like the circuit capacitive reactance to be close to the resistance of the pencil lead in the distance from the point of contact to the place where the measurement terminals are attached. If the resistance is much greater than the capacitive reactance, the signal will be attenuated unnecessarily, making the circuit too susceptible to noise; if it's much smaller, the signal will reach all of the measurement terminals almost equally, and you won't be able to tell where on the paper the touch happened. Let's say the pencil lead resistance is in the 1–30kΩ range, or 10kΩ, to be concrete.

It turns out that people have about 1pF of capacitance to power lines, and also about 100pF of capacitance to ground. If we use that capacitance to ground, then we can use an AC voltage we send into the trackpad ourselves, at a frequency chosen to optimize the capacitive reactance, and most of the circuit reactance will be from the 100pF reactance to ground.

(Note that this depends on your skin. http://dev.laptop.org/ticket/8071 reports that people with calloused hands had trouble with the early OLPC XO's capacitive trackpad because the dry callous put people's blood too far from the trackpad. If we're depending on a dielectric thickness of 13 microns, we'll have the same problem. I guess you can lick your fingers though.)

So if we want 10kΩ of capacitive reactance out of 100pF, $X_C = 1/(2\pi fC)$, so we have $f = 1/(10k\Omega * 2\pi\ 100pF) \approx 160kHz$. It's trivial to generate a 160kHz square wave on pretty much any microcontroller, and if your VCC is 5V, it's pretty much 5V peak-to-peak and 2.5V RMS. That should be easy as pie to sense: with 14kΩ impedance, it's 180 microamps, -3dBm.

To look at it in the time domain instead of the frequency domain, we're charging the patch of graphite under your finger up to 5 volts or whatever VCC is, through the resistance of the graphite between that patch and our probe. The time it takes to charge it up will tell us the

RC product of that system, and we expect it to be on the order of 6µs, which is a time that we can measure to about 6 bits of precision without even using an ADC. If we do use an ADC, like the 10-bit ADC on a lot of AVRs, we should be able to estimate the RC time constant of the charging curve to higher precision.

The RC constant by itself tells us very little, because we don't know the area of the finger contact patch, and that's a factor of C. But if we have several different probes in different locations, we can probe with each of them in sequence; we can assume the finger contact patch isn't changing much in 6µs. This tells us something about the relative distances, from which we can estimate the position of the contact patch; and given that, we can estimate its area.

It's common to have fewer pins capable of ADC input than capable of digital output. But I think that doesn't need to be a limitation for this application; you can use one probe to measure the voltage the contact patch is charged to, while using another one to apply the charging pulse. As long as the input impedance of the ADC pin is high compared to the impedance of the contact patch's path to ground, the ADC probe voltage should follow the contact patch's voltage very closely — unless it's too close to the active probe, in which case it will see some kind of weighted average of the active probe's voltage and the contact patch's voltage. The ATMega328 datasheet (doc8161.pdf) says the input pin capacitance is a max of 10pF (p.321), so the ADC input capacitance shouldn't be a big problem. It should be possible to separate out the weighted-average components, since they'll have two very different time constants.

So I think you should be able to get by with three or four ADC pins at different corners of the touch area, with active probes in potentially more places; at least two active probes and, I think, three total probes will be needed to triangulate unambiguously. So you need at least three ADC pins and two GPIO pins, but the ADC pins can double as GPIO pins.

6µs might be *too* fast; while the ATMega328 can do a sample-and-hold more quickly than this, it needs much more time to do a complete ADC conversion. So you might end up having to build up the picture of the complete charging waveform by sampling one point on each of a number of charging waveforms, by adjusting the "phase" delay between the initiation of a charging pulse and the ADC measurement.

Alternatively, as described in the section above about my experiments, you might be able to simply establish a thinner film of pencil lead to further increase the R part of the RC time constant. 60µs should be plenty of time.

You probably need at least four data points on each curve to untangle the effects of R, C, and the effect of resistively-coupled voltage from the active probe. You're measuring, minimum, six separate curves, so you need at least 24 data points. If you need 13 µs per data point, at which point the ATMega328 datasheet says the error is 4.5 ulps (LSB), which I guess is ±, so you really only have about 7.5 bits of accuracy, then you need 312 µs to take a full position and patch size measurement. You can probably average your measurements over considerably more than that, which means you can measure many more points per curve.

If your final measurements are a result of measuring each curve 64

times instead of 4, giving you 5 ms latency, your estimates for R should be good to about 11 or 12 bits, and your estimates for C (since you're measuring it six times as often) should be good to about 14 bits. All in all, this gives you 6*11.5 + 14 = 83 bits of data, which is probably highly redundant, but which can hopefully localize your finger with great precision.

Resistive-touchscreen style

Wikipedia explains analogue resistive touchscreens as follows:

> two flexible sheets coated with a resistive material and separated by an air gap or microdots.... during operation of a four-wire touchscreen, a uniform, unidirectional voltage gradient is applied to the first sheet. When the two sheets are pressed together, the second sheet measures the voltage as distance along the first sheet, providing the X coordinate. When this contact coordinate has been acquired, the voltage gradient is applied to the second sheet to ascertain the Y coordinate.

This also seems like it might be a reasonable approach to use with two sheets of paper. You could dump eraser crumbs or a few curly hairs or something in between them to keep them mostly apart when they're not being pressed.

Alternative materials

Pencil lead is one useful and easily available resistive material that can be easily deposited in a film, for example on paper; actual lead, aluminum, gold, and carbon black are four others. Lead is about 50 times more conductive than pure graphite, and pencil lead is only part graphite. Aluminum and gold are another 10 times more conductive still.

Aluminum has the potential advantage for this application that it's widely available in the garbage in large, highly uniform sheets sputtered onto Mylar, which are then coated with an insulating layer of plastic even finer than a plastic grocery bag. These layers of aluminum are so thin that they're actually visibly transparent, when there isn't too much ink on top of them. If you put LEDs behind them, you could make an actual touchscreen.

However, I have so far been completely unsuccessful at making electrical connections to these aluminum layers. Aluminum is notoriously tricky to connect to electrically at the best of times, due to sapphire spontaneously forming on its surface, and much more so when it's in the form of a sub-micron-thickness film inside a much tougher plastic film.

Aluminum rubbed onto the surface of paper, like pencil lead or lead, might be more promising, especially if you drip solder or something on top of it immediately afterwards.

Charcoal is, in theory, somewhat conductive, and you can rub it on paper, but in my feeble attempts with a burned match, I haven't been able to get it to conduct. It's easier to come by in garbage than lead, graphite, gold, or bulk aluminum, so it might be worth further investigation.

Gold is expensive, but gold leaf is so thin that it's cheap enough that people throw it away, or even drink it in Goldschlager. I don't have any handy here, but it might work for this.

Carbon black is conductive, and indeed is used in the microphone design that made the telephone practical, since its resistance varies

with pressure. It's also easy to deposit in a film on a surface: you burn something, anything carbon-bearing, with a flame, and the smoke deposits as a film. (If the flame is too clean-burning, you may have to put an object actually inside the flame to get the carbon black to deposit. You can do this with a cigarette lighter.) I think you can get a more uniform film this way than by rubbing solid objects onto abrasive paper, but the film is very fragile, except where it's nestled down into a pore in a porous surface.

However, its very fragility means that you can very easily cut very fine lines into it: if you deposit it onto glass or plastic, you can rub it off with your finger, or the end of a wire, or a piece of dry grass. If you had a way to deposit some kind of insulating fixative on top of it (superglue? boiled linseed oil? dried eggwhite? glue down a shopping bag?) and deposit more carbon black on top of that, maybe you could make the grid of wires used in traditional capacitive trackpads.

Paper, as I mentioned before, is another possible dielectric; its big disadvantage is that it's typically several times thicker than a shopping bag. However, it might have advantages, too. You can write on it more easily.

It would be cool to have a transparent dielectric that can be illuminated from the edge, because then you could have like glowing letters and shit etched into its surface illuminated by total internal reflection, like the menu for every third tourist bar in this damn town. Unfortunately, I don't think you can do that in a sheet of dielectric that's thin enough to leave a reasonable amount of capacitance between your finger and the graphite. If it's 100 microns thick, the thickness of paper, the capacitance drops to 30 pF; if it's 200 microns thick, it drops to 15 pF. It might be hard to sense a mere 15 pF on top of the 7–10 pF on every AVR input pin and the ≈14pF of the sample-and-hold capacitor.

If you find an unbroken LCD screen in the garbage, you ought to be able to use the transparent indium tin oxide ("ITO") electrodes deposited on the glass surface for a high-resolution grid of wires. But it might be cooler to use it as an LCD.

Topics

Texture synthesis with spatial-domain particle filters

Kragen Javier Sitaker, 2016-10-06 (2 minutes)

Can you synthesize textures using particle filters?

The idea is that for each position (block of pixels, or maybe pixel), you have a model of what might appear there, based on what appears above it and to its left, determined by the conditional probabilities of that patch in those environments in some kind of a training set. You maintain a potentially large set of hypotheses for each position, each one with a weight representing its probability weight. When you move to a new position, you resample the set of hypotheses according to the weights, and ultimately you may end up with a few hypotheses with most of the probability weight and you can choose one or average them.

An interesting thing about this is that it allows you to take into account different kinds of information in many different ways. For example, you could know most of the pixels in an image and want to fill in a few (perhaps to replace something you erased or to fill in a seam), or you could attempt to synthesize an image from nothing, or you could try to synthesize an image that approximates some pattern of light and dark, or that has edges in particular places.

This seems to have been done to some extent; http://link.springer.com/article/10.1186/2193-9772-2-2 ("High resolution micrograph synthesis using a parametric texture model and a particle filter", 2013) describes using this algorithm to fill in plausible high-resolution detail on a large low-resolution image by using texture from small high-resolution images. But I think this may be a slightly different algorithm.

Topics

Audio video boustrophedon sync

Kragen Javier Sitaker, 2019-04-03 (2 minutes)

If you're trying to sync audio with video, for example because you recorded them on different devices, you acquired them via different sources (that don't share a timebase), or you're playing them back through devices with different latencies, you traditionally need some kind of interactive successive approximation where you twist some kind of knob or something until you get it right.

A difficulty with this is that only occasionally are there events in the video that you can associate with high precision with the audio. So you have to wait for one of them to happen once for each approximation.

The ideal solution is probably to find a short segment of video with such an event and repeat it at as high a frequency as possible as you adjust the audio–video lag. Around 20 Hz, rhythms — pulse trains — fade into continuous tones. So you need something a little slower than that, which possibly limits you to about 50 ms feedback latency on the knob. Also, you probably need several frames of video to successfully interpolate movement, and video is commonly interpolated at rates as low as 24 Hz, so you might not be able to do better than, say, 6 Hz or 4 Hz for the repetition.

However, at these rates, the impulsive event you're looking to synchronize the sound with — ideally a clapboard, but often in practice a spoken plosive or something — may be overwhelmed by the much larger impulsive event of jumping back in time by 167 ms or 250 ms. A possible solution is to play both the video and the audio segment boustrophedonically: first forward, then backward, then forward again, and so on. This eliminates the first-order discontinuities, though the remaining second-order discontinuities — objects instantly reversing their direction of motion — may be disturbing.

The knob should probably directly control the video displacement, not the audio displacement, to avoid screwing with the pitch.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Audio (p. 3331) (40 notes)
- Video (p. 3768) (7 notes)

Clickable terminal patterns

Kragen Javier Sitaker, 2013-05-17 (2 minutes)

libvte9 includes the ability to tell its clients if a particular position is within a match to any of a list of regexps; this ability is used by gnome-terminal and xfce4-terminal to make links in terminals clickable.

However, the list of regexps is hardcoded. It would be useful to add additional regexps, for reasons like the following:

• Clicking on `grep -n` output like "./terminal/terminal-widget.c:292:" could request Emacs to open the file at the line in question.
• Clicking on a company-internal bug number like "ITN-6748" could open the URL of the ticket in question.
• Clicking on a date could open your favorite calendar app to that date.
• Clicking on a Bitcoin address could open your Bitcoin client to send Bitcoins to its owner.
• Clicking on an IP address could prompt you with various things to do: mtr, nmap, reverse DNS lookup, etc.
• Clicking on a street address could search for it in Google Maps.
• Clicking on a Twitter @username could open their Twitter page.
• Clicking on a DOI number could look it up in a way appropriate to your campus.
• Clicking on a phone number could call it, add it to your address book, or look it up in your address book.

These suggest a configuration file with a syntax like

```
/home/paul/bin/open-ticket ITN-[[:digit:]]+
/home/paul/bin/open-editor ^[./][^ ]*:[[:digit:]]+:
/home/paul/bin/open-calendar \<[[:digit:]]{4}-[[:digit:]]{2}-[[:digit:]]{2}\>
```

where the scripts alluded to will be invoked with the matched string as an argument. `open-editor` could be implemented, for example, as follows:

```
#!/bin/sh
exec >"$HOME/editor.log" 2>&1
IFS=:
set $1
emacsclient +$2 $1 &
```

In the case of xfce-terminal, this would involve:

• Dynamically allocating new pattern types when reading this file, in addition to the four predefined ones at terminal/terminal-widget.c:65.
• Adding the new regexes to the libvte widget, along with the ones from `regex_patterns` at terminal/terminal-widget.c:78, in `terminal_widget_update_highlight_urls` at terminal/terminal-widget.c:772.
• Perhaps factoring the two calls to `vte_terminal_match_check` in `terminal_widget_context_menu` at terminal/terminal-widget.c:292 and in `terminal_widget_button_press_event` at terminal/terminal-widget.c:403 into

a single thing? In any case, wire them both up to invoke a new function to invoke the user-defined action.
• Writing the new user-defined action function, which would spawn off a subprocess with the requested command.

Topics

• Programming (p. 3658) (286 notes)
• Human–computer interaction (p. 3493) (76 notes)

Non-inverting logic

Kragen Javier Sitaker, 2017-02-18 (updated 2019-07-20) (8 minutes)

There were common non-inverting logic families in the past, an idea that always appealed to me in my childhood, since burning power in every logic gate seemed wasteful. (I didn't understand how much it simplified the process of digital design.) Diode logic and threshold logic (McCulloch-Pitts neurons without inhibitory inputs) are two rough families, although they are at different levels of abstraction.

Circuits whose combinational logic is non-inverting

A typical way to manage this, e.g. in Stan Frankel's 1956 LGP-30, was to do the combinational logic with diode logic, then the registers with flip-flops which generated two-rail outputs, i.e. both Q and Q̄. A flip-flop at the time was a pair of cross-coupled vacuum tubes and a pair of resistors, as I understand it. (The apparent design rationale for the 1958 Сетунь, wanting to use a smaller number of flip-flop-flaps rather than a larger number of flip-flops, makes me think this kind of thing was universal at the time.)

The LGP-30-style design combines the functions of amplification, signal level restoration, inversion, glitch elimination, and memory in a single device, the flip-flop.

In case it's not clear how this works: absent restrictions on fan-outs and fan-ins, you can always reduce the combinational logic to this form as follows. Register all the inputs through flip-flops; convert the desired logical formula into sum-of-products form (disjunctive normal form), i.e. $\vee_i \wedge_j X_{ij}$ where each X_{ij} is either 1, an input variable Q_k, or a negated input variable $\bar{Q}_k$; run all the flip-flop outputs, both negated and non-negated, down the columns of your schematic; convert each $\wedge_j X_{ij}$ into a row that is a many-input AND gate taking its inputs from those columns; and finally combine all the rows with a many-input OR gate. You can think of this as a kind of binary matrix-multiplication operation, where the bits of the resulting vector are the inputs of the final OR gate, which computes the Boolean norm† of that vector.

Note that both AND and OR gates are non-inverting, and here you only need two levels of them, so this is a feasible design to realize in unamplified diode logic. It may be possible to do better, since in many cases the inflation factor of reducing a formula to a sum of products is large, but it guarantees that it is always possible.

This kind of thing seems appealing to me somehow even in the world of MOSFETs: an N-channel enhancement-mode‡ MOSFET with a weak pulldown resistor can be thought of as an AND gate where a low-impedance connection to Vdd counts as "1", no such low-impedance connection counts as "0", and no low-impedance paths to Vss exist. With these conventions, an OR is just a wired connection, and the circuit depth problems imposed by diode voltage drops go away. Presumably people have thought of this, but this approach isn't widely used, maybe because the pulldowns are slow to

discharge the gates unless they're burning a lot of power normally. (I thought this might be what is called "pass transistor logic", but that turns out to be a different but related approach.)

Circuits whose combinational and also sequential logic is non-inverting

It wasn't until I was rereading Merkle's 1990 buckling-spring paper "Two types of mechanical reversible logic" today that I understood that the sum-of-products thing generalizes to cases where your memory elements don't do inversion. He makes a casual aside about how Landauer in previous work had assumed dual-rail logic, where each value computed is accompanied by its complement, so negation is obtained simply by switching the two.

The Landauer reference is found in his "Dissipation and noise immunity in computation and communication", on p.781 of Nature, Vol. 335:

> In these schemes we use majority logic to do computation. There is an odd number of inputs (typically three) to a given [potential energy] well. The well under consideration then gets set according to the majority vote of the stages exerting an influence on it. If, for example, we have one input set at zero, with the other two inputs being variable, we get a '1' output [iff] both of the variable inputs are '1'. In this scheme we cannot perform a negation directly. Therefore, we need dual rail logic, in which we carry along a variable and its complement, and perform negation by interchanging the two.

This is a brilliant revelation to me: it means that you don't need negation *inside* the system *at all*, even in memory; you only need to generate the complements of signals *on input* to the system. That is, the feature of flip-flops that they automatically generate complements on every registered bit is completely dispensable, at the cost of twice as much logic, more or less.

This is achieved through simple De Morgan conversion: to compute ~(XY), you compute ~X | ~Y, and to compute ~(X | Y), you merely compute ~X & ~Y, where the operators are as in C.

Consider a basic inverting function which is necessary for binary addition, XOR: $X \oplus Y \equiv X\bar{Y} + \bar{X}Y$. In this dual-rail scheme, you can compute it in just that way, and then, if necessary, its complement, XNOR, as $(\bar{X} + Y)(X + \bar{Y})$. This requires six AND and OR gates, but calculating it with NAND requires four gates, and calculating the complement requires five (with, e.g., ((B|B)|(A|A))|(B|A), where | is now the Sheffer stroke denoting NAND rather than the C bitwise OR.)

In a dual-rail system with absolutely no inversion inside the system — only inversion at inputs — you may need more register bits than you would need in a system in which only the combinational logic is noninverting, since you (often) have to register both a bit and its complement, while a conventional flip-flop will generate one from the other.

Further extensions

The dual-rail system can be thought of as a one-hot encoding with two possible values. It is straightforward to mix dual-rail signals with one-hot signals of other multiplicities, including the Сетунь's three but also four and even five and higher, as in the biquinary encoding used in the IBM 650 Knuth learned to program on; this may afford

economies of logic under many circumstances, as the aforementioned Boolean matrix becomes much sparser.

For e

Note that Landauer's suggested family of approaches performs level restoration at every gate, rather than leaving that up to the flip-flops, and thus doesn't suffer the severe limitations on combinational circuit depth that diode logic did. Circuit depth is still a crucial factor for clock speeds, though. (And the escape from this, asynchronous logic, coincidentally also typically depends on dual-rail logic...)

† I say "the Boolean norm of that vector" because I think that N-way bitwise OR is the only non-constant function that satisfies the definition of a norm $((g \neq 0 \Rightarrow \nu(g) > 0) \wedge \nu(g + h) \leq \nu(g) + \nu(h) \wedge (g \in \mathbb{Z} \Rightarrow \nu(mg) = |m|\nu(g)))$, under the usual interpretations: AND for multiplication, OR for addition, 0 for false, 1 for true, and $0 < 1$.

‡ You could use a P-channel enhancement-mode MOSFET as an AND-NOT (abjunction or negated implication) gate, which provides a limited form of negation: it will only let through a low-impedance connection to the positive power rail if the gate is pulled low, meaning that the gate has no such connection. Abjunction is as universal as NAND if you have access to a constant 1, but it's falsehood-preserving: any circuit made of abjunction gates will have output 0 if all its inputs are 0. Worse (for use as a NAND-like universal gate, anyway, though not in this context) is the fact that you can't build an OR gate out of abjunctions without access to that constant 1, even though the OR gate is also falsehood-preserving.

Topics

Hot air ice shaping

Kragen Javier Sitaker, 2016-10-06 (4 minutes)

You can shape ice with streams of hot air for rapid, inexpensive fabrication.

Water is one of the cheapest materials available, and it freezes and melts at a temperature that's extremely convenient for shaping, although it has its drawbacks for structural use. But it should be possible to use a form made of water ice to give shape to a mold made of some other material that can harden below 0°, or to make an easily-removable mold for such a material.

Ice has the advantage that, because it's used over such a narrow temperature range, it has excellent dimensional stability.

Ice is fairly fragile (about 1 MPa tensile strength, which depends greatly on strain rate and very little on temperature), so precisely stamping things with it is pretty much out of the question; to transfer its shape to something else, that other thing pretty much needs to be liquid. By the same token, though, it's very easy to cut, leading to the ice sculpture competitions in various places every winter, as well as table centerpieces at many corporate events. Other possible ways to shape ice include additive manufacturing (by adding water just above freezing to a below-freezing workpiece) and carving it with high-speed streams of hot air.

For additive manufacturing, it probably would help a lot to include some impurities in the water that increase viscosity and perhaps make the water thixotropic, so that it stays put on the workpiece from the time it comes out of the nozzle until the time it freezes, but without lowering its freezing point too much or promoting large ice crystals. Possibilities that occur to me include tiny air bubbles, gelatin, alginate, agar-agar, carrageenan, konnyaku, gum arabic, xanthan gum, and grease (oil with surfactant), possibly with acids or bases to promote gel formation.

There are a few different hardening processes that could harden the secondary material.

First, obviously, there's freezing, where the liquid cools down below its freezing point and becomes a solid. This has the disadvantage that it releases a large amount of heat, which can melt the surface of the ice, although this is a less serious problem with low-crystallinity materials such as Dairy Queen dip-cone chocolate-flavored paraffin wax. And maybe, since it will happen first at the surface of contact, it won't cause any loss of detail. Hot-beeswax casting has long been used to "record the shapes of delicate ice accretions on aircraft components", according to Reehorst and Richter in 1987.

Second, there's the hydration-driven recrystallization of plaster of Paris. This happens more slowly at 0°, but I'm pretty sure it still happens. It also has the disadvantage that it releases a lot of heat.

Third, there's polymerization, which is what Reehorst and Richter reported good results with, starting with thinned silicone rubber resin at -5° and recooled after degassing. Many silicones will not cure at these low temperatures; Reehorst and Richter found that Dow Corning 3110 or 3112 with "catalyst 4" worked well, and Dow

Corning HS RTV worked best (10 parts base to 1 catalyst with 1 wt% 20-cSt-viscosity DC 200 silicone thinner). Smooth-On Corporation sells a silicone molding compound recommended by Freeze Cast Engineering for this purpose.

Topics

Some notes on FullPliant and Pliant

Kragen Javier Sitaker, 2018-04-27 (9 minutes)

So I've been reading through the FullPliant website.

Summary

Pliant is a programming language that represents an exploration of a promising but underexplored corner of the language design space. It has static typing with a frequently used escape hatch to ad-hoc polymorphism, like Java, but with minimal, extensible syntax, like Tcl, Forth, or to a lesser extent, Ruby, Scheme, or Common Lisp. It uses ref counting.

In more detail: the author, Hubert Tonneau, clearly has an unreasonably high opinion of himself, but the system he's built seems reasonably efficient; it's not quite the full operating system he makes it out to be, but it's pretty close. He's built a text editor; an IDE; a compiler; a graphical user interface all the way down to raw pixels; a database; a web page framework; servers for SMTP, HTTP, and VNC RFB; a bandwidth-optimized GUI remoting protocol; a compiler; and a sampling profiler. The programming language is syntactically close to being a Lisp with some syntax sugar and macros, but it's explicitly and statically typed, allowing a relatively simple compiler to get good performance. Memory management uses reference counting or, optionally, explicit malloc and free. The database is an IMS-like hierarchical database, with a modern disk-as-tape-for-the-transaction-log design. The system is poorly documented, but it appears that the language has both first-class functions and dynamic method dispatch.

The main compiler is written in Pliant, bootstrapped from a subset written in C.

This seems like a pretty reasonable overall design for the stuff he's trying to do, and it seems to be reasonably successful. The signal-to-noise ratio of the example code below is very high.

I wish there were documentation in French, as Tonneau's English is very poor, and although the existing documentation comes out to book length (I've been reading it for days), it's quite sketchy at best.

Tonneau is only casually familiar with the literature and related work in the field, so he has reinvented a number of things.

Syntax and Embedded DSLs in Pliant

One of the more interesting features of Pliant is how he uses the macro system and syntactic sugar to get Ruby-like DSL power for things like the GUI; here's a snippet of an example from the manual:

```
var Float total := 0
each a o:article
  row
    cell
      input "" a:ref
    hook
```

```
    cell
      input "" a:ppu
    cell
      input "" a:count
  change
    section_replay "content"
  cell
    text (String a:count*a:ppu)
    total += a:count*a.ppu
  cell
    button "remove"
      o:article delete keyof:a
      section_replay "content"
```

Pliant's syntax is extensible, so this code can take advantage of Ruby-like blocks passed to macros. It builds a table with the price-per-unit and count cells wrapped in an on-change hook. Even for ordinary functions rather than macros, the argument semantics permit by-reference in-out parameters, so the `input` control can be hooked directly to the database record field with a minimum of fuss. `section_replay` re-executes the code block to draw the named "section" of the UI, which, as it happens, includes the code snippet above. That implies closure semantics; see below.

The `each` macro is not specific to the database module, but in this case is iterating over a database query result; as you can imagine, it declares the variable `a` provided as its first argument, and iterates over the subordinate records in `o:article`.

The whole infix and indentation syntax thing is just syntax sugar over S-expressions; part of the code above is, I think, syntactic sugar for the following S-expression in conventional notation:

```
(cell ({} (text (String (* (a count) (a ppu))))
          (+= total (* (a count) (a ppu)))))
```

The syntax is perhaps unfortunately so minimalistic that I haven't yet figured out how to tell when a bareword will be evaluated as a variable and when it will be treated as a symbol, even in contexts where the head of the clause is clearly a variable and not a macro name. Here, for example, `total` is a variable, but `count` and `ppu` are names of methods to call. I don't know if defining methods creates a variable in the environment that refers to the corresponding generic function or what.

Pliant macros ("meta functions") are not source-to-source transformations; rather, they emit the compiler's intermediate representation for their part of the AST, possibly after incorporating the IR from their child nodes. Tonneau argues that this makes them more orthogonally composable than Lisp macros, but the complexity cost in the implementation of each macro is high, and I am not convinced of his thesis as its benefits.

As an example, Tonneau explains the `thread` macro in the manual. To implement the closure semantic required (similar to those of `button`) it invokes the compiler function `uses` on the emitted IR from the child nodes to determine which variables must be closed over, then explicitly creates code to copy the variable contents into the child

thread. I have no idea how that interacts with the optional reference counting machinery.

Pointer semantics in Pliant

Reference counting in Pliant depends on the use of smart-pointer classes like Arrow and Link, which are instantiated with the type of their targets; I think this makes them parametrically polymorphic, but because the type system is never explained anywhere, I'm not sure.

The usual kinds of pointers in Pliant are implicitly dereferenced; they are transparent references to the underlying variable, and support the same operations as a variable of the underlying type, including `:=`, plus a separate `:>` operation which reseats it to point to a different value. They can refer to variables inside arrays, dictionaries, and other containers, and they can even be used as iterators in something like the C++ style, but address arithmetic is not normal. An example from the manual of iterating over the elements of a dictionary with key elements of some type xxx mapped to value elements of some type yyy:

```
var (Dictionary xxx yyy) d
var Pointer:yyy p
p :> d first
while exists:p
    foo p
    p :> d next p
```

This is equivalent to this use of `each`:

```
var (Dictionary xxx yyy) d
each p d
    foo p
```

There's a separate `Address` type that does support address arithmetic, requires an explicit dereferencing operation, and carries no target type.

Implicit dereferencing clearly poses problems for manipulating pointer chains; there's a `:>>` operation to reseat the second (or possibly ultimate) pointer in such a chain.

I speculate that the implicit nature of Pliant dereferencing is helpful to making embedded DSLs.

Runtime error handling in Pliant

Pliant has a little-used exception system. By default, it indicates errors through abrupt termination of the Pliant process; file I/O operations also have a `safe` option to instead use function return values. There is no apparent support for Golang-like multiple return values with compiler warnings on unused variables, nor for Haskell-like variant types with pattern matching.

When an exception is caught, the exception system does not immediately transfer control to the error handler; it appears to continue execution normally with a pending error stored somewhere, indicated by the `iserror` boolean function.

There is syntactic sugar for propagating exceptions; an equivalent

to the

```
foo bar
if iserror
    return
```

is

```
foo bar ?
```

Abrupt process termination provides a source-level stack trace.

I said the exception system was little-used, but it is used consistently in some places. In particular the `compile` method on AST nodes normally indicates compile errors using exceptions.

Reference Counting and Dynamic Dispatch in Pliant

Any Pliant variable can be an "object", with a header containing a reference count (which I suppose implies it must be heap-allocated) and a type pointer, or not. Various parts of the Pliant machinery assume that they are operating on objects, and this is sometimes verified at compile time. The `Link` smart pointer type has the semantics of `Pointer`, plus increments and decrements reference counts, and the `Arrow` smart pointer type is the analogous reference-counted version of `Address`.

Reference counting in Pliant is apparently thread-safe. I don't know whether this makes it a bottleneck in heavily multithreaded code.

I don't understand the semantics of method calls and dynamic dispatch much at all.

The type pointer can be accessed with the `entry_type` function for dynamic type tests.

The Pliant database

The Pliant database is made of typed records arranged in a tree; some of the fields can be multivalued, and their values have another record type, as in older hierarchical databases like IMS. I haven't found anything permitting polymorphic record types or indices.

Tonneau admits that the database's hierarchical design is less flexible than an RDBMS, but considers it adequate. His idea of RDBMSes is pretty hazy; he thinks they autonomously decide which indices to create.

I don't know what the database's transaction semantics are.

Topics

(2015-05-29)

It should be possible to "perform in-memory computations on large, even decentralized clusters in a fault-tolerant manner", as Apache Spark does, using Vesta-like build-step isolation, but with the shell usability of `redo`, using a `git-annex`-like blob backend, thus expanding the applicability of Spark-like computational structures far beyond the data center.

This requires substantial explanation.

Spark

Apache Spark is a system for making high-performance, fault-tolerant clusters easy to use, by generating and managing things Spark calls RDDs:

> [A]n RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs.…an RDD has enough information about how it was derived from other datasets (its *lineage*) to *compute* its partitions from data in stable storage.… in essence, a program cannot reference an RDD that it cannot reconstruct after a failure.

This sounds terribly similar to `make` and software source control and build systems, doesn't it?

redo

In 2010, Avery Pennarun implemented Dan Bernstein's `redo` design, which is a simpler approach to what `make` does. If you want to build a file named `foo.o`, you run `redo foo.o`, and `redo` will run the shell script `foo.o.do` if it exists, or otherwise `default.o.do` if it exists, or otherwise `default.do`, searching up the filesystem hierarchy. This `.do` file is expected to put the desired contents of `foo.o` into an output file whose name is passed to it.

A minimal useful `default.o.do` might say

```
redo-ifchange "$2".c; cc "$2".c -o "$3"
```

The first command there is a recursive invocation of `redo`, which tells `redo` that this output file is going to depend on `foo.c`, so it had better make sure `foo.c` is up-to-date before continuing. This dependency is recorded for later. (`redo-ifchange` takes multiple arguments to build dependencies in parallel.)

A more complete `default.o.do` would take `#include` dependencies into account.

`redo` does have a couple of limitations. One is that it doesn't handle multiple output files, like those from `yacc`, very well. Another is that it is purely local; it doesn't have a cluster mode, although you may be able to use `distcc` to get some of that yumminess.

ccache

ccache is a build accelerator specific to C, C++, and Objective-C. It hashes your source code, include files, compiler (well, typically just its size, mtime, and name), command-line options, etc., with MD4, and stores the compiler's output (including e.g. warning messages) in an on-disk cache in, normally, your home directory; future recompilation attempts whose inputs haven't changed will just reuse the previous compiler output.

In theory, this means that you could get most of the benefits of `make` for C and especially C++ programs by just wrapping your compiler in `ccache` inside your build script; rerunning the build script would rehash all your source files, copy the object files into your current directory, and then relink the executable. Depending on what you're compiling, this might be almost instantaneous, or it might be very slow.

Because Unix provides `ccache` with no reliable way to get a secure hash of the source files from the filesystem, it has to read them in their entirety to figure out whether they have reverted to an old version. I tried it just now on my netbook on a tiny three-kilobyte GLUT program, and it ended up reading about a megabyte of `.h` files in order to figure out that it could safely reuse the 2.7kB `.o` file from a previous compilation, taking 39 milliseconds in all, even though it made only about 365 system calls.

Database indexes

Suppose I do this:

```
sqlite> create table foo (x varchar, y varchar);
sqlite> create index foox on foo (x);
sqlite> insert into foo values (3, 4);
sqlite> insert into foo values (4, 5);
```

`foox` is the name of an index, which is a sorted copy of a column `x` of table `foo` with rowids.

Now, if I do a query on `foo` like `select * from foo where x = 3;`, this query can use the index `foox` to find the relevant rows. (In fact, in SQLite3, `explain` tells me it does, even when there are only two rows in the table.)

For this to work properly, `foox` has to be updated every time I insert a new row into `foo`. But this is tricky! If I have a million rows in `foo` and I add a new one whose `x` value is close to the minimum of the `x` values, then if `foox` is simply stored as a vector, the database might have to move a million values down by one in order to make room for the new `x` in its proper sorted order. There are several solutions to this, but the typical one (and what SQLite does) is to store your index in a B-tree, which allows you insert in the middle of it relatively efficiently.

A different approach, and more or less the one Lucene uses, is to accumulate your updates in a small "side file" until there are a lot of them, and then apply all of those updates at once to generate a new sorted `foox`. Until `foox` has been replaced with the new version, every query to `foox` must also check the side file to see if there are updates it's interested in; this can be made more efficient by sorting the side file, at which point you may begin to desire to have a side file for the side file.

The database index is data that depends on the table, and it needs to be possible to incrementally and transparently recompute that data when the table changes. This is the same kind of automatic recomputation problem that Spark, `make`, and `redo` attempt to solve.

Vesta

Vesta is a source-control system integrated with a build system; it

versions your whole build environment, and it runs each build step in an isolated chroot environment where it can only access data via Vesta, which functions as an NFS server. This ensures that the build step is only accessing a particular version of the source file, of the compiler, etc., and that Vesta can correctly record these dependencies.

Unfortunately, Vesta was proprietary for many years before finally being released publicly, at which point its authors stopped maintaining it; it had its own purely-functional programming language that you were required to use to describe the build process for your system; because its build scripts were written in that same language, building it from source required having a running instance of Vesta; and, since access was provided via an NFS-server interface, you had to have root (at a time before it was commonplace and easy to run virtual machines in VirtualBox, QEMU, or EC2) to try it. Vesta was used in production by DEC's Alpha processor design team for a couple of years, but perhaps because of obstacles like those mentioned above, it never achieved wide usage, even within DEC or Compaq.

git-annex

git-annex does not do any dependency management. Instead, it manages an efficient, decentralized, redundant immutable blob store with decentralized, replicated, eventually-consistent metadata, built on top of git. It uses symlinks to permission-read-only files to provide normal filesystem access to the immutable blobs without going to the lengths of implementing its own filesystem, as Vesta does; this is usually enough to prevent you from accidentally corrupting your local copy of the blobs, and if you do corrupt one copy of a blob, you can still restore it from a remote repository.

Spark, simplified and made flexible

Spark has the lineage and determinism stuff unnecessarily coupled to a bunch of stuff about keeping Java objects in memory and partitioning and records, which seems like it is somewhat extraneous, although apparently Java does kind of need that in order to run efficiently. (You'd think java.nio.MappedByteBuffer would have largely eliminated this problem, but apparently not.) Spark's ability to "perform in-memory computations on large clusters in a fault-tolerant manner" does not depend in any way on Spark's knowledge of the internal structure of partitions (that they contain records) nor of Spark's knowledge of which partitions are associated together to form an RDD. All Spark needs to know to recreate a chunk of data is really how to redo the computation that created that chunk, and how to recreate the inputs that went into that computation.

However, unlike `redo`, in-memory computations on large clusters definitely need to be able to produce multiple output chunks, or partitions or whatever. And, if those chunks are stored in distributed memory on a cluster, it's probably a good idea to send the computation to where the data is.

Also, the granularity of the computation is likely larger than what `redo` has to deal with, which means that the system has more latitude to do computations and heavy-duty setup than in `redo` or `make`. It could

run an A* search, for example, to choose among possible alternative plans.

So here's what I propose. Build steps nest. Build steps run in a contained environment, like what lxc provides, or if that's not possible, a directory full of symlinks to read-only files from a git-annex-like chunk store. When you launch a build step, which you can do from the Unix command line, you explicitly list its input files, including the script for the build step, which causes them to appear in its contained environment. When it terminates, it leaves behind an output directory. All of the input and output files, as well as the directories containing them, are stored in a distributed chunk store, and their provenance is recorded in a distributed, replicated metadata store; this is very similar to git-annex. Build steps are presumed to be deterministic, and they are isolated from their environment to the extent possible, which reduces their nondeterminism. So, if you invoke a build step for which the system already has the results cached, it will retrieve those results from the chunk store. And the system may invoke the build step on a remote machine, if that's where the data is, and then replicate the results onto your local machine.

Lxc/virt-sandbox might impose some 200ms launch overhead on build steps, but that should probably be okay. If it's not, Debian has fakechroot, which is an extremely efficient way to use a wrapper library to fake out some system calls to trick programs into thinking they don't have access to the whole filesystem.

This allows you to write your build script as a shell script, which can invoke other scripts, possibly in a loop, and use their results. Once you have run your build script, which will usually be instantaneous, you can be sure that all of the build results are present.

When a build step invokes a subordinate build step, it names the input files it wants to provide to the subordinate step using the filenames it knows them by, and optionally put them at a particular place in the child's namespace; but the system invoking the subordinate step uses their immutable blob hashes in the hash keys, not the filenames known by the parent build step. This means that the same build step, running the same commands identically many times in different contexts, can invoke a sub-build-step that does different things.

The top-level build script might also snapshot other parts of the firesystem, copying them into the blob repository, in order to make them available to subordinate build steps. For example, substantial parts of /lib and /usr/lib may be necessary.

(It would be nice to use sysdig or strace or something to figure out what files are actually being accessed, without having to write a filesystem.)

Lazy concatenation and merging

Ideally you could concatenate several chunks into a single big output file in a purely virtual way, i.e. without actually copying the data.

Performance back-of-the-envelope

I have an 800-gigabyte stock-market data set. If this were split into chunks of some 64 megabytes, it would be about 12500 chunks. If I

generate another 200 gigabytes of derived data from that, it will be another 3125 chunks. If each of those chunks derived from, say, 400 input chunks, the SHA-1s of all of those chunks (the lineage) would be 8 kilobytes; all the metadata together would be 25 megabytes. Incrementally replicating 25 megabytes of lineage data will be no problem.

Small-memory performance

One of the interesting things about MapReduce is that any algorithm that can be expressed with it can be implemented with a small number of sequential passes over the data, so you can use it to improve locality of reference as well as for fault-tolerant cluster-scalable computation.

Although this approach is not nearly as extreme as MapReduce in that way, it may also have some virtue in that direction, since it encourages you to break up your computation into small pieces that consume a few small inputs and produce a few small outputs; in cases like the Spark reimplementation of the Pregel programming model, this may involve "transposing" an algorithm, separating things that were previously together and bringing together things that were previously separate. If you were to run only one build step at a time, you might be able to improve your in-memory performance dramatically.

Additions 2016-06-22

Kragen Javier Sitaker, 2015-05-28 (updated 2016-06-22) (16 minutes)

Well, it's been a year and still nobody has done the above, so I guess I should do some work on it. What does a minimal executable version of it look like?

Maybe you have these pieces:

- a content-hash-addressable store which distinguishes between source data and cached data;
- a caching service which maps hashes of (deterministic) commands to hashes of their output;
- an execution service which returns the results of commands, either from cache or by computing them;
- a container to run commands in which grants them access only to read the data in their manifest and to produce an output tree;
- an ability for commands to delegate their output tree to further commands.

Do you really need all of those? What does the interface to creating output files look like?

Maybe a first step would be to think about what distributed MapReduce in this context looks like.

Topics

- Caching (p. 3361) (25 notes)
- Incremental computation (p. 3517) (24 notes)
- Databases (p. 3400) (20 notes)
- Spark (p. 3722) (3 notes)
- Vesta
- Revision tracking

Eur-Scheme

Kragen Javier Sitaker, 2007 to 2009 (13 minutes)

So I was thinking about how to simplify Ur-Scheme further. Ur-Scheme is about 1600 lines of code so far. Something like half or a third of that is assembly code, represented in Scheme, mostly implementing the basic types (strings, conses, closures, heap variables, symbols, booleans, characters, nil, fixnums) and the basic control structures (procedure call and return, including tail-call elimination, closure creation, conditionals, sequences). The primitives thus defined total about 1100 lines of assembly-language output, so they're probably somewhere around 1000 instructions. (But it doesn't have a garbage collector yet!)

Having these things implemented in assembly definitely helps it reach the reasonable performance level that it does. But it would be nice to have a system that was a lot simpler and more portable, even if it was slower.

eForth's approach

The eForth Model 1.0 was a Forth system written to be maximally portable; the x86 version apparently contains 171 machine-code instructions in about 28 primitives, and everything else is built up on top of that in interpreted Forth words. The primitives were something like the following set:

- `lit` push a literal
- `exit` return from an interpreted subroutine
- `execute` call a subroutine whose value is on the stack
- `(if)` branch if top of stack is 0
- `(else)` branch unconditionally
- `next` update loop counter and possibly branch
- `!`, `@` store and fetch a cell in memory
- `c!`, `c@` store and fetch a byte in memory
- `rp@`, `rp!`, `sp@`, `sp!` get and set the return and operand stack pointers (for e.g. exception handling)
- `r>` pop the return stack onto the data stack
- `>r` push the return stack onto the data stack
- `drop` discard an item on the operand stack
- `dup`, `over` copy the top or second stack items onto the top of stack
- `swap` exchange the top two stack items
- `0<` test whether the top item is zero
- `and`, `or`, `xor` bitwise operators
- `!io` I forget what this does
- `?rx` return true if there's input waiting
- and there were primitives to send and receive a byte over a serial line.

I don't remember how eForth accomplished rightward bit shifts (e.g. for division), and I don't have it handy at the moment.

The Eur-Scheme Ideal

Could we build a Scheme with a similar structure — as little as possible in assembly code? Would it be simpler?

Ideally we'd be able to implement not only the normal Scheme

types, but also as much as possible of the run-time system, in Scheme, on top of a minimal set of machine-code primitives. That would mean heap allocation, garbage collection, some parts of closure handling, variadic function calls, and all the types mentioned above; the basic system would only have to support (non-closure) function call and return, some conditional (I'm using `%ifeq`), and `set!`; and primitives for accessing memory.

Two Approaches To `cons`

The implementation of `quote` in a compiler is kind of annoying; it's straightforward in an interpreter or a load-and-go compiler, but in a compiler that might run in a different interpreter, and that generates an output file rather than compiling things into memory, its representations of data have to be compatible with those produced by the definition used at run-time, regardless of whether that run-time definition is in Scheme, C, assembly, or something else.

Ur-Scheme's `pair` structure, created by `cons`, consists of three machine words in order: the magic number `0x2ce11ed` to identify it as a cons, the car, and the cdr.

Here are the two definitions of `cons` from Ur-Scheme, the run-time definition and the compile-time definition used by `quote`:

```
;; We define a label here before the procedure prologue so that other
;; asm routines can call cons
(add-to-header (lambda () (text) (label "cons")))
(define-global-procedure 'cons 2
  (lambda ()
    (emit-malloc-n 12)
    (mov (const cons-magic) (indirect tos))
    (mov tos ebx)
    (get-procedure-arg 0)
    (mov tos (offset ebx 4))
    (pop)
    (get-procedure-arg 1)
    (mov tos (offset ebx 8))
    (pop)))
;; Compile a quoted cons cell.
(define (compile-cons car-contents cdr-contents labelname)
  (rodatum labelname)
  (compile-word cons-magic)
  (compile-word car-contents)
  (compile-word cdr-contents)
  (text))
```

I've been considering a slight variation in which assembly-emitting routines return the location in which their result was placed. That would shorten the run-time code for `cons` above to the following slight variation; I'm not sure if this is clearer or not, but it's certainly briefer, at 6 lines of body instead of 9.

```
(define-global-procedure 'cons 2
  (lambda ()
    (mov (const cons-magic) (indirect (emit-malloc-n 12)))
    (mov tos ebx)
    (mov (get-procedure-arg 0) (offset ebx 4))
```

```
(pop)
(mov (get-procedure-arg 1) (offset ebx 8))
(pop)))
```

It would be ideal if the two could be generated from a single structure specification, but at present there are only three in-memory types that can be generated at compile-time: pairs, strings (which are variable-sized and mostly consist of bytes), and symbols (which point to strings). So it's not clear that such a struct facility would be a net win for simplicity.

Primitives for Accessing Memory: The Eur-Scheme Approach

So the garbage collector, the heap allocator, and the stuff out of which `pair`s and `string`s and so on are constructed need to be able to access raw memory. And it's important that the garbage collector and the heap allocator not accidentally call the heap allocator — you could easily get an infinite recursive loop; so their interface to raw memory shouldn't require heap-allocation.

Perhaps the best way to do this would be to support basically a subset of C: allow routines to declare local variables that are just machine words, which can be used as pointers, are allocated on the stack or in machine registers, and are somehow ignored by the garbage collector (because they aren't necessarily Scheme values or pointers to Scheme heap objects).

But I'm going to talk about another approach, which I think will probably be simpler. You have a `pointer` data type, like the `string` and `pair` data types of normal Scheme, and an explicit stack to save and restore `pointer`s on; and a set of `pointer`s that exist from the time the program starts, and are therefore not heap-allocated. In order to

For concreteness, there could be eight callee-saves pointers, called `%g0`, `%g1`, `%g2`, `%g3`, `%g4`, `%g5`, `%g6`, and `%g7`, and eight caller-saves pointers, called `%c0`, `%c1`, `%c2`, `%c3`, `%c4`, `%c5`, `%c6`, and `%c7`. There might be other `pointer`s around. For example, there might be some `pointer` constants created by the compiler, or `pointer` variables allocated on the heap by routines that aren't part of the garbage collector or memory allocator.

Then we need only a set of primitives for accessing them:

- `(*put! foo %p0)` store the value of a Scheme expression in `%p0` and returns `%p0`.
- `(*get %p0)` returns the value stored in `%p0` as a Scheme expression value. If the value is not, in fact, a valid Scheme object, this is likely to crash the program.
- `(*load! %p0 %p1)` fetch the value `%p0` points to in memory, stores it in `%p1`, and returns `%p1`.
- `(*store! %p0 %p1)` stores the value in `%p0` into the place in memory pointed to by `%p1`, and returns `()`.
- `(*load-byte! %p0 %p1)` and `(*store-byte! %p0 %p1)` are analogous, but only load or store the single low-order byte.
- `(*push! %p0)` and `(*pop! %p0)` save and restore `pointer` values from a special `pointer` stack, which surely has some maximum size, but in any case does not allocate memory from the heap.

• (*get-sp! %p0) and (*get-fp! %p0) get the machine-level stack pointer and frame pointer registers of the caller and store it into %p0, to support stack introspection (e.g. garbage collector tracing or backtrace printing.). They return %p0.
• Optionally, (*set-sp! %p0) and (*set-fp! %p0) set those machine registers, and (*get-psp! %p0) and (*set-psp! %p0) get and set the pointer stack pointer. These primitives could be useful for exception handling, but isn't necessary for the applications I've discussed so far.
• (*add! %p0 12 %p1) does pointer arithmetic, storing the contents of %p0, plus 12, into %p1, and returns %p1.
• (*add! %p0 %p2 %p1) does the same thing, but uses the contents of %p2.
• *and!, *or!, and *xor! are analogous to *add!.
• (*unsigned-<? %p0 %p1) returns a Scheme true or false value: true if the contents of %p0, interpreted as an unsigned number, are less than those of %p1, and false otherwise.

These 20 or so primitives are more or less sufficient for implementing things like the garbage collector and high-level data structures. None of them are particularly complicated; here's a possible implementation of the body of *load!:

```
mov (%ebp), %eax        # fetch argument zero, %p0 (src)
        call ensure_pointer     # error unless it's a pointer object
        mov 4(%eax), %ebx       # fetch its contents (the location to load from)
        mov (%ebx), %ebx        # fetch the pointed-to value from memory
        mov 4(%ebp), %eax       # fetch argument one, %p1 (dest)
        call ensure_pointer     # error unless it's a pointer object too
        mov %ebx, 4(%eax)       # store into its contents.
```

Those seven instructions are wrapped inside seven more instructions for procedure prologue and epilogue. This suggests that the total number of assembly instructions that you *need to write* to implement these primitives on a new CPU is somewhere around 7 * 20 = 140, plus the basic conditional, procedure call and return, and variable access. So you could probably port such a Scheme to a new CPU architecture with 300–400 lines of code or so, around twice the amount needed to port the eForth Model 1.0.

Of those 150–200 instructions that you need to write, some are in procedure prologues and epilogues that get duplicated 20 times for those 20 or so primitives. So you might end up with something like 400–500 lines of assembly output for the primitives.

Two More Approaches to cons

Again, the nested-assembly version of cons looks like this:

```
(define-global-procedure 'cons 2
  (lambda ()
    (mov (const cons-magic) (indirect (emit-malloc-n 12)))
    (mov tos ebx)
    (mov (get-procedure-arg 0) (offset ebx 4))
    (pop)
    (mov (get-procedure-arg 1) (offset ebx 8))
    (pop)))
```

The "subset of C" approach might look like this:

```
(define (cons car cdr)
(let-pointer ((rv (malloc-n 12)))
  (pointer-write (indirect rv) cons-magic)
  (pointer-write (offset rv 4) (pointer-val car))
  (pointer-write (offset rv 8) (pointer-val cdr))
  rv))
```

The Eur-Scheme approach might look like this:

```
(define (cons car cdr)
(*store! cons-magic (malloc-n 12 %c0)) ; cons-magic is a pointer constant
(*store! (*put! car %c1) (*add! %c0 4 %c2))
(*store! (*put! cdr %c1) (*add! %c0 8 %c2))
(*get c0))
```

`cons` doesn't have to save and restore the values of those `pointer`s because they're all caller-saves, and `cons` isn't calling anything else that might use them for its own purposes.

I think that's a substantial improvement in both readability and simplicity over the current assembly version, and it's not obviously worse than the subset-of-C version. This example leads me to suspect that defining routines in this fashion, rather than in assembly, would make for a considerably more concise and comprehensible Scheme system than Ur-Scheme.

References

Ur-Scheme is a subset-of-Scheme compiler I wrote to learn how to write compilers. It compiles itself; it's reasonably fast, despite being safe, and very small.

Topics

- Small is beautiful (p. 3714) (40 notes)
- Forth (p. 3461) (19 notes)
- Compilers (p. 3383) (16 notes)
- Lisp (p. 3552) (9 notes)
- Scheme (p. 3694) (8 notes)
- Ur-Scheme (p. 3766) (3 notes)

Automatic dependency management

Kragen Javier Sitaker, 2015-05-28 (updated 2015-09-03) (5 minutes)

A variety of software systems have used some kind of automatic dependency graph tracking to automatically recompute things, from Lotus 1-2-3 (and derived spreadsheets) up to current JavaScript frameworks like ReactJS and current big-data frameworks like Apache Spark. I'm thinking about some ideas related to this, especially inspired by Spark, and I thought I would look around and see what already exists.

It turns out a lot of fucking things related to this already exist, so I thought I'd write down a summary of some of them.

There are way too many for me to summarize, though.

Candidates not yet noted below: make; React; Meteor; redo; materialized views in databases; STMs; git-annex; database indexes; tabled predicates in Prologs; Merkle trees; immediate-mode GUIs.

Spreadsheet recalculation

Bob Frankston said in March 2015 that VisiCalc, the original spreadsheet, didn't include "natural order" (i.e. dependency-driven) recalculation, in order to fit into 16 kilobytes. He was responding to a tweet from Mitch Kapor explaining that Rick Ross had implemented that for the first time, in 1982, in Lotus 1-2-3. VisiCalc, instead, had an option to recalculate by rows or by columns. Since 1-2-3, though, spreadsheets default to dependency-order recalculation.

Dependency-order recalculation is comparatively easy for spreadsheets (although infamous thieves Rene Pardo and Remy Landau still got a US patent on it in 1983; Pardo claims to have done it in an "electronic spreadsheet" called LANPAR in 1969, but his lawyer denied it in court), because the total number of things that could possibly be recalculated is human-scale, all of them can be recalculated in only a single way, and so you can simply enumerate them and do a topological sort.

Nowadays, dependency-order recalculation is approximately unnecessary in spreadsheets; computers are so fast that human-scale spreadsheets could recalculate in milliseconds, so recalculating the whole spreadsheet after every keystroke would be reasonable. (They don't, but that's another story.)

ReactJS

ReactJS

Deterministic builds

Tor: https://blog.torproject.org/category/tags/deterministic-builds

Chromium: https://www.chromium.org/developers/testing/isolated-testing/deterministic-builds

Debian: https://wiki.debian.org/ReproducibleBuilds

Firefox: https://bugzilla.mozilla.org/show_bug.cgi?id=885777

ccache

ccache is a build accelerator specific to C, C++, and Objective-C. It hashes your source code, include files, compiler (well, typically just its size, mtime, and name), command-line options, etc., with MD4, and stores the compiler's output (including e.g. warning messages) in an on-disk cache in, normally, your home directory; future recompilation attempts whose inputs haven't changed will just reuse the previous compiler output.

In theory, this means that you could get most of the benefits of `make` for C and especially C++ programs by just wrapping your compiler in `ccache` inside your build script; rerunning the build script would rehash all your source files, copy the object files into your current directory, and then relink the executable. Depending on what you're compiling, this might be almost instantaneous, or it might be very slow.

Because Unix provides `ccache` with no reliable way to get a secure hash of the source files from the filesystem, it has to read them in their entirety to figure out whether they have reverted to an old version. I tried it just now on my netbook on a tiny three-kilobyte GLUT program, and it ended up reading about a megabyte of `.h` files in order to figure out that it could safely reuse the 2.7kB `.o` file from a previous compilation, taking 39 milliseconds in all, even though it made only about 365 system calls.

Spark

https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Spark Streaming

https://spark.apache.org/docs/latest/streaming-programming-guide.html

Vesta

http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-177.pdf http://www.vestasys.org/

Bup

Avery Pennarun

https://github.com/apenwarr/bup/blob/89ac418d84e29ba482bbd21ebc1172c2d1ff5507/DESIGN https://github.com/bup/bup https://bup.github.io/

Truth maintenance systems

In Stallman & Sussman 1976, describing their pre-SPICE circuit simulator, we find, "If a user changes some part of the circuit specification (a device parameter or an imposed voltage or current), only those facts depending on the changed fact need be 'forgotten' and re-deduced, so small changes in the circuit may need only a small amount of new analysis." They are describing their invention of "dependency-directed backtracking", which later became known as a "truth maintenance system", and it's built with generalized constraint

propagation, which is substantially more general than the unidirectional dependencies mentioned in the other systems above, and one that supports finding a contradiction and backtracking from it to undo the set of incorrect guesses that led to it, and avoid that set in the future. You could use this kind of system, for example, to solve Sudoku puzzles rapidly.

A TMS, like the other systems above, remembers how every datum was deduced, but it does so not in order to promote computational efficiency by caching results, but rather to track the sources of problems — in this case, logical contradictions.

Self-adjusting computation

http://www.umut-acar.org/self-adjusting-computation

Topics

- Performance (p. 3621) (149 notes)
- Caching (p. 3361) (25 notes)
- JS (p. 3533) (12 notes)
- Dependencies (p. 3405) (7 notes)
- Umut Acar's "self-adjusting computation" (p. 3702) (6 notes)
- Spreadsheets (p. 3728) (3 notes)
- Spark (p. 3722) (3 notes)
- Deterministic builds (p. 3408) (2 notes)

Dollar auctions and tournaments in human society

Kragen Javier Sitaker, 2013-05-17 (7 minutes)

Dollar auctions

A "dollar auction" is a simple kind of grift that involves no deception. You auction off a dollar bill, starting bids at five cents, under slightly unusual rules: the winning bidder pays their bid and gets the dollar, but the losing bidder pays too --- they just don't get the dollar. (In some variants, it's only the runner-up who pays; in others, everyone who bid.)

So, you can imagine, initially people are eager: they can get a dollar for five cents, if nobody else bids. But once other bidders start in (after all, they can get a dollar for ten, or fifteen, or twenty-five cents) the early bidders are faced with a difficult choice: lose their five or ten cents for nothing, or increase their bids to fifteen or twenty cents, netting some eighty cents --- less than they'd hoped, but gaining eighty cents is better than losing ten.

And the other bidder has the same incentive: they can bid the price up to thirty cents (gaining seventy cents) or stand pat at fifteen (losing fifteen). And so the bids keep going up.

Typically the bids will reach two or three dollars before the bidders give up. All the bidders lose money, even the winner. The auctioneer multiplies her dollar. None of the bidders want to play again.

Now, I don't know how to analyze this game with game theory. But, in terms of human psychology, it seems straightforward: however much you're losing now, you'll lose less if you increase your bid just that little bit more and win the dollar --- and sooner or later the other bidder will give up. But at some point it becomes clear that they probably won't give up until you've raised your bid by another entire dollar.

It's a great deal for the auctioneer.

But what does that have to do with the real world? We don't see dollar-auction epidemics, do we?

Graduate school

To get a Ph.D. in, say, neuroscience, you have to spend some five or ten years working as a Ph.D. student, typically as a research assistant in your advisor's lab. The RA position requires great intelligence and extremely specialized skills, the skills that will eventually justify granting you a doctorate, but it's paid just above minimum wage. Also, it typically involves a great deal of painstaking and boring work. So it represents a substantial opportunity cost to the student, at the end of which time you're awarded a Ph.D. in neuroscience.

But the Ph.D. is, in itself, worthless. (They usually don't even inscribe them on real vellum anymore.) It's a sort of license to apply for a neuroscience professorship [0], or a similar position in industry. These are really great jobs: you become world-famous (within the neuroscience community --- that is to say, there are thousands of

people around the world who will recognize your name and think you're cool), you get paid pretty well, the vacations are fantastic (without even taking into account travel for conferences), you have a staff of very intelligent and skilled research assistants to do the painstaking and boring part of your work for you, and in academia, the tradition of academic freedom means you can say almost whatever you want without fear of reprisal. (Unless it represents academic fraud.)

But there are many fewer such positions opening up each year than there are new Ph.D.s graduating. I don't have the numbers handy here, but it's something like one tenure-track position for every ten graduates. In order to get one of those positions, you have to appear better than somewhere around nine tenths of the other neuroscience grad students: based on the papers you published in grad school and whatever else you can persuade the school is relevant.

Which is to say, all of the Ph.D. students put in their years of hard work at low pay, but only the most productive ten percent actually win the "auction". The other nine tenths are left with an impressive degree and no job in the field.

Sounds a lot like the dollar auction, doesn't it? The only question is whether the graduate students are "bidding up" the "price" of a new tenure-track position to the point where it's "worth less" than what you "paid" for it, in terms of reduced earnings and scutwork. The psychology of the dollar auction suggests that they will, given the opportunity.

Who benefits in this case? Well, human knowledge, arguably the professors (certainly the well-established ones who had much less competition, thirty or forty years ago), and the university administrators.

[0]: I'm simplifying a bit here; the actual professorship is what you get after, typically, another ten or fifteen years of postdocs, adjunct faculty positions, and associate and assistant professorships.

iOS and Android Apps

You've probably heard the story of iFart, the 99-cent [1] iPhone app that plays recorded fart sounds and which sold millions of copies. Every programmer hears about it from their family, who want to know why they're not rich too. Couldn't you have written iFart?

But it turns out that nearly all the apps in the Apple iPhone App Store sell only a few hundred copies. There are hundreds of thousands XXX of iOS apps now, but only N million iPhone users (let's say 100M) who each only have around 100 apps installed. That means ten billion app installs, which means an average of 100k installs per app.

Who benefits? Apple, Google, and arguably the users of their devices

[1] I can't remember the details or look them up at the moment, so they may be wrong. It might have cost three dollars, say, or only sold hundreds of thousands of copies.

Garage bands

Most bands, famously, never earn any royalties; the record label pays them an advance against the royalties, but they never make enough royalties to pay back the advance. The advance itself is paltry

money. XXX

How to kick Kickstarter's ass

So the dollar-auction dynamic XXX

How about making the reward levels fixed-count instead of fixed-dollar? Say, the top ten donors each get a gold-plated widget, the next twenty each get a nickel-plated one, and the next forty each get a widget. Could this inspire the donors to donate more than they do today?

Topics

Capacitive droppers and transformerless power supplies

Kragen Javier Sitaker, 2019-09-18 (11 minutes)

Looking at STMicroelectronics's appnote AN392, "Doc ID 1863". It's a little bit shocking, though hopefully not literally; it contains a very cost-optimized, compact power supply circuit for microcontroller projects that can be kept safely away from the delicate bodies of the humans.

Quoting Horowitz and Hill:

> Never build an instrument to run off the powerline without an isolating transformer! To do so is to flirt with disaster. Transformerless power supplies, which have been popular in some consumer electronics (radios and televisions, particularly) because they're inexpensive, put the circuit at high voltage with respect to external ground (water pipes, etc.). This has no place in instruments intended to interconnect with any other equipment and should always be avoided. And use extreme caution when servicing any such equipment; just connecting your oscilloscope probe to the chassis can be a shocking experience.

Consequently they do not explain how to build such consumer electronics (though they promise that chapter *9x* will.) This appnote is about how to do it, a topic also occasionally discussed in the YouTube channel of bigclivedotcom. So how do they suggest doing it?

Outline of the capacitor-dropper circuit

They're running an ST6210 8-bit microcontroller directly off the hot side of mains current, connected to the touch sensor at the human's finger through two or three 4.7-MΩ resistors (thus at 340V peak you get 36 microamps or less, not enough to feel). The 5-volt rail is literally the hot side of the power line (though fused), but the "ground" is floating; a simple regulated capacitor dropper produces about 5 volts on the ground rail as follows. The neutral side of the power line is connected to an 820Ω half-watt resistor, which is connected to a 220-nF 400-V capacitor, which is connected to the low side of a 5.6-volt zener diode, whose positive end is connected to the 5-volt rail. A 1N4148 clamps the ground rail to be no more than 0.7 volts above the low end of the zener, and there's a 100-µF 10-volt energy storage capacitor connected between the ground and the 5-volt rail.

(This appnote is from 1998, so this is not capacitive touch sensing; instead, the human's finger forms part of a voltage divider between the hot and neutral powerline rails, protected by the megohms.)

How the regulated capacitor dropper works

This is not quite a standard capacitive voltage divider circuit.

The 220-nF capacitor produces a 12–15-kΩ reactance at 50–60 Hz, losslessly limiting the current to 20 mA by itself; I assume the 820-Ω resistor is to limit inrush current, because it has ≪1% effect at powerline frequencies. At steady state, with the storage capacitor charged, this current sloshes back and forth across the zener (remember that the high end of the zener is directly attached to the hot side of the power line), but when the zener is reverse-biased and

the capacitor is discharged, there's an easier return circuit path through the capacitor and 1N4148, which initially only has a 600-mV voltage drop before starting to conduct heavily rather than the zener's 5.6 volts. Once the capacitor is charged, though, the zener becomes the easier current return path, and the return current flows through it instead.

20 mA through 820 Ω gives you 330 mW, which is why you can't use a ¼-W resistor. The zener is also going to dissipate 55 mW. An ideal dropper capacitor wouldn't dissipate anything but real dropper capacitors will have some dielectric heating from the constant 20 milliamps, or 10 milliamps at 120Vrms; this will probably be on the order of 1 mW. The rest of the circuit should have much smaller power consumption.

They say their board uses 3 mA; the 100-µF storage capacitor will then discharge at 30 V/s, or 0.6 V in the 20 ms between 50-Hz peaks.

The results: regulated 5V from the powerline with five discrete passives

So with five simple passive components and no electromagnetics you get a regulated 5 volts directly out of the power line. It's just not safe to touch the circuit while it's turned on, or to connect it to any other circuits, and the dropping capacitor is a bit of a beast. In Capacitors: some notes on tradeoffs (p. 134) I have some pricing info on smaller, totally inappropriate capacitors; the 19.4¢ Nichicon UWT1H470MCL1GS is 47µF and 50V, so you would need strings of 8 of them to get the required 400 volts, but each such string would only have 6 µF, so you would need 38 such strings in parallel for a total of 304 capacitors costing a total of US$60. Capacitors designed for such applications would probably be smaller, cheaper, and cooler.

You probably don't need a bleeder resistor across the dropper cap (and the ST appnote doesn't show one); if you unplug it at the wrong part of the cycle, it could have 340 volts across it, but since it's only 220 nF, that's only 18 mJ, not enough to be dangerous to a human. It'll blow holes in your MOSFETs, though.

Unlike a standard capacitive voltage divider, the output voltage barely depends at all on the input voltage or the dropper capacitance. The dropper capacitor just serves to limit current and thus power dissipation. The output voltage is determined by the zener, or more precisely the zener minus the rectifier diode's forward voltage; the only relationship to the input voltage is that the output can't be more than the peak-to-peak input voltage (minus a couple of diode drops) and if it gets close you will have less current draw. Moreover, this power supply always draws very nearly the same current, whether anything is running from it or not, unless the current load is so high as to substantially drop the voltage in the power-storage capacitor.

This means, of course, that its efficiency is always terrible: as bad as 0% (when the load is turned off) and never much more than 30% or so.

The context: controlling mains power with triacs

The appnote isn't about regulated capacitor droppers; it just

mentions in passing that "the board supply comes from the mains through a simple RCD circuit". The appnote is actually about controlling triacs ("the least expensive power switch to operate directly on the 110/240 V mains"); it recommends using a microcontroller to inject a turn-on pulse at the appropriate point in the cycle. In the appnote ST recommended a BTA 16-600CW triac for motor control so as to need no snubber, but it needs 60 mA to trigger it. The more common (?) 95¢ T405Q-600B-TR I mentioned in My attempt to learn about jellybean electronic components (p. 1974) would work and only needs 5 mA, but would presumably require a snubber.

So in ST's appnote the inefficiency of this power supply is insignificant: if you're controlling a 700-watt vacuum cleaner with a 15-mW microcontroller, it hardly matters that you're burning 200 or 400 mW to get a regulated power supply, as long as you don't have dozens and dozens of vacuum cleaners plugged in for every one you're using.

Capacitive droppers for micropower IoT?

In Notes on the STM32 microcontroller family (p. 3176) I calculated datasheet power consumption for a number of STM32 processors; even without using power-down modes, a number of them would run at under 50 µA at 131 kHz, and with power-down modes you could reasonably reduce power consumption by another factor of 1000, although as mentioned in Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509), it's easy to leak multiple microamps through your bypass capacitors. This suggests that 10 µA might be a reasonable current to design an embeddable powerline-powered IoT device for. (As long as it doesn't have to be controlling a triac or something, anyway.)

Adapting the above design for the lower current level, though, the dropper capacitor could be 220 pF instead of 220 nF, the storage capacitor could be 0.1 µF instead of 100 µF, and the input resistor would, I think, be unnecessary. Using the $C = \varepsilon A/d$ formula for capacitance, ε_r of 3 and an egregiously large plate separation of 1 mm, you could get the capacitance you need from 83 cm² of foil "shielding" wrapped around a plastic-insulated electrical line. (3 is a reasonable guess for many plastics.) That's a rather large chunk of foil, though, and a better option might be to run a high-efficiency buck regulator off a smaller piece of foil instead of just regulating with a zener.

If you're going to try to go with such a capacitive connection, you might want to do it on both the hot wire and the neutral wire, despite this requiring four times as much foil. That way, you have no dc connection to the powerline at all.

In theory at these current levels you could use reeally thin wires. In Balcony battery (p. 2377) I estimated that 142-µm copper wire would probably work for five-amp fuse wire. Suppose that's correct. Here the wire needs to carry ten thousand times less current, so it can have ten thousand times less surface area per resistance, which means it could be 21 times narrower and thus have 21 times less surface area and 441 times more resistance. But that would be 7-µm copper wire, which is going to be hard to find and maybe even a bit dangerous to

handle. If we take copper's resistivity to be 16.78 nΩ·m, as in Executable scholarship, or algorithmic scholarly communication (p. 2137), that wire is 436 Ω/m.

I'm pretty sure a 220-pF 400-V X7R ceramic capacitor would be a few millimeters in size and cost under a dollar, and that's a much better option than meters of foil snaking around your conduits and junction boxes. But it means that you need a direct non-galvanically-isolated electrical connection to the mains power and thus a fuse.

Topics

Real time windowing

Kragen Javier Sitaker, 2017-08-03 (9 minutes)

Your user interface is a hard real-time program and should be designed and implemented as such.

I've been thinking a lot about how to build a full computing environment "from scratch", in the sense that it wouldn't depend on any previously-designed hardware or software. This is for a few different reasons. One is Dan Ingalls's contention that if the system is too complicated for a person to understand, they are limited in how they can use it as a medium for creative expression. Another is that current systems are insecure by design, and for economic reasons, this is unlikely to change. A third is that current systems are irreversibly addicted to the astoundingly-high-performance computing hardware that we currently only know how to produce with correspondingly-astounding concentrations of investment, to the point that only three companies in the world — TSMC, Intel, and Samsung — are currently able to produce devices at the 10-nm process node.

But a fourth reason is that in some important ways, in particular user interface responsiveness, current systems are dramatically worse than they could be; fixing it will require a rewrite. There are other such ways, such as security, but I will focus on responsiveness here.

Hard real-time programming versus normal programming

There's an established discipline of "hard real-time programming", which I will distinguish from normal programming by first describing normal programming. In normal programming, the most important thing is that, if your program runs successfully, the output it produces is correct; the second most important thing is that it run successfully, or at least produce a helpful error message explaining what caused it to fail; and, if these criteria are met, it is always more desirable for the software to run faster, especially on average, so we would like the software to be as fast as possible, as long as that does not interfere with correctness or success.

Hard real-time programming is very different from normal programming; it's programming software for things like antilock brakes and jet engines. What distinguishes it from normal programming can be summarized in two slogans:

- Late answers are wrong answers.
- Failure is not an option.

That is, in hard real-time programming, each piece of output has a deadline; if it misses that deadline, it is worthless. Overdue answers are unusable. Consistency of execution time is more important than its shortness. Also, almost invariably, it is not acceptable for a program to fail some of the time, for example due to lack of resources. The antilock braking program must work every time you press the brake pedal, not 99% of the times.

Hard real-time programming isn't "hard" in the sense of

"difficult"; you can do it on an Arduino. It's "hard" in the sense that the deadlines that the program's execution must meet are hard.

Hard real-time programming and what I'm calling "normal programming" are two ends of a spectrum, representing two different reasons we might use computers. In normal programming, we run a program because we do not know what the output will be, but we want to find out, because we imbue it with some kind of meaning. In hard real-time programming, we know exactly what the output will be, and rather than goggling at it, we want to use it to control some aspect of our world.

In between there are many shades of gray, and most programs have some aspects of both. But it isn't unidimensional. For example, in "soft real-time programming", it's important to not miss deadlines, but results computed too late for their deadline are still of some use. In another category, somewhat incorrect answers are acceptable, as long as they are on time; there exists a whole field of algorithm design for this, known as "anytime algorithms", many of which are for mathematical optimization problems. More about this later!

Most of academic computer science focuses on normal programming, and most programming languages and tools are written with normal programming in mind. Enormously more code is written for normal programming than for hard real-time programming.

As a consequence of these profound differences, it is often impossible to repurpose code written for a normal program as a real-time program, or vice versa.

The user interface should be written as a hard real-time program

My core argument is that the part of the user interface closest to the user should be written as a hard real-time program, for the following three reasons:

• because human beings are very sensitive to user interface latency;
• because a crashing user interface is very similar to a whole crashing computer (or maybe a whole crashing cluster of computers);
• because they interact directly with video hardware, which generally pumps out frames of video at a constant rate, so pixels that are computed too late will be lost.

By "the part of the user interface closest to the user" I mean the hardware and code that interfaces with human-interface output devices, such as sound cards, video cards, or laser projectors, and human-interface input devices, such as touchscreens, accelerometers, microphones, cameras, mice, keyboards, and joysticks. If the same interface device is used as an interface to more than one different program, the user interface code in question has the task of safely multiplexing that device among those programs.

How are hard real-time programs written?

They are much smaller than other programs, because — partly because of being written with tools designed for normal programming, and despite what I said above about "hard" — they are actually harder to write than normal programs.

Typically they don't do any dynamic memory allocation, both because dynamic memory allocation generally takes a nondeterministic amount of time and because it can fail, which is not an option. As a result, they usually run in statically bounded memory space, making them formally executable by finite-state machines.

Typically when they do do dynamic memory allocation, they use simple arena allocators or per-type allocators, which are constant-time.

They virtually never use virtual memory. Virtual memory is instant death to hard real-time systems, except in the unusual case that the deadline is so long that it can tolerate disk seeks.

Typically they use very simple algorithms, because these are often easiest to prove time bounds for and because more complicated algorithms often require dynamic allocation.

More often than you would expect, they are written in assembly language, which simplifies the task of worst-case time analysis. Occasionally they are written in ladder logic or other such hardware-focused formalisms.

Often, they run on dedicated hardware, which is often a stripped-down microcontroller; timing becomes more predictable with less peripherals, no virtual memory, and less tasks per processor. Since in a sense the desired outputs are already known before the program starts, it is often possible to perform useful tasks even with very little processor power, so even today, many hard real-time systems run on 8-bit processors such as 8051s, PICs, and Z80s.

Formal methods of logical proof are somewhat more frequently used for hard real-time programs than for normal programs.

The above is, of course, somewhat idealized. But it provides the general outlines of the situation.

Anytime algorithms are not currently widely used for hard real-time programming, perhaps because they are viewed as exotic. An "anytime algorithm" is one that can provide an answer at any time after starting, but will produce better answers if allowed to run longer. Most often, these work by computing a series of progressively better answers, each an improvement on the previous answer. Many mathematical optimization algorithms, such as those often used to approximately solve constraint-satisfaction problems, work in this mode normally.

Instead of anytime algorithms, hard real-time programs typically use algorithms that execute in constant time or a time with a hard upper bound. Instead of amortized-constant-time or amortized-logarithmic- time algorithms, they must use algorithms with a worst-case constant or logarithmic time bound, and keep their dataset size small enough that the necessary linear-time-or-worse algorithms don't miss deadlines.

It's common to use bounded-size nonblocking FIFOs to communicate with real-time code.

How can a windowing server be a real-time program?

So suppose we undertake to make a user interface layer

Topics

Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums

Kragen Javier Sitaker, 2018-06-05 (4 minutes)

All of what is below the line below is somewhat wrongthink. A cube is the wrong shape; what you want is some set of perhaps abundant, perhaps relatively prime *lags* with (possibly running) totals along each lag.

Running totals make incremental updates in the middle inefficient, but they permit efficient selection of arbitrary time-domain ranges. Running *second-order* totals permit efficient selection of arbitrary time-domain trapezoidal windows, which may actually be more valuable in many cases.

Here's an example:

```
 b   a             b   a             b   a             b   a             b
o

 2,  7,  6,  8,  8,  1,  9,  7,  2,  9,  8,  7,  1,  2,  3,  5,  4,  4,  6,  7,  4
o x

 2,  7,  6,  8,  8,  3, 16, 13, 10, 17, 11, 23, 14, 12, 20, 16, 27, 18, 18, 27, 20
o running sum by 5

 2,  7,  6,  8,  8,  5, 23, 19, 18, 25, 16, 46, 33, 30, 45, 32, 73, 51, 48, 72, 52
o running sum ii by 5

 2,  7,  6,  8,  8,  1, 11, 14,  8, 17, 16,  8, 12, 16, 11, 22, 20, 12, 18, 23, 15
o running sum by 6

 2,  7,  6,  8,  8,  1, 13, 21, 14, 25, 24,  9, 25, 37, 25, 47, 44, 21, 43, 60, 40
o running sum ii by 6
 c                   c                   c                   c
```

If we want the sums of the periodicity-5 component of x, we can just take the 5 last values of the second row. For example, 27 (column a) is 7 + 9 + 7 + 4, and 20 (column b) is 2 + 1 + 8 + 5 + 4. If we want the sums of the periodicity-5 component of some substring of x, we can subtract the 5 corresponding values from earlier; for example, the values 7, 6, 8, 8, 3 represent the totals after the first 6 elements, and we can subtract those from the final 5 elements to get the sums over this shorter interval.

The "running sum ii" lines are the running sums (with lags) of the corresponding running sum lines. This allows us to compute average values of the running sum over some arbitrary interval; with two such average values of the running sum, we can calculate the sum over an interval of the original signal, but with fuzzy boundaries on

the window.

Suppose that as you acquire samples from some signal, you assign them raster-wise to elements of a 6×5×7 "cube", maintaining a running total of the samples for each of the 6×5 one-dimensional slices along the Z dimension. This requires two memory updates (one add) per sample. When you finish with one such 210-sample cube, you move on to another.

Now, if you want to take the dot product of a 210-sample cube with an arbitrary waveform of period 6, you can start by generating the stride-6 totals from the pre-existing totals on the 6×5 face of the cube, which requires 24 additions. Then you perform your 6 multiply-adds and get your result.

You can take those 30 totals and add them up differently to get the periodicity-5 component of the waveform.

This structure accelerates several other variants of the same computation, too:

- If you only want to perform the dot product on a fortunately aligned 30-item or 60-item substring of the samples, you can take one or two rows of the 5×6 totals, rather than all of them.
- If you want to take dot products with several different period-6 waveforms, you can use the same totals.
- If the waveform is actually of period 2 or 3, rather than 6, you can do 4 or 3 additions instead of some of the multiply-adds.

Other dimensions may have different advantages. 12×14×13, for example, with 12×14 totals, gives you somewhat efficient dot products with waveforms of 2, 3, 4, 6, 7, 12, and *24* samples — because the 12×14 totals can be added up in groups of 7 to get the period-24 wave.

This data structure, then, allows you to do certain computations with short-period waveforms at very low cost, while also permitting efficient incremental updates.

Topics

Who is inventing the future in 2013?

Kragen Javier Sitaker, 2013-05-17 (1 minute)

Alan Kay, whose team invented object-oriented programming and the modern graphical user interface, famously said, "The best way to predict the future is to invent it." If that's true, then the people who *control* the future are the ones who are inventing it. So let's invent a future we want to live in!

Who are the people and projects inventing a future worth living in today?

• Yochai Benkler, inventing the economics of the post-scarcity world.

• Michel Bauwens, also, with his Peer-To-Peer Foundation.
• Marcin Jakubowski, and his Factor e Farm/Open Source Ecology/Global Village Construction Set project. See http://opensourceecology.org/wiki/Marcin_Log for what he's doing today.
• Nick Mathewson and the rest of the Tor project, trying to enable anonymous speech in the age of the internet.
• Kickstarter.
• Wikipedia, which has organized the world's knowledge and made it universally accessible and useful, completing the project of Diderot's Encyclopédie a few hundred years late. Jimmy Wales is its most visible spokesperson, but the vast majority of work is being done by thousands, if not tens of thousands, of volunteers.
• The Internet Archive.

Topics

Immediate mode productive grammars

Kragen Javier Sitaker, 2018-09-13 (8 minutes)

Immediate-mode GUI libraries, which are popular for some kinds of games nowadays, allow you to define your GUI structure by using the execution trace of a callback instead of an in-memory data structure. They can be used with immediate-mode graphics libraries to draw the GUI bit by bit as the function runs, requiring a really minimal amount of RAM. Aside from the advantage in memory consumption, this approach reduces or eliminates the cache-invalidation problem of redrawing the view when the underlying model changes; whenever you draw a frame, your drawing function fetches the most current state directly from the model.

An interesting feature of this approach is that there are at least two different reasons the library might call your callback: to *draw* the GUI or to *react to an event* in the GUI. For example, you might have a `button(x, y, w, h, label)` function with a boolean return value; when called in drawing mode, it draws the button and always returns false, but when called in reacting mode, it draws nothing, and returns true if the user just clicked on the button. So really your callback isn't so much a "draw interface" callback as a "describe interface" callback.

So I was thinking about using this approach for serialization of data structures.

Example application

I'm writing a VNC server, and the VNC protocol (though a beautiful dream compared to, for example, the Spice protocol or the X11 protocol) has some godawful things like `PIXEL_FORMAT` and `SetEncodings` in it. `PIXEL_FORMAT`:

No. of bytes	Type [Value]	Description
1	U8	bits-per-pixel
1	U8	depth
1	U8	big-endian-flag
1	U8	true-color-flag
2	U16	red-max
2	U16	green-max
2	U16	blue-max
1	U8	red-shift
1	U8	green-shift
1	U8	blue-shift
3		padding

`SetEncodings`:

No. of bytes	Type [Value]	Description

1	U8 [2]	message-type
1		padding
2	U16	number-of-encodings

This is followed by number-of-encodings repetitions of the following:

No. of bytes	Type [Value]	Description
4	S32	encoding-type

As it happens, the server is supposed to be able to either encode or decode `PIXEL_FORMAT`. It would be nice to be able to describe the `PIXEL_FORMAT` encoding with a function and derive both the decoder from it automatically; something like this, in Golang syntax:

```
func (p *PIXEL_FORMAT) format() {
        u8(p.bits_per_pixel)
        u8(p.depth)
        boolU8(p.big_endian_flag)
        boolU8(p.true_color_flag)
        u16(p.red_max); u16(p.green_max); u16(p.blue_max)
        u8(p.red_shift); u8(p.green_shift); u8(p.blue_shift)
        padBytes(3)
}
```

Now, as it happens, the way I'm doing this right now is with the Golang `encoding/binary` module, which uses reflection to slowly read the above description from the definition of a struct type:

```
type PIXEL_FORMAT struct {
        bits_per_pixel, depth, big_endian_flag, true_color_flag byte
        red_max, green_max, blue_max                            uint16
        red_shift, green_shift, blue_shift                      uint8
        _, _, _                                                 byte
}
```

That's all the code that's needed to describe the format above to `encoding/binary`, which can then both write and, in theory, read it (though I haven't actually tried this yet), and this is awesome. But `encoding/binary` doesn't support any variable-length data like the `encoding-type` list in `SetEncodings`, nor can it do things like "deserialize a client-to-server message", in which the first byte of the message contains the message-type.

This suggests an approach reminiscent of recursive-descent parsing with backtracking, which is a simple and fully general approach to parsing context-free languages, although it takes exponential time in the worst case (though see the Packrat parsing algorithm for a fix). When parsing, the input stream can manage an error status and a stack of backtracking points; when a parse fails, it sets the error status, which prevents further parsing from doing anything until the format

function backtracks to a non-erroneous backtracking point. This allows ordered choice among a set of possible parses.

A proposed solution

So let's think about what `SetEncodings` might look like in this form:

```
func (e *SetEncodings) Format(s *Stream) {
        LiteralByte(s, 2)
        PadBytes(s, 1)

        n := U16Split(s, len(e.encodingTypes))

        if s.Parsing() {
                e.encodingTypes = make([]encodingType, n)
        }

        for i := 0; i < n; i++ {
                &e.encodingTypes[i].Format(s)
        }
}
```

When marshalling, the `LiteralByte` call emits the byte 0x2; the `PadBytes` call emits the byte 0x0; the `U16Split` call emits two bytes with a big-endian encoding of `len(e.encodingTypes)` (say, 0x00 0x03, if it's 3), and returns the number it just encoded; `Parsing` returns false; and then each of the three `Format` calls to the items in `e.encodingTypes` invokes an `S32` function to emit four bytes serializing that encoding type.

When unmarshalling, the `LiteralByte` call consumes a byte, and if it's not 0x2, it marks the `Stream` as failed, so that all the future calls on it (until possible backtracking) will be no-ops. If it was successful, though, `PadBytes` consumes and discards 1 byte, and `U16Split` ignores its second argument, decodes two input bytes, and returns the decoded value. Then `Parsing` returns true, so the format function allocates the slice, and then the iteration parses each encoding type in turn, by invoking its `Format` method.

I'm not familiar enough with Golang's type system yet to know if there is a better way to express this function:

```
func formatSliceU16(s *Stream, items *[]Formattable, make_item func() Formattable
o) {
        n := U16split(s, len(*items))

        if s.Parsing() {
                *items = make([]Formattable, n)
        }

        for i := 0; i < n; i++ {
                items[i] := make_item()
                items[i].Format(s)
        }
}
```

The difficulty here is that the items in the slice of interfaces (assuming `Formattable` is an interface!) needs a separate factory function

to instantiate them, since this function doesn't have any other way to invoke the proper `Format` for the particular type of `Formattable` the caller was hoping for. This is pretty bad compared to just having a slice of `int32` values with some nominal type; if you have 60 of them, you have 61 heap allocations totaling 1680 bytes (assuming interface values are three 64-bit pointers, and not counting the size of the slice itself) instead of 1 allocation of 240 bytes.

Anyway, this `formatSliceU16` function would reduce the above `Format` method to this:

```
func (e *SetEncodings) Format(s *Stream) {
        LiteralByte(s, 2)
        PadBytes(s, 1)

        formatSliceU16(s, &e.encodingTypes, func() Formattable { return &encoding
Type{} })
}
```

Using the same API, and allowing the various scalar functions to be variadic, the earlier-mentioned `PIXEL_FORMAT` structure can then serialize and deserialize as follows:

```
func (p *PIXEL_FORMAT) Format(s *Stream) {
        U8(s, &p.bits_per_pixel, &p.depth)
        BoolU8(s, &p.big_endian_flag, &p.true_color_flag)
        U16(s, &p.red_max, &p.green_max, &p.blue_max)
        U8(s, &p.red_shift, &p.green_shift, &p.blue_shift)
        PadBytes(s, 3)
}
```

The KeyEvent client-to-server message can be formatted as follows:

```
func (e *KeyEvent) Format(s *Stream) {
        LiteralByte(s, 4)
        U8(s, &e.downFlag)
        PadBytes(s, 2)
        U32(s, &e.keySym)
}
```

Backtracking — not sure if this is the right approach

Suppose we are receiving a message from the client which might be either a `KeyEvent` or a `SetEncodings` (or other possibilities we might add). We could imagine writing an `Any` function something like the following:

```
func Any(s *Stream, result *Formattable, fs ...Formattable) {
        if !s.Parsing() {
                result.Format(s)
                return
        }

        s.SaveBacktrackingPoint()
        defer s.DiscardBacktrackingPoint()
```

```
        for _, f := range fs {
                f.Format(s)
                if !s.Failed() {
                        *result = f
                        return
                }
                s.Backtrack()  // Preserves backtracking point
        }
}
```

And then we could call it with something like the following:

```
var ke KeyEvent, se SetEncodings, msg Formattable
switch Any(s, &msg, &ke, &se); msg.(type) {
```

I'm not sure exactly how that would work for output; ideally you'd like to be able to use that same code to *generate* a client message, leaving the parsed message in the same place when parsing that it would have found it when emitting, so that any subsequent conditionals or logic on what the actual message was will be unified between the parsing and emitting paths.

In the particular case of client-to-server messages in VNC, this bidirectionality probably isn't that useful, because the client can just as easily call `&KeyEvent{downFlag: true, keySym: key}.Format(s)` in the appropriate place as it can call `formatClientToServerMsg(s, &KeyEvent{downFlag: true, keySym: key})`. But in cases where the variant type is embedded down inside some other data structure, the simplification could be considerable.

Topics

- Programming (p. 3658) (286 notes)
- Parsing (p. 3618) (15 notes)
- Program design (p. 3654) (11 notes)
- Object-oriented programming (p. 3606) (10 notes)
- Golang (p. 3477) (7 notes)
- Serialization (p. 3707) (6 notes)

Kafka-like feeds for offline-first browser apps

Kragen Javier Sitaker, 2017-08-03 (5 minutes)

Suppose you want to build an offline-first browser app. A simple way to structure this is using a Kafka-like approach: all of the data in the app is stored in some set of append-only logs (or "feeds", as Secure Scuttlebutt calls them, or "topics", as Kafka calls them), each corresponding to a single writer (or "producer", as Kafka calls them, or "identity", as SSB calls them; for example, a particular profile in a particular browser on a particular user account on a particular computer). Synchronization of these logs is very simple, as long as there are never conflicting writes: for nodes A and B to synchronize log L, one tells the other the last entry they have in log L, and the other responds either by requesting the entries they don't have or by sending the other entries:

```
<A> my last entry in log L is 3258
<B> please send entries in L from 3201 to 3258
<A> entry 3201 in L is "joajgoiagjaesog"
<A> entry 3202 in L is "jogwj03280t02380"
...
<A> entry 3258 in L is "320820231di0w02"
```

or

```
<A> my last entry in log L is 3258
<B> entry 3259 in L is "302808gwahjg0saigj"
<B> entry 3260 in L is "]0ga0gjewagj0iew"
```

As long as there are never two machines creating conflicting entries in L, this protocol is simple, correct, and eventually consistent, regardless of the topology. SSB ensures this by requiring a public-key signature on each entry ("message") to prevent the propagation of unauthorized messages, and a previous-entry hash in each entry to prevent the log owner from propagating modifications to previously published messages, although they can still provoke desynchronization.

Nodes A and B might be a server and a browser, a browser and a server, or two browsers. As long as they are careful never to share information with anyone who isn't authorized to have it (SSB implements this by encrypting anything nonpublic, so that untrusted nodes can safely forward any message) it doesn't matter.

By itself, this provides, essentially, a group chat application. But an append-only data store can be used for any application.

For example, you could implement a centralized key-value store in a single log by appending (key, state, value) entries to it, where "state" is either "existing" or "deleted". The current state and value for a given key is just the most recent one in the log. This allows read-only slaves, and if you are sufficiently confident in your failover mechanisms, it could even allow for recovery after the loss of the master node.

If you want to allow multiple writers, though, you can achieve this with multiple logs, but you probably want to be able to at least detect lost-update conflicts; this requires expanding the entry tuple to ((key, parent), state, value), where "parent" is some ABA-problem-proof identifier of the previous state of the key, such as a secure hash of the entry that state was set in. If there are ever two entries with the same (key, parent), those entries are in conflict; the conflict must be resolved, through some application- specific mechanism. (In Git, this is done with a commit that has two parents; in Bitcoin, you instead use the block with the longest chain length from the root. The Git mechanism has the advantage that it records explicitly that the conflict has been resolved rather than forgotten.)

You could imagine a more sophisticated conflict-detection mechanism; for example, to commit a transaction, you could write some (key, transactionid, state, value) entries for the values modified, some (transactionid, entryid) entries for the entries that were read during the transaction, and finally a ("commit", transactionid) or ("rollback", transactionid) entry. Two or more committed transactions conflict if there does not exist an ordering in which none of them read versions of data that had been overwritten by a previous transaction.

A convenient way of structuring a key-value store program that uses this data store is to have it iterate over the entire history at startup time, constructing, say, an in-memory hash table of the latest value associated with each key — essentially replaying the history of the database. To reduce startup time, it could checkpoint a snapshot of the hash table along with the current entry numbers in each log; then, upon restart, it need only replay the entries since those offsets (as Kafka calls them). Indeed, it could store these snapshots in its own private log.

For some applications, it's reasonable to store the entire history of the application, either because the total volume of data is relatively small, or because the total amount of relevant data grows almost as fast as the entire history does. In other applications, it is necessary to forget old data because it takes up too much space. Kafka's approach is to, usually, store only the most recent data. This is probably the only approach compatible with the simple synchronization algorithm given above.

Topics

Measuring submicron displacements by pitch bending a slide guitar

Kragen Javier Sitaker, 2019-05-05 (18 minutes)

We should be able to get astoundingly precise positional and force measurements using the principles by which musical instruments are tuned, especially string instruments.

Genesis

A friend invited me to a work of musical theater tonight; despite the astoundingly adept dances, I was captivated by the designs of the improvised musical instruments — one being a Blue-Man-Group style PVC flapped pipe organ, where the players activate particular pipes by whacking their upper ends with heavy rubber flaps, and others being a sort of dulcimer or harp with six to ten strings each, in which one end of the string was anchored to the center of the bottom of a topless tin can, which can was screwed down to a wooden table, and the other end of the string was anchored to a hard object fastened to the table some distance away (I'd call it a "barrel", but in this context that word might be taken literally).

The thing that most captivated me was the extreme pitch bend the player would sometimes extract from the string by squeezing the can a little bit. Perhaps the rim of the can would lift by ten millimeters under this treatment, so the bottom where the string was anchored might be changing its natural position by two millimeters or so, but in all likelihood the can top was substantially more compliant than the string, so perhaps the string end was being displaced by half a millimeter or less out of the 500 millimeters or so of the string's length: a variation in length on the order of one part in a thousand. Nevertheless, the pitch bend was quite audible and even extreme, maybe more than a semitone.

Position sensing mechanisms

It occurred to me that this pitch-bending could be the foundation of very precise measurement techniques for measuring distance and thus size.

Audible pitch bending from string tension change

Electric guitar players commonly do pitch bends by shoving their guitar strings a centimeter or so to the side along the fret, thus lengthening the string. (Or they use the whammy bar, if they have one.) If the string is one meter in length, this would lengthen it to $\sqrt{(1\ \text{m}^2 + 1\ \text{cm}^2)} = 1.00005$ m, about 0.05 millimeters over a meter. Clearly the pitch bend is giving us a measurement of the string length that is sensitive to variations of some 50 parts per million, even better than I estimated in the paragraph above. (But maybe the strings connected to the cans weren't steel.)

How sensitive should we expect pitch bending to be to the position on the end of the string?

The Wikipedia article on the musical cent says that humans can directly hear pitch differences once they're larger than 5–25 cents, depending on musical training, on how high the pitch is, and on the harmonic content of the sound. A cent is a variation in frequency of about 578 parts per million, so the just-noticeable difference is on the order of 5000 to 20000 parts per million. The psychoacoustics article claims human frequency resolution is about 3.6 Hz in the 1–2 kHz octave, which is 1800 to 3600 ppm.

The Wikipedia article on classical guitar strings says and the Wikipedia vibrating-string article confirms and explains in more detail that the velocity of waves in strings is $\sqrt{(T/\mu)}$, where μ is the linear mass density and T is the tension. We can extrapolate that when the strings approach zero tension, the wave velocity (and thus the frequency) approaches zero, and when the tension varies by 1000 parts per million, the wave velocity varies by 500 parts per million. The frequency should vary by less than 0.5%, since the length of the string and its mass density are also changing, but the difference in those is much smaller, because the tension varies proportional to the difference from the string's natural length, while the length and mass density vary proportional to its difference from zero length, which is orders of magnitude larger.

So we should expect an audible pitch bend when the string *tension* changes by something like 5000 to 10000 parts per million, 0.5% to 1%. At most, for a steel string, the tension elongates the string by about 1% — after that, I think music wire will break, though most steels would deform plastically first — so you should always be able to hear an elongation of 2% of that, 0.02% of the total length, 200 parts per million. This is pretty close to the electric guitar number above.

But that's the *worst* case! We can do much better by putting the string under less tension. In theory, this should give us arbitrarily precise measurement of the string length, though only over correspondingly arbitrarily short distances. Indeed, I think this is one reason musical instruments are strung tightly, so that they won't go out of tune easily, and so that the string frequency doesn't rise for louder notes. In practice, I'm confident you can get one order of magnitude improvement: a length resolution of 20 parts per million.

Inaudible pitch bending and frequency counters

That, though, is assuming we're trying to detect the length by ear alone. Even a purely acoustic apparatus could improve on that: use two strings — a reference string of fixed length tuned to, say, 3000 Hz, while the measurement string is tuned to 3010 Hz at its default position. These will audibly beat at 10 Hz, more so if the second harmonic is attenuated (for example, by plucking or striking them in the center). It should be easy to hear differences in the beat frequency of 2 Hz or less, allowing an experimenter to hear variations in pitch of some 700 ppm, and thus variations in tension of 1400 ppm and in length of about 1.4 ppm.

But if we're thinking of an electronic measuring apparatus, rather than a purely acoustic one, we could straightforwardly just use a frequency counter to measure the frequency; these routinely have absolute accuracy of better than 1 ppm, and short-term precision even better than that. I think that would allow you to measure variations in length of about 0.001 ppm, 1 ppb.

At audio frequencies with guitar-sized 1-meter-long strings, 1 ppb is 1 nm, about 10 carbon atoms. If you use the same string under the same tension but only 100 mm of length, you get three octaves higher pitch (on the order of 10 kHz instead of 1 kHz) and resolution of 0.1 nm, about one carbon atom.

Electric guitar pickups

You can use an electric guitar pickup to detect very small movements of the string. Linearity isn't important, since the frequency is what we're interested in.

The slide guitar mechanism

The string-stretching mechanism described so far (call it the "whammy bar mechanism") has one big drawback: the apparatus is very large compared to the displacements being measured. So our hypothetical 100-mm-long, 0.1-nm-resolution sensor described above is only capable of making any measurement at all over the range of about a millimeter before breaking the string, and only about 300 μm with precision in the 0.1-nm range.

As an alternative, instead of altering the pitch of the string by stretching it, we could alter its pitch by sliding a "bottleneck" along it, as in slide-guitar playing. Only the length of the string up to the bottleneck vibrates, so its frequency gives us a proportional measurement of the bottleneck's position. This way, a meter-long sensor, for example, could read out a location anywhere within a 500-mm length while staying within a single octave.

The precision is correspondingly lower, but if you're using a 1-ppm-error frequency counter, you still get 1-micron resolution over a meter.

Trombone pipes

As an alternative to strings, you could use a column of air as your resonant medium, sliding one pipe inside another to continuously vary its length. The only advantage of this that occurs to me is that you can use any material at all. The precision and range should be comparable to the slide-guitar mechanism.

Delay line noise correlation and matched impedances

The above resonant mechanisms have certain problems with noise susceptibility and ringdown: it's quite reasonable to imagine that there might be vibrations in the environment within the range used by the instrument, and it would pick them up and could give erroneous readings as a result. Moreover, once they are resonating at some frequency, that vibration itself could bounce around inside solid bodies and be picked up even after the distance has changed — in effect, the instrument in the past produces its own interference in the future.

Instead of measuring a resonant frequency, though, you could generate random noise, or better still LFSR noise, and feed it into the measurement medium at one point and then read it back out — either at the same point after it's rebounded from the far end, or electronically at some other point, and measure the time lag with maximum correlation instead of a resonant frequency. In this case, the resonance of the medium is actually undesirable, and you can use the same matched-impedance technique used in electronic signal

transmission lines to prevent repeated reflection back and forth and the resulting resonance. That way, the signal you detect is a clean copy of a single lagged version of your input signal, not a sum of many past segments of the signal at different lags, progressively more attenuated.

A particularly interesting approach here is the slide-guitar approach, using non-contact sensing of vibrations in a wire as they travel past, for example using magnetic pickups, so that several sensors can share a single wire, which can end damped by a felt pad or something like that, beyond the sensors; or the trombone approach.

Free-air measurements

Once resonance is no longer needed, you might be able to dispense with the string or pipe and just transmit the ultrasonic noise signal through free air, permitting distance measurements at many points in space and thus triangulation. Stray reflections may give rise to multipath ghosting, but hopefully they can be kept manageable.

Sources of error

There are a variety of reasons that the temporal measurement from any of the three mechanisms discussed above might vary for reasons other than the displacements that we want to measure. We can try to eliminate these, or we can try to measure them and correct for them. For example, a free-air system should have at least one microphone a known, fixed distance away from the sound source, in order to correct for variations in the speed of sound in air.

Temperature

The natural length of the string will vary as it expands and contracts under the influence of temperature, potentially altering its tension, but in itself this need not introduce a large error — if the frame it's stretched on expands and contracts by the same proportion, its tension should remain constant. This may be difficult, since making the frame from music wire is probably not practical, and even if it is, the properties of music wire should vary by diameter.

The slide-guitar mechanism will, however, have a temperature-proportional error in position: if a movement causes the frequency to change by 0.1%, that represents a movement of 0.1% of its total length. But if it has expanded from 1000 mm to 1000.5 mm due to temperature, that 0.1% is now 0.10005% of its original length.

I think the whammy-bar mechanism will also have an analogous error: the Young's modulus of the string material will not remain constant with temperature, and indeed I think should have roughly the same thermal coefficient as the material's natural length.

Steel's coefficient of thermal expansion is about 10.8 ppm/° around room temperature, so we're easily looking at a 100 ppm error here if temperature is not controlled.

The trombone mechanism, however, suffers greatly from temperature drift, since the speed of sound in gases varies as the square root of the temperature. A variation of 2% in the temperature (5.5°) thus changes the tuning or time lag of the tube by 1%.

A potentially much bigger problem for the whammy-bar mechanism is that, when the surrounding temperature changes, the string will reach the new temperature much sooner than the frame

will; and if a human handles the device, they will warm it up where they touch it. If the string is steel 10° warmer than the steel frame, it will be ≈108 ppm longer, but if its strain was only 1000 ppm, that's an error of 11%, 110'000 ppm, in the strain.

The use of Invar or some similar material might be worthwhile if the apparatus cannot be protected from such variations in temperature.

Humidity and pressure

Solid materials should be pretty immune to pressure (at least until the mass of the air around the string becomes significant compared to that of the string — steel weighs about 8000 times as much as the same volume of air, and air's density varies only proportional to pressure, so this should be a source of uncontrolled variation of frequency in the 1 ppm range). Steel is pretty immune to humidity, too, but other possible string materials might be hygroscopic.

The trombone mechanism, including the free-air version, suffers the most here: although to first order the speed of sound in air doesn't vary with pressure, it does vary with humidity; since water replaces some air molecule with water molecules of roughly half the weight, it speeds up sound transmission by up to about half a percent, introducing a half-percent error (5000 ppm) in the distance measurement.

Creep

Creep can reduce tension over time if a string is under constant tension. Steel doesn't creep much at room temperature, which is how pianos can stay in tune for months or years at a time, but other possible string materials (and, even more so, frame materials) might creep rather badly. And it might be that these mechanisms are sensitive enough to detect creep phenomena in steel that usually go undetected.

Latency

It's desirable for position transducers to respond as rapidly as possible to help keep control loops stable.

The resonant approaches suffer from the need for vibrations to build up over potentially several round trips; the noise-correlation approach avoids this, but still requires at least one round trip if the microphone is colocated with the speaker. Sound in air travels at only 343 m/s, so measuring a distance of a meter is going to take about 3 ms. Music wire can transmit vibrations faster but you're still potentially looking at milliseconds of latency if you wait for the reflection.

However, with the noise-correlation approach, if you locate the sensor at the place whose position you're measuring, you don't suffer this latency (except in the whammy-bar mechanism, which inherently averages the propagation time over the time the signal is traveling through the string). You just need enough signal to correlate so that you don't get misled by noise. So you could have a feedback latency measured in microseconds (assuming the vibration frequencies are sub-MHz) instead of milliseconds.

Measuring force instead of displacement

The wire we're talking about here is elastic, which is how sound waves can travel over it in the first place. Up to now, the force needed to stretch it has been nothing more than a nuisance — hopefully we can use a thin enough wire that it doesn't exert too much force and disturb the thing we're trying to measure. But what if we use that very elasticity? Instead of stretching the string on a frame, hang an unknown mass from it and weigh it by way of measuring the sound propagation speed on the string. It's like the whammy-bar mechanism, but now our guitar has no neck.

Again, we should expect to be able to measure the lag with an error of about 1 ppm, translating to 2 ppm error in the tension, with (I'm guessing) 11 ppm/° of thermal error.

Topics

A minimal dependency processing system

Kragen Javier Sitaker, 2017-09-21 (3 minutes)

(See also Minimal transaction system (p. 2460).)

This is a miniature operating system in which computations are executed in response to changes in the filesystem, and which in turn can create more changes in the filesystem.

Transactions run. How they were started, I do not know; but they do run. A transaction looks at some part of the filesystem, then creates more files, then exits successfully, making its output files available to other transactions. The input files and dirents that went into creating them are automatically recorded; if any of them change, the transaction is re-executed, updating the files.

When a file is created, its creator specifies an invalidation policy. With the default invalidation policy, the invalid version of the file remains available while the transaction is being re-executed, and if the transaction doesn't change the file's contents, then transactions that depend on that file won't be re-executed either. There's also a "strict" invalidation policy which immediately invalidates any transactions that accessed the file, re-executing them afterwards. Attempts from outside of transactions to access a strict file that's being recomputed will block or give an error, I don't know.

(I'm not sure whether I need the same thing on read edges.)

A pure memoization variant

The fundamental operation is to apply a function to an argument, producing, eventually, a result. The argument and the result are namespaces — filesystem directories, basically; the function is a blob, a program. The system monitors which parts of the argument are accessed by the function and what system resources it needed and caches the production of the result on that basis.

On Linux, the minimal grain size for process execution is on the order of a millisecond; a fork/exit/wait loop takes about 130 µs per iteration at best on my laptop, up to a few milliseconds with large memory maps or on slower processors. If a function is going to take much longer than a millisecond, it should farm out the work to subfunctions as much as possible, enabling both caching and distribution. We might be able to do better and get down into the deep submillisecond range.

130 µs is about 200 base cases of `fib = lambda n: 1 if n < 2 else fib(n-1) + fib(n-2)` in Python on the same machine.

How much data are we talking about caching? `yes` can feed data to `dd` at about 800 MB/s, or 800 kB/ms. `seq` can generate about 128 MB of numbers per second (128 kB/ms) and `gzip -9` compresses them by about 4× at about 1.8 MB/s (1.8 kB/ms) output, or 7.2 MB/s (7.2 kB/ms) input. So the individual output files we're caching could reasonably be from about a kilobyte up to about a megabyte, but of course larger results will contain many such files together.

Topics

Top algorithms

Kragen Javier Sitaker, 2018-07-29 (4 minutes)

A compilation of "top" algorithms from different sources.

First, the algorithms that appeared on multiple independent lists:

- The FFT 0 2 4 6
- The simplex method for linear programming 0 2 6
- The QR algorithm and related decompositions (LU, SVD, etc.) for computing eigenvalues and least-squares solutions 0 2 4
- Monte Carlo methods and the Metropolis algorithm 0 2
- Mergesort 3 5
- Krylov subspace iteration methods such as conjugate-gradient and Lanczos 0 2
- Quicksort 0 6
- Hashing and hash tables 3 6
- Dijkstra's algorithm 4 6
- PageRank and related link-analysis algorithms 1 6
- Huffman coding and other data compression 4 6
- Newton and quasi-Newton methods 1 4
- Binary search 3 6
- RSA 4 6
- Heapsort 4 6
- Diffie-Hellman key exchange 4
- The quadratic sieve and other integer factorization algorithms 4
- The Ford-Fulkerson algorithm for maximum flow 4
- The decompositional approach to matrix computations 0
- The Fortran I optimizing compiler 0
- fast multipole methods (for e.g. N-body simulation) 0
- the Kalman filter 1
- Dynamic programming 4
- Gradient descent 4
- A* search 4
- PID control 5
- RANSAC 4
- Q-learning 4
- Gaussian elimination 2
- the Viterbi algorithm 4

Here are the ones that I think appeared on only one list:

- Integer relation detection 0
- JPEG 1
- least-squares fitting 2
- Gauss quadrature for numerical integration 2
- Adams formulae for ODEs 2
- Runge-Kutta formulae for ODEs 2
- finite differences for PDEs 2
- floating-point arithmetic 2
- splines (including Bézier, de Boor, and others) 2
- stiff ODE solvers 2
- the finite element method for PDEs 2
- orthogonal linear algebra (Givens rotations, etc.) (maybe this should

be part of the QR item?) 2
• preconditioning of linear systems 2
• spectral methods for PDEs 2
• MATLAB 2
• multigrid methods for PDEs 2
• IEEE floating-point arithmetic 2
• nonsymmetric Krylov iterations 2
• interior point methods for optimization 2
• wavelets 2
• automatic differentiation 2
• Rabin-Karp string matching (e.g. for plagiarism detection) 6
• the Gayle-Shapely algorithm for the Stable Marriage problem 6
• Bloom filters for e.g. malicious site filtering 6
• LALR parsing 6
• Red-black trees 6
• Bresenham's algorithms for drawing circles and lines 6
• Kruskal's algorithm 6
• Depth-first search 6
• Breadth-first search 6
• Convnets 6
• the forward algorithm for hidden Markov models 6
• Raycasting 6
• Knuth-Morris-Pratt string search 6
• Radix sort 6
• Markov-chain Monte Carlo/particle filters 6
• Shor's algorithm 6
• prune-and-branch tree search 6
• Beam search 4
• Branch and bound (is this the same as "interior point methods"?) 4
• Buchberger's algorithm (generalization of Euclid's) 4
• Discrete differentiation 4
• Euclid's algorithm 4
• The expectation-maximization algorithm (EM-training) 4
• Binary heaps 4
• Karatsuba multiplication 4
• Lenstra-Lenstra-Lovasz (LLL) lattice reduction 4
• The Schönhage-Strassen algorithm 4
• Strukturtensor 4
• Union-find 4
• SHA-1 and SHA-2 5
• PRNGs 5

Topics

Processing halftoning

Kragen Javier Sitaker, 2019-09-01 (15 minutes)

Lots of images in the modern world are overlaid with some kind of regular screen pattern. Halftoned images from newspapers, posters, and magazines; images printed on woven cloth or art paper with cloth imprinted on it; images photographed through window screens, security grilles, or microwave oven doors; signs printed on cloth-backed vinyl, illuminated from behind; GIFs using patterned dither; woodcut and metal engraving images; bus-window advertising, with its pattern of regular holes; shadow-mask and Trinitron patterns from CRTs; pixel-grid patterns from LCD matrices, especially low-resolution ones; silkscreened images with incomplete coverage; objects seen through mostly-opaque cloth; patterns of illumination on woven and nonwoven cloth and legs wearing fishnet stockings. Additionally, there are one-dimensional equivalents — PWM signals, audio distorted by clipping, and AM radio.

I suspect that these patterns are amenable to a number of interesting image-processing and other signal-processing algorithms.

Dehalftoning

Traditional newsprint monochrome halftone

A classic newspaper-article monochrome halftone image consists of a square lattice of black dots on a white background, rotated to some odd angle to minimize aliasing artifacts, whose varying diameters give varying "tones", which is to say brightnesses, to different areas of the image. In blacker areas, the dots lose their roundness and start to merge together, leaving isolated white dots, which start to become round.

In the frequency domain, we can analyze a small area of the image as containing some DC level, plus "dithering noise" added by the halftoning, which is entirely concentrated in the even harmonics of two spatial frequencies, one 90° rotated from the other but otherwise identical — the frequencies of the warp and weft of the halftone screen. If we look at different small areas of the image, the amplitudes (but not the phases) of these harmonics vary (in varying, non-monotonic ways) with the DC level; looking at a larger area of the image, this amplitude modulation manifests as sidebands on the "carrier signal".

The underlying process is not simply AM; the carrier signal is *added to* the underlying image gray level, and then the sum is thresholded (originally, through nonlinear chemical processing). The result has a maximum carrier amplitude at 50% gray, and zero carrier amplitude at either pure white or pure black; the second harmonic of the carrier is zero at pure white, pure black, and 50% gray, while reaching amplitude maxima at 25% and 75%, I think; and higher harmonics vary in ampitude more rapidly.

XXX even in one dimension, the Nth harmonic has N maxima and N+1 nulls; the odd harmonics are not missing (except at 50%!) so the below algorithm is wrong (and probably something simpler is

doable)

Linearly, shift-invariantly dehalftoning the newspaper

Given the carrier frequency and angle, it is fairly straightforward to construct a zero-phase linear filter kernel that sharply notches out the carrier frequency itself and precisely its even harmonics, leaving just the information-carrying sidebands, using filter inversion. The filter for the even harmonics is a ("feedback") comb filter; spatially it looks like a square grid of points (positive impulses) at the same angle as the halftone screen and twice its frequency; the filter for the fundamental looks like the heights of egg-carton foam, also making a square grid at the same angle, but without impulses or sharp edges; and inverting either of those turns it negative and adds a new, much stronger positive impulse in the center.

The convolution of these two notch filters provides a filter that eliminates the halftone screen entirely. However, without any spatial windowing, they only remove the part without any spatial variation whatsoever; you need to window them down to a spatially reasonable area, for example with a Gaussian window, in order to remove the harmonics that are prevalent in a given area.

This filter might still seem to have alarmingly large support for practical spatial-domain convolution, but due to its special structure, the multiplication-free sparse-kernel-cascade techniques described in Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) can compute convolutions with it very inexpensively.

There's one problem left over: DC is an even harmonic of the halftone-screen frequency — the zeroth harmonic — so the even-harmonic filter removes the DC signal we wanted to save, and we're left with only edges detected. To solve this problem it is necessary to high-pass-filter the even-harmonic-selecting kernel so that its DC response is zero, or at least less than ½ or so, ideally without affecting its relative response to the other harmonics. This is clearly doable by, for example, bidirectionally filtering the signal the even-harmonic filter detected with a high-pass Chebyshev filter before subtracting it from the original image pixel to invert it; but it's not immediately evident to me if there's a way to do it with the sparse multiplication-free approach.

The frequency-selectivity of this approach should give much better high-frequency response than the usual approach of just blurring the image — that is to say, it should preserve edges that are close to the frequency of the halftoning screen itself, and even those that are higher, assuming the halftoning approach is the traditional photographic analog approach, from which the dots can come out off-center and non-round.

Minor nonlinear enhancement

If the resolution is high enough, the intendedly-bilevel nature of the halftoned image can be used to nonlinearly correct some other errors: uneven illumination, uneven paper color, and uneven ink blackness can all be corrected by contrast-stretching the image with locally-varying black and white levels; a bit of morphological opening and closing might help due to the intendedly-contiguous nature of the halftone dots. Doing this for CMYK images is more complex, due to the 16 possible combinations of ink colors, but is

certainly possible with high enough resolution.

Some kind of nonlinear preprocessing is probably needed for CMYK images, because each color channel is normally printed with the halftoning screen at a different angle in order to reduce aliasing artifacts, and the inks interact nonlinearly (though I conjecture that in a negative image they might be closer to linearity). So you need to recover some kind of per-color-channel estimate before you go trying to dehalftone.

Linearly, shift-invariantly dehalftoning other images

The other kinds of images I mentioned above — backlit images printed on cloth-backed vinyl, images printed on Lambertian cloth, images with CRT shadow-mask patterns, and so on — mostly don't have the pattern of modulation of harmonics that characterizes the kind of screen halftoning used by newspapers and magazines. Instead, they have a grid pattern whose amplitude is multiplied by the image, pixel by pixel, without modulating the spatial spectrum of the grid pattern (unless there's sensor saturation, say). Nevertheless, the same linear filter should work.

This process is a much purer spatial equivalent of amplitude modulation as used in radio, although the carrier signal is not sinusoidal, and the baseband signal is still present; in general it consists of all the harmonics of two spatial frequencies, which is to say, the lattice generated by two vectors in frequency space.

Morphologically, shift-invariantly dehalftoning those images

In these cases, it ought to be possible to morphologically open or close the image in order to reduce the halftoning artifacts — depending on whether the artifacts are dark, as they usually are, or light, as in a photo taken through a window screen. This approach won't work (or will work very poorly) with newspaper halftoning, because the image data isn't encoded in the brightness or color of what is seen through the screen, but of the size of the "apertures". But for linearly-encoded data like this, it ought to work well.

This is closely analogous to how a crystal radio set demodulates AM radio, though without a tuner: the nonlinearity of the diode detector spreads the peak of the detected wave across the valleys in between.

Chroma subsampling

CRT and (low-resolution) LCD images are somewhat special among linearly-encoded halftoned images in that they have, effectively, color filters — the red, green, blue, and sometimes white subpixels are each limited in what colors they can carry. Nevertheless, in CRTs and in LCDs using subpixel rendering, they are used to carry both chroma and brightness information, relying on the lower resolution accorded to chroma information by the eyes of the humans.

Probably the best solution here is to do the dehalftoning process entirely in brightness-space, then add blurred chroma information to it afterwards.

Non-shift-invariant dehalftoning

A potentially more interesting approach is to try to identify which pixels depend on the "underlying image" and which are just part of the halftone screen and should be filled in based on some kind of filling operation from the underlying image (whether via mathematical morphology or some kind of more elaborate texture-synthesis algorithm). This is likely to work a lot better for, for example, photos taken through security grilles or microwave oven doors, than the algorithms above.

Low-resolution images are likely to have a lot of mixed pixels, so the model needs to handle partly-screen pixels.

The simplest kind of model here is one that estimates the phase, angle, and perhaps waveform of the screen; then multiplies the image by it pixel-by-pixel to selectively amplify the underlying image; then blurs that image or comb-filters out the screen frequency, as before, to estimate the underlying image. This is precisely analogous to an AM radio receiver mixing with a local oscillator.

Video or stereo data is probably very useful for these models, since correlations over time or over viewpoints will tend to separate the underlying image from the screen.

Optimization-based approaches

Given a computation that generates a halftoned image from an underlying image and other parameters such as halftone-screen angle and phase, an error metric accounting for the likelihood of things like sensor noise and ink squeeze, and a prior probability distribution over underlying images, you can use a generic high-dimensional optimization algorithm such as the popular variants of gradient descent to infer likely underlying images.

(See Image approximation (p. 2394) for more explorations along these lines.)

Image separation by demodulation

Consider a photo taken through a microwave-oven door. Most of the image consists of a reflection of the room behind the camera, but through the holes in the RF shielding grille we can see the scene inside the microwave.

As I said above, the inside scene is an amplitude-modulated signal, and we can recover it in the usual way, by mixing with a local oscillator or morphologically "rectifying" it, and then filtering out the carrier. However, another alternative is to filter out the carrier *and the sidebands*, leaving only the reflection of the room.

An improved estimate of either the baseband non-modulated image of the room or of the interior, amplitude-modulated image, allows us to do a better job of canceling its contribution to the combined image, getting a better estimate of the other image. In this way we can, to a significant extent, separate the original combined image into its constituent layers.

(The same approach can be used, of course, with temporally-modulated illumination; in a loose sense this is the basis of structured-light 3-D scanning.)

Canceling reflections of amplitude-modulated textures

As another example of the same phenomenon, on the bus home, there was an illuminated sign indicating which branch of the line the

bus was serving, made of fabric-backed vinyl and illuminated from within. This sign was just inside the windshield, near the bottom, so it created a strong reflection on the windshield, obscuring the image of the road.

With this approach, it should be possible to subtract the cloth-modulated reflection and see the road — if JPEG compression doesn't get to your image first and erase the subliminal road data.

Detecting amplitude-modulated textures in superimposed images

Consider instead looking out the window of a café where the sun is falling on a table. The newspaper on the table, lit by the sun, is reflected in the window; its reflection is superimposed on the image of the tree outside the window. Can you recover the newspaper front-page photo from a high-resolution combined image?

Newspaper halftoning has the solarization ambiguity I mentioned earlier — the spatial frequency spectra are symmetric around 50% gray. But, in theory, even the fundamental frequency and its second harmonic are sufficient to recover the gray level up to that ambiguity — but only if you know beforehand how strong they would be at their strongest. The third and fourth harmonics help further, because their own minima allow us to precisely calibrate the strength of the first and second harmonics.

(I say "minima" because the nulls are far sharper than the maxima, being as they are the crossing of a sinusoid with the X-axis, while the maxima are the peaks of those sinusoids.)

A similar problem is removing glare from a photograph of a glossy magazine. If the sensor didn't saturate and the photo didn't suffer lossy compression, the halftone-frequency information in the glare area contains most of the information, again up to the solarization ambiguity.

All of these presuppose that you can perspective-correct the image and obtain a reasonably correct estimate of the halftoning frequencies and angles, of course.

For mesh sizes below about 200 μm, diffraction starts to become a problem, and geometric-optics-based algorithms become increasingly inaccurate. The Airy radius $\sin\theta = 1.22\lambda/D$ of a 100-mm mirror at 555 nm is 6.7 μradians, which means that from more than about 30 meters away, a 100-mm mirror or lens can't image through a 200-μm hole; the mirror itself is visible, subtending about 0.2°, half the apparent size of the sun or moon. More-distant sensors would need to be proportionally larger. So you don't need to worry about drones too far away for you to see using these algorithms to see through your bedroom curtains.

Topics

• Sensors (p. 3706) (12 notes)

Bottle washing

Kragen Javier Sitaker, 2014-04-24 (7 minutes)

Suppose you're washing out a bottle. You want to achieve the maximum dilution of the contaminants in the bottle for a given amount of water; a miser might phrase this as minimizing the amount of water needed to get the needed dilution of the contaminants.

If we abstract a little, we're repeating this process some number of times:

• dumping all the water out of the bottle except for the little bit that won't come loose, which contains all the remaining contaminants; call that remaining water volume V0;
• adding more water to dilute the water already in the bottle, up to some volume V1, diluting the contaminants by a factor V1/V0 by using an amount of water V1-V0, which we will then dump out when we repeat step 1.

After cycling some quantity V2 of water through this process, we've repeated it V2/(V1-V0) times, more or less, and achieved a dilution of (V1/V0)^(V2/(V1-V0)) times.

You might think that you don't want to go to extremes here. If V0 is one milliliter, and V1 is two milliliters, then on each cycle, you reduce the contamination by a factor of 3; so if you use twenty milliliters, you've reduced it by 3^{10}, almost sixty thousand. By contrast, if you dump all twenty milliliters in at once, you achieve only a dilution of 1/21, which is pathetic; and if you go toward the other extreme and use one microliter of water on each rinse cycle, you go through twenty thousand rinse cycles, each providing a 1001/1000 dilution, which you might think wouldn't make much difference. But it turns out that that adds up to a dilution factor of 480 340 920 times. (You need the first thousand rinse cycles just to get to a dilution of roughly e, 2.717, but every thousand rinse cycles then adds up to another factor of e, and e^{20} is almost five hundred million.)

That's about as good as you can do, though: rinsing with 20 times the amount of residual water that you can't get out will give you a dilution of up to e^{20}. And you can reach almost that optimum, three hundred million, say, with only 400 rinses, each of 50 microliters. At only 20 times the residual water, you don't get much additional benefit from reducing the amount of water on each rinse cycle below a twentieth or so of the residual water. However, because it's an exponential process, small differences in dilution will grow into larger ones as you repeat the process.

Which is to say that, perhaps unsurprisingly, a more or less continuous rinse is the ideal; you want to remove rinse water as fast as it's added. Allowing any amount of rinse water to accumulate that's significant compared to the residual, unremovable water will dramatically reduce the dilution you achieve. Once your effective number of rinses drops to 50, in the scenario above, you've lost more than an order of magnitude of dilution over 20 rinses.

Of course, if you have a way to reduce the residual water, that will be even more effective. Reducing the residual water is equivalent to

increasing the rinse water by the same factor, which increases the exponent on e; in the above example, reducing the residual water by 50 microliters pumps up the optimum from e^{20} to e^{21} (plus a bit), improving your final dilution by a factor of 2.7!

(This exponential sensitivity of the final result to the precise dilution ratio suggests use for sensitive measuring equipment. If you have a 1%-precision measure of the final concentration, say, you can derive an 0.05%-precision measure of the geometric-mean ratio of the ratio of the residual and rinse water in each step, or more if you repeat the dilution procedure more times.)

On the other hand, since it's an exponential process, the absolute magnitude of the problem is not so terrible.

The ideal is to achieve dilutions in the mole range (10^{24}), such that even if mole quantities of the contaminants were originally present, no molecules of them remain afterwards. (Succussion has not been shown to help.) The minimum to reach a dilution of 10^{24}, with the e^{20} limit above, is when $20 \approx 55.3$; that is, you need 55.3 times as much rinse water as the residual water. If for some reason you're stuck in a low-efficiency regime where you can't avoid rinse-water accumulation, such that each rinse cycle only dilutes contaminants by a factor of 2, you need 80 rinse cycles, which means you need 80 times as much rinse water as residual water. At 55 rinse cycles, you will have only reached a dilution of about 3×10^{16}, thirty million times more concentrated than with the optimal strategy.

That is, in more or less the worst case, your non-optimal rinsing strategy costs you only about 45% more water, even though when using the same amount of water, it's thirty million times worse.

(It's kind of amazing that it's feasible to achieve such extreme dilutions at all, let alone achieve them in your kitchen. Of course, there are always tricky details to get right, like that congealed grease spot full of PCBs that isn't getting diluted by your water. Don't get cocky.)

If your desire is not to remove every single molecule of whatever contaminants you have, but simply to reduce the pH of your laundry to the point where it won't deteriorate while drying or in storage due to the alkali, you have a much easier task. I'm going to guess that the pH of alkali laundry systems rarely goes above 10 and almost never goes above 11, and that reducing the pH to 8 before drying the laundry will provide adequate protection; that's a dilution of only 10^3, which you can achieve with 10 rinse cycles in what I described as "more or less the worst case" and 7 "rinse cycles" in the best case of continuous mixing.

Welp, guess it's time to rinse this laundry.

Topics

Solving the incentive problem for censorship-resistant DHTs

Kragen Javier Sitaker, 2016-09-07 (updated 2019-05-21) (3 minutes)

I was thinking about the incentive problem in DHTs.

DHTs sort of inherently mean you can't choose which node will host your data, right? You can choose which network, but if the nodes that your key hashes to are unfriendly, they can forget you ever tried to store data on them; and so there's no way to set up BitTorrent-like reciprocity, where other DHT nodes have an incentive to store my data so that I'll be more willing to store their data.

I'm wondering if there's a way to solve that problem without completely losing the desirable aspects of DHTs. In particular, node load in a DHT scales sublinearly with the system size, assuming churn is low (which is an incentive problem!), node count scales with data size, and update rate doesn't change. That would fit in, say, 64 bytes. But nodes might discard your data item because it's deemed to be an anti-Islamic data item, for example, if the DHT node happens to be hosted in Saudi Arabia, or a heretical data item if it happens to be hosted in a Christian country.

The data that is thus incentive-protected need not be large, because the data items can be very small and still be useful. For example, you could store a (key, favorableserver) pair that redirects querents to a server that you do have some kind of ongoing mutual relationship with, such as ownership, payment, or reciprocal storage.

One possibility is that there's some way to ensure that the DHT node can't erase *your* data with certainty without erasing *everyone's* data with some probability, which converts the incentive from a relationship with only you to a relationship with the entire network. (This is the motivation behind Verónica Estrada's helical entanglement work.) Like, if the node couldn't tell which data was yours.

Typically, DHTs already do store your data on multiple DHT nodes to provide some resilience in the face of node churn; even Karger's original "Consistent Hashing" paper from 1997 that Akamai was founded on explains that you can hash the URL to several different points on the circle, which will probably map to different servers; but this doesn't really solve the incentive problem.

As another way of thinking about the incentive problem, consider the case of expiry times. You have a DHT that typically maintains object validity fo ra day. But you want to store data in it that will be consulted perhaps once a week. If your publisher is reliable, it will re-store the data in the DHT nodes about every 12 hours, which is to say about 14 times as often as it's consulted. It would be a lot more efficient on an overall basis to persuade the DHT to store your data for a week or ten instead of a day, and also be more resistant to you being arrested for publishing un-Islamic material.

DHTs suffer badly from churn, and churn in practice is pretty high, which is arguably another aspect of the same incentive problem — how do you incentivize people to keep their nodes running?

Topics

Patterns for failure-free, bounded-space, and bounded-time programming

Kragen Javier Sitaker, 2018-04-27 (updated 2019-09-10) (42 minutes)

Often, the most convenient way to program a piece of software is to use garbage collection, recursion, and the Lisp object-graph memory model they were born in, often along with closures and dynamic typing. But these approaches have their drawbacks: almost any part of your program can fail or require unbounded time to execute. Sometimes it is useful to write programs that will not fail, even on a computer with finite memory, and will make progress in bounded time. For example, if your user interface stops making progress, your computer becomes unusable, even if the rest of its software is working fine (see Real time windowing (p. 891)); and, if the program on a microcontroller fails, there is often no sensible way for it to report failure, and physical processes it was controlling may run out of control, perhaps damaging equipment or killing some of the humans.

The basic incompatibilities are the following:

• In the object-graph memory model, heap allocation happens often, usually implicitly (although not in early versions of Java), can always fail, and can almost always take an unbounded amount of time.
• Closures in particular — at least, the way they are normally implemented — give rise to implicit heap allocation.
• Recursive function calls can potentially use both unbounded space (for the stack) and unbounded time. If you have no recursion and no closures, you don't strictly need a stack; it's purely an optimization.
• With dynamic typing, any primitive operation — and, in OO languages, any method call — can fail due to a type error.

Here I will discuss approaches that can be used to write programs that can execute in bounded time, bounded space, and without the possibility of error conditions arising.

These approaches are usually used in contexts where system resources are very limited, and so they are usually used in conjunction with a lot of optimization, which can reduce both the average-case resource use and the worst-case resource use of the program. However, they are conceptually distinct from optimization, even though they may be confused with it.

Static checking, including type checking

The most general technique is to check invariants before the program is run. An invariant that is (correctly) verified to hold by static reasoning cannot be violated give rise to a run-time error. For example, object-oriented programs in languages such as OCaml are guaranteed not to compile a method call on an object that might not support that method. This is almost true in C++, but C++ has enough undefined behavior that it is in practice impossible to make

any compile-time assertions about program behavior.

Such static checking can happen after compile time as well as before. For example, for TinyOS, a stack depth checker was developed that statically verifies the maximum stack depth of a machine-code program. (See also Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) for more on how to do this.)

This generalizes to proving arbitrary properties of programs, as was done for CompCert and seL4.

Pointer-reversal tree traversal

Lisp systems traditionally used a mark-and-sweep garbage collector, which first *marks* all the nodes in memory that are accessible from "roots" (such as the stack or global variables), then *sweeps* through memory to find any unmarked objects, adding them to the free list. A simple implementation of the mark phase that handles only conses might look something like this:

```
(defun gc-mark (node)
  (when (and (consp node)
             (not (eq (mark-field node) *collection-number*)))
    (setf (mark-field node) *collection-number*)
    (gc-mark (car node))
    (gc-mark (cdr node))))
```

This is simple enough, but it sweeps a critical issue under the carpet: those recursive calls to `mark` eat up stack space. How much stack space do they need? Well, it can be as much as one level of stack for every live cons, if they're all in a single list! Normally traversing a tree through recursive calls like this is a reasonable thing to do, but this function is being invoked because the interpreter ran out of memory, except what it needs for the stack, and needs to free some up. So statically bounding the stack usage, as mentioned in the previous item, would be really super useful.

You might think we could rewrite it into a non-recursive loop:

```
(defun gc-mark (node)
  (loop while (and (consp node)
                   (not (eq (mark-field node) *collection-number*)))
    do (setf (mark-field node) *collection-number*)
    do (gc-mark (car node))
    do (setf node (cdr node))))
```

That way, if we have one huge long list `(a b c d ... zzz)`, we don't eat up our stack recursing down it; each cons of the list gets processed in the same stack frame as the previous one. But we still have a recursive loop which can still eat up space bounded only by the number of conses — it's just that now it has to look like `(((((...zzz...)))))` instead.

The fundamental problem is that every time we encounter a new cons, we encounter not one but two new pointers to follow, and so whichever path we choose to take, the number of paths not traveled by can always keep growing, one new path for each cons node we traverse.

If we could only leave some sort of breadcrumbs in those conses in order to avoid needing to allocate that data on the stack! Too bad they're already full of essential data.

Consider this implementation of NREVERSE for lists:

```
(defun nreverse (list)
  (loop with next with reversed = nil
     if (null list) return reversed
     do (progn
          (setf next (cdr list))
          (setf (cdr list) reversed)
          (setf reversed list)
          (setf list next))))
```

It isn't recursive and doesn't allocate any memory other than its three local variables. Nevertheless, `(nreverse (nreverse list))` will traverse the same list nodes twice, once forwards and once backwards, and it leaves them in the same state as before. By reversing the pointers, it maintains the state it needs to traverse the list again in reverse order.

This implementation of NREVERSE is a simple case of pointer-reversal tree traversal. Where regular tree traversal maintains a single pointer to the current node, we maintain two pointers: the current node and the previous node. (While executing a movement along the tree, we have an additional temporary pointer, called `next` above.) The previous node used to have a pointer to the current node, but no longer does; it now points to its own previous node instead. It happens that the backward traversal in this case is precisely the same as the forward traversal, but that is not the case in general.

To walk the whole tree, sometimes you'll be descending down the left branch (the car) and sometimes down the right branch (the cdr). That means that sometimes the previous node will have its car reversed and pointing to its parent, in which case when we return back up to it we want to descend down the cdr branch instead, and sometimes it will have its cdr reversed and pointing to its parent, in which case when we return back up to it we want to keep on returning back up to its parent. To distinguish these two cases, we need to store that one bit of per-node state somewhere, just like the mark bit, and typically we use a bit somewhere in the node itself, though there are other options. If you have three or four children on a node, you need two bits instead of one, and so on. When you finish walking the tree, you have undone all your mutations.

And that's Deutsch–Schorr–Waite tree traversal, which needs only a bit per node in the worst case instead of a word per node.

Deutsch–Schorr–Waite tree traversal is designed as a failure-free algorithm because, like most algorithms, garbage collection can only fail by running out of memory; but, in the GC case, that failure is an overwhelmingly likely outcome if you don't make it impossible.

You can use it for things other than garbage collection. You can implement a binary-tree search function using Deutsch–Schorr–Waite traversal, for example; you just choose which child to recurse down by comparing the key, and when you're traversing back up the tree, you always just go up to the parent, rather than sometimes traversing down to another child, as you do for

garbage collection. More interestingly, you can implement red-black tree insertion this way.

Using it for DAG traversal may be hairier; GC uses the mark bit to avoid endlessly revisiting the same nodes, but other traversal algorithms may not be able to.

An interesting thing about this traversal is that it's achieved by using mutation instead of the usual side-effect-free algorithms to traverse the tree, because the alternative to storing the breadcrumbs with mutation is to allocate memory for them, and that introduces failure. (See The Z-machine memory model (p. 2903) for some notes on a memory model that's all about tree mutation and was at one point used by a significant number of hackers.)

Of course, such a mutating tree traversal would be a disaster if it could be interrupted in the middle, for example, by an out-of-memory exception or a program interruption from the keyboard. But since what we're exploring here is precisely how to eliminate such exceptions from our system, we are justified in taking advantage of the benefits thus gained.

Note that, in the above loop, the number of pointers to a given node remains constant, except for the ephemeral copy in `next`. I think this is a necessary property of all such traversal algorithms on trees, which is to say, data structures with only a single pointer to each node; my reasoning is as follows.

If the reference count of such a linearly-referenced node drops, it drops to zero, and the node leaks. If you have a garbage collector, it may recover the node later, and you may be able to avoid turning a memory leak into an actual failure, but for reasons explained in the introduction, I don't think garbage collection is likely to be compatible with failure-free computing.

For the reference count of such a linearly-referenced node to increase, either it must be overwriting some other reference and decreasing its reference count, or it must be allocating new memory; neither of these is compatible with failure-free computing.

(The potential hole in this reasoning is that it's legitimate for the traversal algorithm to have some finite number of local variables like `next` above that potentially alias pointers found in nodes on the heap, but which may be the only access path to a node at some time. I'm not sure this is ever essential but I don't have a strong argument that it isn't.)

Swap is an adequate primitive for expressing such arbitrary permutations of pointers, and it guarantees that no pointers are duplicated or destroyed. The pointer permutation in the above NREVERSE loop could be thus expressed without temporary variables as follows, given a SWAPF analogous to SETF:

```
(swapf reversed (cdr list))
(swapf list reversed)
```

However, I do not have faith that such a microscopic approach to eliminating failure can scale to a whole computing system. In particular, a pointer cycle can be cut off from the rest of the object graph through such operations, and deciding whether or not that happens potentially requires whole-program analysis (or, worse, run-time whole-heap analysis).

Note that this approach to tree traversal does not bound the time needed for a tree traversal, only the space, and indeed it prevents you from killing the traversal process after a timeout to bound the traversal time cheaply.

(Linear trees (p. 1811) has more thoughts on this.)

Pagaltzis's wall-following tree traversal

You can do flexible immutable tree traversals in constant space if the tree nodes have parent pointers.

Aristotle Pagaltzis wrote the fascinating Tree traversal without recursion: the tree as a state machine in 2007, describing an algorithm for traversing a binary tree in constant space, if each node has a parent pointer as well as the usual child pointers; HN user kofejnik pointed out that the algorithm is equivalent to the standard wall-following maze algorithm, where you just keep your right hand on the right wall as you walk through the maze, so that whenever you re-enter a node with multiple exits, you exit through a new exit, your only state being your position and which way you're facing.

In summary, the next node and direction is either:

• the current node's left child, traversing down, if you're traversing down;
• the current node's right child, traversing down, if you're traversing up and the previous node is the current node's left child; or
• the current node's parent, traversing up, otherwise (when you're traversing up and the previous node is not the current node's left child; instead it will be the current node's right child).

This algorithm generalizes to general ordered tree traversal (you have to search through the child node pointers until you find the right one) and to cases where you don't want to traverse the entire tree for things such as database index traversal — you just pretend to have traversed the subtrees of no interest.

Pagaltzis pointed out in his article that, using node locks, the traversal can continue successfully even in the face of tree mutations as long as they don't detach a non-leaf node. But of course it won't traverse a consistent snapshot of the tree, and node locks eliminate its failure-free, bounded-time nature.

The Pagaltzis algorithm takes constant space and constant traversal time per tree node. However, to put bounds on the Pagaltzis algorithm's time to traverse the next leafnode — the metric of interest in many cases — you need to bound the depth of the tree, e.g., ensure that it doesn't degenerate into a linked list. If you can do that, though, you can also bound the space and time usage of a traditional recursive tree traversal.

Moving things in linked lists

Given an item `i` that is a member of a linked list and a place `p` in (the same or a different) linked list after which to insert it, you can move it to its new position in a failure-free, allocation-free, bounded-time fashion:

```
node *n = *i;
*i = n->next;
n->next = *p;
```

```
*p = n;
```

This is similar to the pointer-reversal tree-traversal approach described in the previous section; the end result is that you have permuted the pointers at `n->next`, `p`, and `i`. It is still failure-free even if `n->next` is a NULL pointer, but of course `p` and `i` must not be, and `i` must point to a node rather than null.

This operation is very commonly used in contexts like operating-system schedulers, where you might use it to move a job between different mutually exclusive queues. If one of the lists is a free list, this pattern becomes the next pattern, "fixed-size object pools".

Fixed-size object pools

If a program may be called upon to store some arbitrarily large amount of data at once, it cannot be both failure-free and bounded-time; however large its memory is, there is always the possibility that it will be called upon to store more data than there exists storage on the computer it is running on, and then either it must fail, or it must wait until more storage becomes available. So if a program is not bounded-space, it cannot execute failure-free in bounded time.

However, if there is an up-front limit to the amount of data the program needs to manage, and it's running on a computer with that much memory, it can be failure-free and bounded-time within that limit. Moreover, unless defeated by a too-clever-by-half operating system, it can verify that amount of memory is available at startup time — so it may fail to start up if running on a too-small machine, but then be failure-free thenceforth.

Preallocated object pools are a standard strategy for keeping time bounds on execution in this context: if your game needs to handle up to, say, 64 sprites at a time, you can preallocate and perhaps even preinitialize 64 sprite data structures and make a linked list of them. Then, whenever you need to allocate a new sprite, you grab the first node on that free list, a constant-time operation; to deallocate a sprite, you put it at the head of that list, also a constant-time operation.

Fixed-size queues to permit communication with non-failure-free systems

It's quite common for a computer system to contain some parts that have to be failure-free and use bounded space and bounded time, and other parts that don't; for example, a robot control system might contain one component that periodically toggles a pin to generate a waveform to control a servomotor, another component monitoring sensors and running a PID control algorithm that commands the first component what signal to emit, and a third motion-planning component that tells the PID control what set-point to use. If the waveform generator misses its deadlines (a few hundred µs of error at most is acceptable; RC model servos use a 50Hz pulse-position-modulation signal with a pulse width of 500µs to

2500µs encoding the commanded position, so 100µs of error is a 5% error), the waveform will have the wrong duty cycle, and the servomotor will receive the wrong command, possibly breaking the robot or a person near it. If the PID control algorithm misses its deadlines (typically a few milliseconds), the control system will at least oscillate and possibly go out of control. If the motion-planning algorithm runs slowly, though, the robot just takes longer to get things done.

This gives you a lot of freedom to have failures and missed deadlines in your motion-planning algorithm, but not if they can propagate to the waveform generator or even the PID controller. But clearly these components of the system need to communicate somehow.

The simplest approach is for them to share an atomically-updated variable; in an analog system, this might be the voltage on a wire connecting two of the components, or if we're running the PID controller on the same microcontroller that generates the waveform, a `volatile` variable written to by the PID controller process and read by an interrupt handler that generates the waveform, or an I/O register in a waveform-generation peripheral integrated into the microcontroller, like the AVR's timer/counter modules.

In the case of a `volatile` variable, though, we need to be careful of the danger of "torn reads": perhaps you're updating the value from 0x00ff to 0x0100, but on an AVR, that update involves sequentially setting two different bytes of RAM. If an interrupt handler runs between the two updates, it might see 0x0000 or 0x01ff and the robot might kill somebody.

(The "variable" might be arbitrarily large; you can think of framebuffers as being such "variables" shared between the main program and the display hardware, or in the case of the Arduino TVout library, the interrupt handler that generates the video signal. In this case "tearing" is manifested as on-screen "tearing" of images.)

This is similar to the CRT ADC case for which Gray invented Gray code, and so in this case you could try using Gray code to limit the magnitude of the torn-read error, but there are other more general solutions. One is to use a circular buffer to enqueue values to be communicated between components with different deadlines, a fixed-size queue that never blocks, in order to isolate failures within a given component.

XXX include implementation

This allows the PID component, for example, to isolate itself from failures in motion planning — if it goes to read messages from motion planning and none are there, it immediately gets a queue-empty message, a totally normal situation, and goes on about its business. Similarly, the waveform generator, if it has no new messages from the PID component, can go on about its business; and if there are interrupt handlers for the sensors feeding the PID controller, they too can add messages to a queue for it, not concerned about the PID controller's lack of adherence to their deadlines. If the queue gets full because the PID controller is failing, the sensor-reading messages will be lost, but the sensor-reading interrupt handler won't itself fail.

The same approach is essential in non-isochronous communication networks. If a gateway receives packets on one or more input streams whose maximum total input bandwidth is A, and forwards those

packets to one or more network interfaces whose total minimum guaranteed output bandwidth is B, then as long as A > B, it needs a bounded-size queue for outbound packets, and may drop packets at times. (Most commonly B = 0 because the common types of networks do not provide any guaranteed bandwidth.)

Anytime algorithms

"Anytime algorithms" or "interruptible algorithms" are a family of algorithms which can be interrupted at any time and get some kind of answer; if allowed to run longer, they can produce better answers. In particular, iterative approximation algorithms, which produce a series of progressively closer approximations to a mathematically correct answer, can be used as anytime algorithms.

For example, this Python implementation of the method of secants (from Separating implementation, optimization, and proofs (p. 780); see also Using the method of secants for general optimization (p. 1773)) is an anytime algorithm. It tries to find a zero of the scalar function `f` starting from guesses `x0` and `x1`, which are in general vectors:

```
def sec_seq(f, x0, x1):
    y = f(x0), f(x1)

    while True:
        yield x1
        x0, x1 = x1, x1 - y[1]*(x1 - x0)/(y[1] - y[0])
        y = y[1], f(x1)
```

After each iteration, `sec_seq` yields control back to the main program along with its current best solution; the main program can elect to resume it for another iteration or to accept that solution, either because it's adequately precise, because it's run out of time, or perhaps because a different approach to finding a zero is working better. (CPython doesn't allow us to prove tight bounds on the execution time of a single iteration, or to interrupt it either safely or in bounded time, but you could imagine a language that did; and, in some cases, even CPython's loosey-goosey behavior will be good enough.)

Anytime algorithms can exist in non-continuous domains, too — an optimizing compiler, for example, could reasonably work by constructing a correct but possibly slow compiled piece of code, then attempt various ways of optimizing it as long as there is time.

Mathematical optimization

A particular special case of anytime algorithms is the case of iterative mathematical optimization algorithms. Optimization, in this sense, is the mathematical problem of calculating the minimum of a function; for example, the minimum of $x^2 + x + 1$ is at $x = -½$, which can be calculated in closed form; minima of more complex functions are more difficult to find, and often we settle for local-search results rather than finding the global minimum.

In most cases, the objective is to reduce the function as low as possible, rather than rigorously guarantee that no other value is lower, and iterative optimization works by finding progressively lower and lower values.

So, for example, in SKETCHPAD, while it's *desirable* to satisfy the user's drawing constraints as quickly as possible, it's *necessary* to redraw the screen frequently in order to remain usable; so the relaxation procedure that seeks a satisfying value of the constraints runs a few steps before each screen redraw.

Optimization is a fairly general approach to solving "inverse problems", where we have a description of what properties a solution would have; optimization algorithms work especially well given a *fuzzy* description.

Constant-space representations

Traditional Lisps, Squeak, and newer versions of Python transparently overflow integer arithmetic into bignum arithmetic. This has the advantage that the results of integer arithmetic operations are always correct, while in many other programming languages, they may be only approximate, or they may overflow, either crashing the program or producing dramatically incorrect results.

However, this more common approach of producing approximate or dramatically incorrect results is often the price of failure-free, bounded-space, and bounded-time computation.

In signed 16-bit arithmetic, for example, 32767 + 1 = -32768, and 32767 + 3 = -32766; the first Ariane 5 rocket was destroyed by an arithmetic exception resulting from such an arithmetic overflow (though on a conversion from floating-point, rather than addition.)

Floating-point, as used for integer arithmetic in particular in JS, gives incorrect results for most operations, though not all. In 64-bit IEEE-754 floating-point, integer arithmetic (except for division) is exact up to 2^{53} = 9007199254740992.0; one result of this is that 9007199254740992.0 - 1 gives the correct result in most programming languages, while 9007199254740992.0 + 1 simply gives 9007199254740992.0 again. Under many circumstances, these errors are tolerable, although you can easily spend years on techniques to prove bounds on the errors in particular algorithms; and in many cases they produce results that are just wrong.

But both two's-complement integer arithmetic and floating-point arithmetic are usually constant-space and constant-time, and is often failure-free, although floating-point division by zero may produce a failure-free $\pm\infty$ value or, as in Python, an exception, depending on how things are configured; two's-complement integer division by zero almost invariably produces some kind of exception.

So what's the disadvantage of defaulting to bignums? Well, one of the first things I did with Squeak Morphic was to write an escape-time Mandelbrot-set renderer, but I had to set it to 16×16 pixels with a maximum number of iterations of about 10 or 12. (The Mandelbrot set is the set of complex numbers c for which the recurrence $z_i = z_{i-1}^2 + c$ remains bounded rather than zooming off to infinity as i increases, starting at $z_0 = 0$; escape-time rendering colors each pixel according to the number of iterations of the recurrence needed to exceed some limit, usually $|z| > 2$.)

When I set my iteration limit to a more reasonable number, like 256 or 100 or 30 or even 20, it took an unreasonably long time to render, or would hang completely. I was mystified; why was Squeak so slow?

A little investigation showed that the values of c, derived from

dividing pixel positions by the width and height of the array, producing exact rational numbers; so, for example, $z_4 = ((c^2 + c)^2 + c)^2 + c$ might involve calculating the eighth power of a number like 11/16, which is 214,358,881/4,294,967,296; but then z_5 would include its 16th power, which is 45,949,729,863,572,161/18,446,744,073,709,551,616; and so on. So my little Mandelbrot-set renderer was taking time — and memory! — *exponential* in the number of iterations.

Using an exponential amount of time is bad enough, but using an exponential amount of memory guarantees that you'll run out of memory before too many iterations.

Adding a decimal point somewhere was sufficient to get Squeak to switch from using exact arithmetic to using floating-point arithmetic, and suddenly I could use thousands of pixels and thousands of iterations.

So, constant-space arithmetic that doesn't crash on overflow is a useful way to make your arithmetic bounded-space, bounded-time, and failure-free, in the sense of not crashing (because of running out of memory or for any other reason).

More generally, if you are going to compute with some kind of entity, whether a number or not, in a failure-free fashion, you need to be able to represent it with a constant space bound, and usually in constant space. If your representation uses unbounded space, you cannot guarantee that your program will not run out of memory. (If it uses variable space with a constant bound, to get failure-free computing, you need to somehow ensure that space is always available when it needs to expand to its maximum size, which is feasible but usually more difficult than just always using the maximum size.)

But constant-space arithmetic gives you the wrong answer some of the time, which you could consider a software failure, even if the software doesn't consider it an "error condition arising". It is a widely shared observation that sometimes subtly wrong answers are worse than an outright failure, to the point that it's one of the core design principles of Python.

So, to use constant-space arithmetic without that danger, you need some extra care. There are three basic approaches: numerical analysis, overflow-safe arithmetic, and self-validating arithmetic.

Numerical analysis

Numerical analysis consists of statically analyzing your software to prove that it doesn't provoke the arithmetic errors inherent to constant-space arithmetic representations — in particular, the errors caused by the particular representation you're using — or that, if it does, the errors are acceptably small.

For standard C `signed` arithmetic, this involves statically ensuring that no overflow happens, because C `signed` arithmetic is undefined on overflow, which basically means that the compiler is free to break your program. The simplest approach is to statically associate constant upper and lower bounds with every arithmetic expression in the program, but this works only in the simplest cases; it fails for any code containing nontrivial loops. So, generally, you need to use bounds that are some kind of algebraic expression, rather than constants.

For wrapping binary arithmetic such as C `unsigned` arithmetic and the two's-complement arithmetic performed by CPUs when they are doing signed arithmetic, in many cases it isn't necessary to ensure that intermediate quantities don't overflow; it is only necessary to ensure that final results don't overflow. This is true for addition, subtraction, and I think multiplication, but not division; I think it is necessary that neither dividends nor divisors overflow.

For floating-point arithmetic, it's usually not a question of whether the result is correct — it's not — but of computing bounds on its error. The IEEE-754 standard guarantees that fundamental operations — +, -, ×, ÷, and perhaps surprisingly √ — are correct to within half an ulp, but offers less stringent guarantees for other operations, including exp, ln, exponentiation, and trigonometric functions.

Overflow-safe arithmetic

Self-validating arithmetic (e.g., interval arithmetic and affine arithmetic)

The object-embedding memory model

The object-embedding memory model eliminates certain failure modes from the object-graph memory model. In its pure form, all allocation is static, so there is no chance of any allocation failure. Embedded objects cannot be NULL, cannot be aliased (except via a pointer), and cannot be of the wrong type.

I suspect that this is one of the reasons for the empirically-observed reliability of C programs — although C has lots of undefined behavior and provides ample ways for programmers to shoot themselves in the foot, introduce failure, and introduce unbounded time and unbounded space, much of a typical C program is not touching those bits of the language. Although C has pointers, substantial parts of C programs use object embedding where programs in Java or Python would use pointers.

Records and sum types rather than arrays

Every time you index into an array (except with a constant index such as `a[0]`), you have a potential failure: the index may be outside the bounds of the array. Typical static program analysis techniques are too weak to prove that this cannot happen. In some cases this amounts to an allocation failure, as in the innumerable stack buffer-overflow bugs in 1990s C programs like NCSA httpd.

By contrast, if you have a (non-nullable) pointer to a known type of record, accessing the fields of the record is statically safe; it is constant-time and cannot produce failures. The pattern-matching approach used in ML allows this approach to extend to arbitrary Lisp-style object graphs; the compiler can statically verify that your pattern matching is exhaustive, statically ruling out run-time failures due to incomplete case analysis.

Arena allocators

If you allocate memory from an arena (or "region", in MLKit's term, or "pool", in the terminology of the Apache APR library) that is all freed at once, you get constant-time allocation and constant-time deallocation. To eliminate failure and bound time and

space usage, you then must merely prove a worst-case bound on the allocation of the program, and allocate a bigger arena than that.

Functional iteration/concurrency

Iteration protocols and per-array operations rather than array indexing

Hard priority scheduling

Wait-free and lock-free synchronization

Handling concurrency with locks or monitors introduces unwanted coupling between the time bounds of different processes: the worst-case execution time of code in a high-priority thread that needs to enter a monitor must include the time it needs to wait on whatever other thread might currently hold that monitor, which is to say, all code that can execute inside that monitor in any thread. Priority inversion, where a thread holding a lock that is blocking a high-priority thread has to sleep while an intermediate-priority thread runs, is perhaps the most severe manifestation of this problem, but it is more general.

Moreover, locks can produce deadlock, which results in a program failure; global analysis is required to rule deadlock out.

Interrupt handlers never use locks, because there is no mechanism to put the interrupt handler to sleep until the main program releases the lock, then wake it back up again. In some cases, the main program uses a lock (often with the big-hammer approach of disabling interrupts entirely) to lock out the interrupt handler entirely for a short time, but more generally, if the main program is doing something that an interrupt-handler execution might interfere with, it maintains the shared data in a consistent state the whole time, atomically committing its "transaction" at the end, typically with a compare-and-swap operation — a commitment which might fail if an interrupt handler ran in the meantime, requiring the main program to restart its transaction.

This same approach works for multiple threads of a program concurrently updating a memory area. It introduces a soft sort of failure — the need to retry a transaction — but, unlike locking protocols, it guarantees progress. In combination with hard priority scheduling, it entirely prevents a lower-priority process from slowing down a higher-priority process by forcing it to wait. However, the price is that if the lower-priority process can be forced to retry a transaction repeatedly, it can no longer guarantee time bounds on its execution.

Lock-free and wait-free synchronization algorithms are notoriously difficult to implement correctly.

Bounded-time, failure-free restricted virtual machines

Earlier I discussed how fixed-size queues can firewall failures and unbounded delays from propagating from one domain into another, enabling more-sensitive domains to be failure-free and bounded-time

while less-sensitive domains can be programmed in easier, more general ways. But fixed-size queues can only support a certain kind of arm's-length interaction; in many cases, more intimate levels of cooperation are desirable.

Bitcoin Script, BPF, and BPF's predecessor CSPF are loop-free virtual machines; a trusted virtual machine runs untrusted code in a timing-sensitive context (an interrupt handler in the case of BPF and CSPF!), which is safe only because the execution time is bounded by the script size. This allows fast-and-loose code like tcpdump, or your half-assed shell script invoking it, to safely execute code in a kernel interrupt context. And that's damn cool.

See Scriptable windowing for Wercam (p. 1256) for some thoughts on how to get a failure-free, bounded-time GUI with techniques like these.

Abstract interpretation with non-standard semantics

However, if you're writing most of your program in something comfortable like JS or whatever, it's going to feel pretty clumsy to have to write part of it in loop-free bytecode for a register machine.

Consider the case of pubsub. In the most general case, each subscriber has some Turing-complete filter function; you apply it to every message published to determine whether to deliver that message to that subscriber. It's often convenient to be able to run these filter functions in some kind of centralized message router, so that you don't have to send messages across a network to a subscriber that is just going to ignore them. But then, if the filter function uses unbounded space or time, the router might seize up. (Or it might fail.)

(There are some notes in Fast secure pubsub (p. 545) about how to do this by running the filter function in a transactional-memory-like sandbox.)

So suppose you have such a client–server pub-sub system, with a client library in Python, and you give it this filter function:

```
def wanted(message):
    if message.domain in whitelisted_domains:
        return True
    if message.length > 8192 or message.domain in blacklisted_domains:
        return False
    return spamminess(message) < spamminess_threshold
```

Python allows you to override attribute access and comparisons. This allows you to write an object like this:

```
class Comparator(object):
    def __init__(self, desc):
        self.desc = desc

    def __getattr__(self, attr):
        return Comparator(self.desc + '.' + attr)

    def __eq__(self, other):
        print(self.desc, '==', other)
```

```
        return False

    def __gt__(self, other):
        print(self.desc, '>', other)
        return False

if __name__ == '__main__':
    x = Comparator('x')
    x.domain in 'this is an example'.split() or x.length > 8192
```

When executed, this code prints:

```
x.domain == this
x.domain == is
x.domain == an
x.domain == example
x.length > 8192
```

That is, the `x` object has access to each of the things it's being compared with, and can choose the result of the comparison. If you invoke the above `wanted` function with such a `Comparator` object, it might produce output like the following before, presumably, it crashes the `spamminess` function:

```
message.domain == gmail.com
message.length > 8192
message.domain == godaddy.com
```

A slightly more sophisticated object could allow the caller to choose the results of the comparisons, in an effort to probe the tree of possible execution paths of the filter function. In this case, it could determine that if the first comparison returns `True`, then the filter function returns `True`; if not, but the second or third one returns `True`, then the filter function returns `False`; and then perhaps the function goes off into further calculations which raise an exception and terminate the probing process.

These observations permit the client library to compute a string of bytecode for the sort of restricted virtual machine described above, a bytecode function that calculates the same results as the filter function under some circumstances; this bytecode can then be sent to the server to prefilter the set of messages that get sent to the client.

I've used Python's operator overloading here because it's convenient, but operator overloading is just a particularly simple way of doing abstract interpretation with nonstandard semantics, not the only way. It's not even a particularly good way, in this case; it requires restarting the function from the beginning to explore each new execution path, and if `whitelisted_domains` is a `set` rather than a `list`, it fails silently.

The key relationship here, though, is that the failure-free, bounded-time code for the virtual machine produces a conservative approximation of the result of the more unrestricted code for the unreliable, loosey-goosey CPython virtual machine.

Stream processing

Topics

Forth assembling

Kragen Javier Sitaker, 2019-12-08 (updated 2019-12-11) (18 minutes)

Could you write an assembler with a smooth, backward-compatible path from raw binary (or, say, hexadecimal or octal) code up to a reasonable macro-assembler programming environment? Could it be smaller than existing assemblers? Maybe, but it's much easier to make a wrong turn and fall into the Turing tarpit.

The background

In an assembly-language program, traditionally, you have a sequence of operations that add bytes to the program being assembled and resolve labels, which may result in changing bytes that have already been assembled. The simplest operation is something like `db`:

```
db 0x1a                 ; clear screen
```

This appends the byte 0x1a at the current pointer (which I think is called `$` in traditional assembly-language syntax used by, for example, the CP/M assembler for the Intel 8080, as in this case; or `.` in AT&T or gas syntax) and advances that current pointer by one byte.

In an RPN calculation, by contrast, you have a sequence of operations that add numbers to a stack, advancing the stack pointer. If your RPN calculator is in hexadecimal mode and it processes the operation `1A` then it appends the number 0x1a at the current stack pointer and advances that current pointer by one stack item.

There is an obvious similarity between these two operations. But adding programmability to an RPN calculator is very easy. Can we exploit this? What happens if we try to incrementally add RPN calculation abilities to a minimal assembler? Can we put entire instruction sets into macro libraries, as Assembler bootstrapping (p. 2922) suggests (and points out was common in the past), if we simply conflate the operand stack with the memory space we're assembling intoo?

The simplest assembler

Jeremiah Orians took exception to my claim in Assembler bootstrapping (p. 2922) that a *minimal* assembler would have labels and macros, since his stage0, like Edmund GRIMLEY EVANS's bcompiler and various things we've discussed on kragen-tol and kragen-discuss over the years, starts from simply converting hexadecimal or octal into machine code. Now, it's merely a question of semantics whether such a program, which recognizes only hexadecimal or octal input, should be called an "assembler" or not --- normally an assembler recognizes instruction opcode mnemonics and handles labels --- but it's undeniably a very useful tool for bootstrapping.

So, for example, here's an MS-DOS 64-byte demo I wrote a few years ago, in an octal format such a program might accept, which can be obtained from `od -vbAn`:

```
260 023 315 020 304 057 211 350 367 056 001 001 211 305 061 306
061 322 211 301 061 333 210 367 211 337 301 373 002 001 337 211
303 301 373 010 001 337 001 367 133 046 210 075 103 123 211 303
301 373 004 001 332 211 323 301 373 004 051 330 342 326 353 306
```

As explained in An 8080 opcode map in octal (p. 1059), the Intel family of machine languages, including the 8080, the 8086, the i386, and amd64, and even the 8008 (as explained in Further notes on algebras for dark silicon (p. 1753)), are much easier to read in octal than in hexadecimal, and the code to convert from octal to binary is simpler than the code to convert from hexadecimal to binary (especially traditional 0123456789abcdef hexadecimal instead of 0123456789jklmno).

Here's such an octal converter in C:

```
#include <stdio.h>

int main() {
    int b = 0, c, d = 0;
    while ((c = getchar()) != EOF) {
        if ('0' <= c && c <= '7') d++, b = (b << 3) | (c - '0');
        else if (d) putchar(b), b = d = 0;
    }
    return 0;
}
```

`gcc -fomit-frame-pointer -Os -Wall -Werror -std=gnu99` on i386 compiles this function into 29 instructions in 63 bytes, although the executable file is 7340 bytes with 434 bytes of .text and hundreds of bytes of other cruft.

This is an assembly version of the machine code GCC generated:

```
        .globl main
main:   push   %ebp
        mov    %esp, %ebp
        push   %esi
        push   %ebx
        and    $~0xf, %esp
        sub    $0x10, %esp
reset:  xor    %esi, %esi
        xor    %ebx, %ebx
next:   call   getchar
        cmp    $-1, %eax
        je     end
        sub    $'0, %eax
        cmp    $7, %eax
        ja     emit
        shl    $3, %ebx
        inc    %esi
        or     %eax, %ebx
        jmp    next
emit:   test   %esi, %esi
        je     next
        mov    %ebx, (%esp)
        call   putchar
```

```
        jmp     reset
end:    lea     -8(%ebp), %esp
        xor     %eax, %eax
        pop     %ebx
        pop     %esi
        pop     %ebp
        ret
```

This is pretty reasonable code but it can obviously be cleaned up and whittled down a bit, especially if we suppose that crt0 doesn't care if we preserve its %esi and %ebx:

```
        .globl main
main:   xor     %esi, %esi
        xor     %ebx, %ebx
next:   call    getchar
        cmp     $-1, %eax
        je      end
        sub     $'0, %eax
        cmp     $7, %eax
        ja      emit
        shl     $3, %ebx
        inc     %esi
        or      %eax, %ebx
        jmp     next
emit:   test    %esi, %esi
        je      next
        push    %ebx
        call    putchar
        pop     %ebx
        jmp     main
end:    xor     %eax, %eax
        ret
```

That's 49 bytes of machine code, but of course it depends on C stdio and exiting via crt0, and also 8 of those 49 bytes are C stdio addresses. So a bare-kernel version might be more appealing:

```
        .globl _start
_start:
main:   xor %esi, %esi          # flag for whether we have data
        xor %ebp, %ebp          # the data we maybe have
next:   push %esi               # stack balance, also zero the buffer
        xor %eax, %eax
        mov $3, %al             # __NR_read
        xor %ebx, %ebx          # fd = stdin, 0
        mov %esp, %ecx          # buf on stack
        xor %edx, %edx          # count =
        inc %edx                #         1
        int $0x80               # system call, results in %eax
        dec %eax
        test %eax, %eax         # if not 1:
        jnz end                 # bail out
        pop %eax                # fetch character read
        sub $'0, %eax
```

```
        cmp $7, %eax
        ja emit                 # if non-digit, emit buffered byte if any
        shl $3, %ebp            # otherwise shift to make space for digit
        inc %esi                # and set flag
        or %eax, %ebp
        jmp next
emit:   test %esi, %esi         # Do we have buffered data to emit?
        je next
        xor %eax, %eax
        mov $4, %al             # __NR_write
        xor %ebx, %ebx          # fd =
        inc %ebx                #      stdout, 1
        push %ebp
        mov %esp, %ecx          # buf on stack again
        xor %edx, %edx
        inc %edx                # count = 1
        int $0x80
        pop %edx
        jmp main
end:    xor %eax, %eax          # __NR_exit =
        inc %eax                #            1
        int $0x80
```

This is considerably more instructions, 37, and back up to 67 bytes of code. But, linking with `-nostdlib`, the static stripped executable is 416 bytes with only 67 bytes of `.text`, without any trickery with minimizing ELF headers; Brian Raiter's whirlwind teensy tutorial succeeded in getting a 45-byte Linux ELF executable, so probably it's possible to get this executable to be about 112 bytes with such trickery.

Backwards-compatible assembly features

An interesting idea here is to add more traditional assembly-language capabilities in a backward-compatible way to this octal-dump language. A really straightforward thing to add would be an RPN operation like `|`, which replaces the last two bytes emitted with their bitwise OR. (More traditional operations include things like `+` and `-`, but `|` is more immediately useful.) So, for example, you could write the `211 305` in the above code, meaning `mov %ax, %bp`, as `210 1 | 0 5 300 | |`; in this case `210` is `mov` (or `lea` or `pop`), and `1` is the Ev <- Gv variant of it, while `300` is the "mod" for the register-register mod-r/m byte, `0` is `%ax`, and `5` is `%bp`.

This requires buffering up the program to be output, unlike the "assemblers" above. This expands it significantly to 88 bytes of code:

```
        .globl _start
_start: mov $output, %edi       # output pointer
reset:  xor %esi, %esi          # flag for whether we have data
        xor %ebp, %ebp          # the data we maybe have
next:   push %esi               # stack balance, also zero the buffer
        xor %eax, %eax
        mov $3, %al             # __NR_read
        xor %ebx, %ebx          # fd = stdin, 0
        mov %esp, %ecx          # buf on stack
```

```
        xor %edx, %edx          # count =
        inc %edx                #         1
        int $0x80               # system call, results in %eax
        dec %eax
        test %eax, %eax         # if not 1:
        jnz end                 # bail out
        pop %eax                # fetch character read
        sub $'0, %eax
        cmp $7, %eax
        ja emit                 # if non-digit, emit buffered byte if any
        shl $3, %ebp            # otherwise shift to make space for digit
        inc %esi                # and set flag
        or %eax, %ebp
        jmp next
emit:   test %esi, %esi         # Do we have buffered data to emit?
        je ops
        mov %ebp, (%edi)
        inc %edi
ops:    cmp $('| - '0), %eax     # was the character a |?
        jne reset                # if so, OR the last two bytes:
        dec %edi
        mov (%edi), %dl
        or %dl, -1(%edi)
        jmp reset
end:    xor %eax, %eax
        mov $4, %al             # __NR_write
        xor %ebx, %ebx          # fd =
        inc %ebx                #      stdout, 1
        mov $output, %ecx       # buf = output
        mov %edi, %edx          # count = output pointer
        sub %ecx, %edx          #                       - buf
        int $0x80
        xor %eax, %eax          # __NR_exit =
        inc %eax                #             1
        int $0x80
        .bss
output: .fill 65536, 1, 0
```

That 88-byte program, given the input `300 50 1 | | 300 50 1 | 300 50 1`, does output the bytes (represented in octal) 351 300 051 300 050 001.

Passing that assembly through `cc -nostdlib`, `objcopy -S -R .note.gnu.build-id`, and `od -vbAn` gives the following byte listing, from which the executable can reproduce itself:

```
177 105 114 106 001 001 001 000 000 000 000 000 000 000 000 000
002 000 003 000 001 000 000 000 270 200 004 010 064 000 000 000
050 001 000 000 000 000 000 000 064 000 040 000 003 000 050 000
004 000 003 000 001 000 000 000 000 000 000 000 000 200 004 010
000 200 004 010 020 001 000 000 020 001 000 000 005 000 000 000
000 020 000 000 001 000 000 000 000 020 000 000 000 220 004 010
000 220 004 010 000 000 000 000 000 000 001 000 006 000 000 000
000 020 000 000 004 000 000 000 000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000 000 000 000 004 000 000 000 000
004 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

```
000 000 000 000 000 000 000 000 277 000 220 004 010 061 366 061
355 126 061 300 260 003 061 333 211 341 061 322 102 315 200 110
205 300 165 045 130 203 350 060 203 370 007 167 010 301 345 003
106 011 305 353 334 205 366 164 003 211 057 107 203 370 114 165
314 117 212 027 010 127 377 353 304 061 300 260 004 061 333 103
271 000 220 004 010 211 372 051 312 315 200 061 300 100 315 200
000 056 163 150 163 164 162 164 141 142 000 056 164 145 170 164
000 056 142 163 163 000 000 000 000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
013 000 000 000 001 000 000 000 006 000 000 000 270 200 004 010
270 000 000 000 130 000 000 000 000 000 000 000 000 000 000 000
001 000 000 000 000 000 000 000 021 000 000 000 010 000 000 000
003 000 000 000 000 220 004 010 000 020 000 000 000 000 001 000
000 000 000 000 000 000 000 000 001 000 000 000 000 000 000 000
001 000 000 000 003 000 000 000 000 000 000 000 000 000 000 000
020 001 000 000 026 000 000 000 000 000 000 000 000 000 000 000
001 000 000 000 000 000 000 000
```

More or less translating this back to C, we get this:

```c
#include <stdio.h>

char output[65536];

int main() {
    char *op = output;
    int b = 0, c, d = 0;

    while ((c = getchar()) != EOF) {
        if ('0' <= c && c <= '7') b = (b << 3) | (c - '0'), d++;
        else {
            if (d) *op++ = b, b = d = 0;
            if (c == '|') op--, op[-1] |= *op;
        }
    }

    fwrite(output, op - output, 1, stdout);
    return 0;
}
```

Or, in GForth:

```
: getchar tib 1 stdin read-file  swap 1- or if 0 else tib c@ 1 then ;
: read-to-end  begin  getchar while  over execute  repeat  drop  ;
create output 65536 allot   output value op  0 value b  0 value d
: digit 3 lshift or ;  : handle-digit  b digit to b  1 to d ;
: out d if  b op c!  op 1+ to op  then  0 to d  0 to b ;
: do-or op 1- dup to op  @  op 1- @ or  op 1- ! ;
: dispatch [char] | = if do-or then ;
: octal [char] 0 - >r  r@ 0 >=  r@ 7 <=  and if r> 1 else rdrop 0 then ;
: handle-byte dup octal if handle-digit drop else out dispatch then ;
: main ['] handle-byte read-to-end  output op output - type  bye ;
```

(That took me an hour to get working; I didn't know how file I/O

worked in ANS Forth, and it took me quite a while to work out that I'd left out the `drop` in `handle-byte`. I ran this as `gforth osmdf.fs -e main < foo.oct > foo.com`; it doesn't do the right thing when fed from a terminal.)

Or, in Python 2:

```
import sys

output = []
b = None

for c in iter(lambda: sys.stdin.read(1), ''):
    if '0' <= c <= '7':
        b = ((b or 0) << 3) | int(c)
    else:
        if b is not None:
            output.append(b)
            b = None
        if c == '|':
            output[-2:] = [output[-2] | output[-1]]

sys.stdout.write(''.join(map(chr, output)))
```

The next obvious step in the direction of a real assembler with mnemonics is enough of a macro system to enable you to write that as `mov1 %ax %bp rr | |`, which requires only being able to define words that expand to numbers, like EQU. And then after that you probably want something that allows you to write `%ax %bp mov-rr`, which requires `mov-rr` to be able to somehow insert an instruction byte *before* its arguments. We'd like to be able to define it at least as something like `: mov-rr swap 3 << | 300 | 211 swap ;` and maybe better as something like `: mov-rr { src dest } 211 300 src 3 << | dest | ;`.

So we end up in Forth

And the off-the-shelf design for that kind of thing is an indirect-threaded Forth. Minimally, it has an operand stack (in this case, this doubles as the code being generated, although with one word per item rather than one byte per item), a return stack so that subroutines can call other subroutines, and a dictionary it can look subroutines up in. Some subroutines in the dictionary, like the hardcoded `|` above, are implemented by machine code (the six bytes 117 212 027 010 127 377 above --- `dec %edi; mov (%edi), %dl; or %dl, -1(%edi)`). Normally it also has `@` and `!` operations to access a random-access memory. It has the possibility of adding new items to the dictionary, which can only be freed in a LIFO order.

And it's fairly straightforward to see how you could accumulate label relocations in the Forth dictionary as linked lists of pointers into the growing code, skip the operand-stack pointer around as desired to deposit code into different segments, and the other kinds of things you'd want to do in an assembler.

And, once we've settled on having a flexible interpreter in our assembler, we can implement core "assembler" functionality in the interpreted language. Above there's an example of how 10 lines of interpreted Forth can implement the functionality we needed 43 lines

of assembly language for. The Forth should have better compositionality, although in this case, the difference is mostly because the assembly is in a more vertical format --- the Forth is about the same amount of text.

But maybe we wouldn't want the interpreter to be Forth. If we're just looking for the absolute minimum interpreter that lets us shove the main assembler functionality out into macro libraries that get loaded when we run the "assembler", there may be a number of Turing tarpit possibilities, things like the lambda-calculus or SKI-combinators, although probably in Polish Notation or RPN syntax. So maybe the question of "the minimal assembler for bootstrapping" boils down to the well-trodden ground of designing a minimal interpreter instead. And maybe that's still true even if your interpreter also contains, in some form, the capability of the 49-byte octal bootstrapping program above, so that if you feed it a stream of octal bytes it will still faithfully convert them to binary.

Topics

- Programming (p. 3658) (286 notes)
- Independence (p. 3520) (63 notes)
- Small is beautiful (p. 3714) (40 notes)
- Assembly language (p. 3328) (25 notes)
- Forth (p. 3461) (19 notes)

Wikipedia people

Kragen Javier Sitaker, 2016-06-01 (6 minutes)

In Neal Stephenson's novel *Seveneves*, there is a group of people called Cycs who have undertaken to collectively memorize a carefully preserved five-thousand-year-old Encyclopedia Britannica. One of the characters is named Sonar Taxlaw, after the 17th volume, which she has full mastery of at her 16 years; she also has a loose acquaintance with the rest.

It beggars belief that a novel written so recently would have chosen the Britannica over a paper copy of Wikipedia, but what would the Wikipedia-based equivalent look like?

The English Wikipedia is about 5 million articles at this point and about 50 gigabytes of UTF-8 text, for a mean of some 10 kilobytes per article. The "Vital Articles" lists are carefully chosen subsets of 10, 100, 1000, and 10 000 (actually 9841) articles, which tend to be longer than the mean (which is dragged down substantially by stub articles, which are vanishingly unlikely to be Vital), at around 22 printed pages or 320 kilobytes each.

If we take Stephenson at his word that a 500-or-so-page Britannica volume can be adequately mastered by one bright person, then we have about 23 Wikipedia articles per person; so the Wikipedia Vital 100 could be summarized by four or five people, the Vital 1000 (close to Britannica in size) by about 44 people, and the Vital 10000 by about 440 people.

However, it is often the case that useful knowledge arises only from connections between topics. Suppose that instead we want to ensure that each *pair* of articles is present in the mind of at least one person, so that at least syntheses that need to draw on any two different articles have a chance to arise.

Here's one scheme for how to do that: divide the articles, perhaps randomly, into groups of 11, and make a matrix with a row and a column for each group of articles: 91 groups to cover the Vital 1000, say, gives us a 91×91 matrix. Then, assign a person to each above-diagonal cell in this matrix, (* (/ 90 2) 91) = 4095 people in this example. Have that person learn all 22 articles from their row and their column.

This seems somewhat wasteful: each person contains (* (/ 22 2) 21) = 231 pairs of articles, but (* (/ 10 2) 11) = 55 of them are shared with everybody else on their row, and another 55 with everybody else on their column. So only (- 231 55 55) = 121 of their pairs, just over half, are unique. So out of the total (* 4095 231) = 945945 pairings of articles in people's heads, you only have (* (/ 1000 2) 999) = 499500 unique pairs. Furthermore, a person who can master 23 articles could actually contain (* (/ 22 2) 23) = 253 pairs, which could potentially reduce the necessary number of people further to (/ 499500 253.0) = 1975. But it's not clear how to reach or approach that bound.

Regardless, the number of people needed to cover all the article *pairs* in this way is proportional to the square of the number of articles, or rather, the amount of information. So to cover all the pairs in the Vital 100 with the simple assignment scheme above, you would need only ten people.

Printing out all of English Wikipedia on paper at full size would use about 3.4 million pages; linearly reduced 4:1, at which point it's 3-point text that's barely still readable without a magnifying glass, would be about 212 thousand pages, or 212 reams of paper.

Wikipedia:Size notes, "In 2015, Michael Mandiberg published the English Wikipedia in 7473 volumes of 700 pages each via Lulu, an online e-books and print self-publishing platform, distributor, and retailer." Probably nobody has yet printed out all 7473 volumes, a total of 5.2 million pages of paper at a cost of US$500k, but he did print out 106 volumes for an art exhibit in New York. The Print Wikipedia blog post notes that the table of contents occupies 91 volumes.

Presumably the same 4:1 reduction trick would bring this down to 325 000 pages, a mere 465 volumes or 325 reams at a cost of US$31,250. At the standard 80 g/m², an A4 sheet weighs 5 g, so this printout would weigh 812 kg. Staples sells 75 g/m² US Letter paper in a case of 10 reams for US$56, so the paper alone would cost US$1820. However, their acid-free paper costs US$22 per ream and weighs 89 g/m², which would raise the price to US$7150. Presumably you could get the weight down by using thinner paper, as was normally used for encyclopedias and dictionaries, perhaps at a higher dollar cost. Wikipedia says "onionskin" typically weighs 25–39 g/m²; 30g would lower the weight to (* 325 500 (/ 30 16.0) .001) = 305 kg. Amazon lists a "9-pound" "FIDELITY" brand onionskin bond paper for US$25 per ream, weighing 3 pounds per ream, which works out to about 34 g/m². It's 100% wood pulp, but despite that, it's buffered, pH neutral, and 100-year rated. (1000 years is readily achievable.). If it's 50µm thick, then the shelf length in A4 size would be (* 325 500 50 .001 .001) = 8.125 m.

(Working out from the shitty basis weight standard, the basis ream for bond paper is 500 sheets of 17" × 22", which is how we know it's 34 g/m².)

Here in Argentina, the terminology seems to be "papel alcalino", and 75g "papel alcalino" costs only about US$10 per 500-sheet A4 ream. This would bring the paper cost of the whole Wikipedia project to US$3250.

Topics

Byte-stream pipe and antipipe façade objects for editor buffers

Kragen Javier Sitaker, 2017-04-10 (3 minutes)

I was pondering a design for gap buffers in C, trying to figure out the interface. At first I thought two access functions would be enough, one to read a range and one to write a range, in both cases to a raw char pointer. (Plus a function to get its length.)

Then I realized that meant that reading from or writing to a file would require an extra inefficient copy through an intermediate buffer. So I thought I'd want to add functions that take file descriptors, too, because that's a pretty important thing to do with strings, really the only thing other than copying them around in memory if you're on Unix.

But then I realized that I'd probably want to copy data from one gap buffer to another at some point, but for that I'd still need the extra inefficient copy. So I was thinking of adding another function, and then I realized what I was doing.

All that's needed here is a generic interface for piping byte streams around, which could be either push (a byte-sink interface) or pull (a byte-source interface). Then we just need pipe and antipipe façades for gap buffers, files, and raw char pointers.

C isn't that great for generic interfaces, but you could do it this way:

```
typedef struct {
    int (*write)(void *user_data, const char *s, size_t len);
    void *user_data;
} byte_sink;
```

Given a `byte_sink`, you can invoke it several times with different pieces of data. A gap buffer asked to write its contents to a `byte_sink` might invoke it twice, once for the part of the text before the gap and once for the part after it.

(In other contexts, a byte-sink interface might also include a close method, an error method, and so forth. In this case, the `write` function has a return value that I intend to use to indicate errors.)

Here `user_data` is used to distinguish different objects of the same type, since C function pointers aren't closures. I originally heard the term in the context of Tcl, but by now Lua, OpenGLUT, GLFW, GTK+ and GLib, Core Audio, systemd, and Box2D also use the term to mean more or less the same thing.

Then, to write to gap buffers, file descriptors, or raw byte buffers requires three different `byte_sink` write functions, and functions that return `byte_sink` wrappers for them; to read from gap buffers, file descriptors, or raw byte buffers, we need three different functions that take `byte_sink` arguments.

So instead of five functions, now we have nine functions. But now, in addition to being able to copy data into and out of gap buffers, we can also copy data from one file descriptor to another; and if we add a new string buffer type in the future (like an array of gap buffers) we

can copy data to and from it in the same way.

This is still kind of crappy for reading data from files, because it still involves an extra buffer copy — the file descriptor byte source function is obligated to allocate it.

Topics

Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code?

Kragen Javier Sitaker, 2018-04-27 (4 minutes)

You can take the Lisp 1.5 metacircular interpreter (from e.g. http://www.righto.com/2008/07/maxwells-equations-of-software-oexamined.html, originally from p.13 of The Lisp 1.5 Programmer's Manual http://www.softwarepreservation.org/projects/LISP/book/LISP%20o1.5%20Programmers%20Manual.pdf, in M-expressions and directly write them as stack-machine code.

```
\ evalquote[fn;x] = apply[fn;x;NIL]
:EN$;
\ apply[fn;x;a] = [atom[fn] -> [eq[fn;CAR] -> caar[x];
\                               eq[fn;CDR] -> cdar[x];
:$r!x!f!f@a[f@1=[x@AA;]f@2=[x@AD;]
\                               eq[fn;CONS] -> cons[car[x];cadr[x]];
f@3=[x@Ax@DAk;]
```

This amounts to 54 characters for these four lines. Proceeding more or less in this way, the 21 lines of code from the Lisp manual should be about 280 characters, or about four lines. The non-Lisp primitives used here are taken from StoneKnifeForth, largely from Forth, and the meanings of the characters include:

- `:` defines a (one-byte) label as a function entry point;
- `;` returns from the current function;
- `!` stores NOS (next-to-top-of-stack) at the address in TOS (top-of-stack), popping both;
- `@` fetches the value at the address in TOS, replacing it as TOS;
- `[` pops TOS and conditionally jumps to the matching `]` if it was zero, thus enclosing a conditional;
- `=` pops TOS and NOS and pushes a value that is zero unless they were equal;
- literal numbers push themselves on the stack;

StoneKnifeForth defines 21 such primitives.

The other characters used are:

- `r`, `x`, and `f` are presumed to be elsewhere-defined addresses of memory cells that we can use for convenient storage of parameters `r`, `x`, and `fn` respectively — recursive calls, however, may need to explicitly save and restore such things on a Forth stack;
- `E` represents `evalquote`, and `$` represents `apply` (after its meaning in Haskell, I suppose);
- `A` is `car`, `D` is `cdr`, and `k` is `cons`.

You can implement the A (CAR) and D (CDR) operations with something like the following, presuming labels `c` and `d` pointing to appropriately-sized arrays in which to store the car and cdr pointers

themselves:

```
:Ac+@;
:Dd+@;
```

My recollection from doing this in C in 2007 https://www.mail-archive.com/kragen-hacks@canonical.org/msg00164.html is that there are a fair number of other things you need to implement that are sort of hidden under the covers here: input parsing, output printing, atom interning, memory management, call/return ("activation record management", as I said), and error handling. The C version was 154 lines of code that worked, plus another nonworking 23 lines of code for garbage collection, for a total of probably about 177 lines. Of that, 40 lines of C was devoted to the 21 lines of code from the Lisp 1.5 metacircular interpreter which could maybe be compressed down to 4 lines (284 characters) of line-noise stack code. If the same ratios held, the whole working Lisp would be 1254 characters of stack code, which is about 20 lines; 15–25 lines is probably a reasonable estimate, or maybe 45 lines if it's formatted to be as readable as possible.

It might be a somewhat better idea to do a simple Scheme, Lua, or Smalltalk instead, with proper lexical scoping, instead of the dynamic-scoping early-bound mistake that was 1960s Lisp. There's no reason to expect that they'll be much more code, but if they are, 1960s Lisp might be good enough.

Something like this is probably a reasonable way to bring up a more or less high-level language in a fairly minimal amount of code. This provides the following estimate for the part of the abstraction ladder to get to a high-level programming environment:

- 158 lines of Verilog: a CPU like the J1a;
- 132 lines of quasi-Forth: a quasi-Forth compiler to native code like StoneKnifeForth;
- 198 lines of Python or similar: a bootstrap interpreter to run it to get the initial native code, as in StoneKnifeForth;
- 45 lines of code like the above: a high-level programming language.

Total: 533 lines of code.

(Maybe 90 lines is a better estimate for the Lisp part.)

Of course, this doesn't include operating systems, filesystems, text editors, fonts, font rendering, networking, Verilog logic synthesis, and so on. But it's a start.

Topics

- Programming (p. 3658) (286 notes)
- Independence (p. 3520) (63 notes)
- Small is beautiful (p. 3714) (40 notes)
- Instruction sets (p. 3526) (40 notes)
- Archival (p. 3322) (34 notes)
- Stacks (p. 3730) (21 notes)
- Forth (p. 3461) (19 notes)

Toward a minimal PEG parsing engine

Kragen Javier Sitaker, 2018-06-06 (4 minutes)

So, tonight, prompted by last night's frustration with μSQL parsing of input, I hacked together a PEG parser with syntactic sugar in Python. It lets you write PEGs that look like this:

```
class Arithmetic(Grammar):
    sp = Lit(' ') | '\n' | '\t'
    _ = sp + _ | ''
    digit = Lit('0') | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    digits = digit + digits | digit

    number = (digits + '.' + digits | digits + '.' | Lit('.') + digits | digits) o
o+ _
    exponentiation = number + '**' + _ + number | number
    multiplicative = (exponentiation + '/' + _ + multiplicative
                      | exponentiation + '*' + _ + multiplicative
                      | exponentiation)
    additive = (multiplicative + '+' + _ + additive
                | multiplicative + '-' + _ + additive
                | multiplicative)
```

Then you can invoke e.g. `Arithmetic.additive.parse` on a string. Like μSQL, it has some major problems, but it more or less works — to a much greater extent than μSQL, in fact.

The implementation of this is about two pages of code, but I trimmed a version of it with slightly less magic down to under half a page:

```
from collections import namedtuple
class PE:
    __add__ = lambda s, o: Seq(s, o if isinstance(o, PE) else Lit(o))
    __or__  = lambda s, o: Alt(s, o if isinstance(o, PE) else Lit(o))
    __invert__ = lambda self: Neg(self)
class Lit(PE, namedtuple('Lit', ['text'])):
    def parse(self, text, position):
        if self.text == text[position:position + len(self.text)]:
            return self.text, position + len(self.text)
class Seq(PE, namedtuple('Seq', ['a', 'b'])):
    def parse(self, text, position):
        a = self.a.parse(text, position)
        if a:
            b = self.b.parse(text, a[1])
            return ([a[0], b[0]], b[1]) if b else None
class Alt(PE, namedtuple('Alt', ['a', 'b'])):
    def parse(self, text, position):
        return self.a.parse(text, position) or self.b.parse(text, position)
class Neg(PE, namedtuple('Neg', ['negated'])):
    def parse(self, text, position):
        return None if self.negated.parse(text, position) else (None, position)
```

```
class Nonterminal(PE):
    def __init__(self, name):
        self.name = name
    def parse(self, text, position=0):
        r = self.rule.parse(text, position)
        return ((self.name, r[0]), r[1]) if r else None
```

This half-page of code leads me to thinking about StoneKnifeForth and its kin, which was of course why I started playing with PEG parsers in the first place ten years ago. Such a parsing engine on top of a more primitive programming language would necessarily be more complicated — the code above implicitly depends on garbage collection, dynamic dispatch, recursive subroutines, Python's lax encapsulation, and even operator overriding for its brevity.

But I don't think the penalty would be that bad. It does need recursive subroutines, and Alt and Neg in particular need to save a previous text position on the stack. Seq doesn't need to save a previous text position, but it does save the result from its left-hand side to include in its own result. Nonterminal only needs to remember its own name.

If you're building up some kind of parse tree, which is probably a good idea if only to fix the associativity problem introduced by PEGs' lack of left recursion, you need to allocate it somewhere — but typically a very simple allocation strategy is adequate for that.

And the dynamic dispatch here could be taken care of rather simply with a switch over the different types of node in the tree representing the grammar. Something like

```
switch(node->type) {
case LIT:
   return memcmp(tp, node->v1, node->v2) ? fail : advance(node->v2);
case SEQ:
   if (fail == parse(node->v1)) return fail;
   char *v1 = result;
   return (fail == parse(node->v2)) ? fail : succeed(sequence(v1, result));
case ALT:
   char *saved = tp;
   if (fail != parse(node->v1)) return succeed(result);
   restore(saved);
   return parse(node->v2);
case NEG:
   char *saved = tp;
   if (fail != parse(node->v1)) return fail;
   restore(saved);
   return succeed(NULL);
}
```

although I'm probably glossing over any number of important things there.

Topics

Generic programming with proofs, specification, refinement, and specialization

Kragen Javier Sitaker, 2017-05-10 (6 minutes)

Here's a specification for sorting:

A. sorted(x) is some permutation p of x such that ∀i∈[0, #p-1): ¬ $p_i > p_{i+1}$.

Given a way of enumerating the permutations of a finite sequence and some relation >, this specification A is executable: you can enumerate all the permutations of x, and if you find one that satisfies ∀i∈[0, #p-1): ¬ $p_i > p_{i+1}$, you return it.

You can consider this algorithm, or an implementation of it for a given machine, a refinement of specification A, specialized for a given permutation-enumeration algorithm, data type, and ordering relation. You can write a proof that this plan correctly implements specification A, which is to say, it's a refinement of it. It isn't a very efficient plan, since it takes superexponential time in the worst case; you can also write a proof of that.

There isn't a clean separation between programs, algorithms, plans, and specifications; you can continue refining and specializing that specification toward something executable on a particular machine. Let's call the brute-force sorting algorithm that you would thus derive specification B.

(Some specifications may be uncomputable; for example, if you specify that a program should be fun to use, or compute Chaitin's number omega for a flavor of Turing machine, you will not be able to execute those specifications, even inefficiently. Even if the specification says to find a real number for which some potentially computable property holds, that may be uncomputable if the number itself is uncomputable. But if the specification says to find a computable real number for which the property holds, if one exists, then you can satisfy that specification by enumerating all possible programs. Henceforth I will disregard such uncomputable specifications in this essay.)

Given the premise that your relation > (or rather its negation ≤) is at least a weak partial order, you can write a proof that such a sorted permutation exists. Also, you can write a proof that the original specification implies that ¬∃i: $p_0 > p_i$, and that all the subsequences of p are sorted versions of themselves, including in particular p[1:], if it exists.

Given these (plus a couple more things), you can also write a proof that the following specification is equivalent to, and thus a refinement of, specification A:

C. sorted(x) is some permutation p of x such that ¬∃i: $p_0 > p_i$
and, if x is nonempty, p[1:] = sorted(p[1:]).

With a naïve backtracking search, this leads fairly quickly to

selection sort, given a proof of the usual linear-time minimum-finding algorithm; this is already a much more efficient sort. And you can write a proof of that. Let's call this algorithm specification D.

(I've been writing `sorted(x)` as if it were a function, but of course it is a relation; it is a bit of an abuse of notation to say that `p[1:] = sorted(p[1:])`.)

There are several characteristics of what we are doing here:

• Specifications are written in a form that is entirely independent of the other relations and data they are defined in terms of. Specification A does not say that `>` is a weak partial order, or that `x` is a finite sequence, or what algorithm to use to enumerate permutations. Instead, somewhere else, we write a proof that, IF `>` is a weak partial order, THEN specification A describes a nonempty set of permutations, and also specification B is equivalent to specification A.

• Some specifications are refinements of others, in the sense that they logically imply the specifications they are refinements of. Some of these refinement relations are only valid given certain premises, which amount to specializations of a specification. Some of these refinements are derived automatically; others are written by hand and then proved to be refinements.
• Some specifications are sufficiently specific that we can prove efficiency properties about them, such as specifications B and D; others are not, such as specifications A and C.

We can take as an analogy the process a SQL database uses to evaluate queries. First we write a specification in SQL of the query we want to run; then the database derives a plan that it proves is a refinement of our query — in this case, the plan will produce a specific sequence of tuples that is one of the sequences of tuples that would be a correct response to our query. We can understand much of compilation in this fashion, as well.

Moreover, we can do generic programming in this way.

What I'm describing here is very different from the usual programming process. Normally, we write a given subroutine only once, and we mix premises about what kind of data it's operating on and which other operations it's invoking in with the code. The compiler might derive a refinement of it for a given machine, but we don't derive them ourselves, and the compiler doesn't provide us with a proof that the machine code is a sound refinement of our source code, nor does it prove efficiency properties of the machine code.

Topics

Printed circuits on fired-clay ceramic

Kragen Javier Sitaker, 2019-08-13 (11 minutes)

Suppose you want to fabricate a grid of wires, like for a capacitive touchscreen. In theory there are lots of ways you can do this: etching or electroless plating of printed circuit boards (especially including flex), weaving insulated wires (ideally multistranded) into a flexible cloth, slicing aluminized mylar into ribbons and then gluing the ribbons onto paper, drawing graphite lines on both sides of a sheet of paper, etc., but I wanted to focus on the case of glazed ceramic.

The basic touchpad design

Let's say you start by making a bisque ceramic plate with a bunch of narrow gold-leaf ribbons running in parallel across it. Then you can add an engobe or other glaze on top of those ribbons (except, perhaps, along one edge where you are going to attach electrodes), then another set of ribbons running across that, perpendicular to the first. With the proper choice of glaze, this new set of ribbons will be insulated from the first set by the glaze. If you then want them to be insulated from the world, you can do a second glaze firing of the plate with another, lower-melting glaze on top of the ribbons, again with the possible exception of one edge.

This procedure should give you a grid of capacitively-coupled wires whose coupling is dependent on you touching the plate. This is also potentially useful as a macroscopic ROM: if you add some conductive paint to the surface at some junctions, it can act to increase the capacitive coupling at those junctions, thus representing information. Under suitable circumstances, this reader might even be able to capacitively read heavy graphite marks on paper.

Concrete calculations

Consider the case where each layer of glaze is 100 μm thick and has a relative permittivity of 5 (that of glass — see Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033)), the ribbons are 1 mm wide and 5 mm apart, and you place a conductive circle of 11 mm in diameter over one of these junctions. The capacitance $\varepsilon A / d$ between the wires without the circle is $5\ \varepsilon_0\ 1\ \text{mm}^2 / 100\ \mu\text{m} = 0.44$ pF, a capacitance small enough to be hard to measure. (ε_0, the vacuum permittivity, is about 8.85 picofarads / meter.) The capacitance between the upper wire and the circle is $5\ \varepsilon_0\ 11\ \text{mm}^2 / 100\ \mu\text{m} \approx 4.9$ pF, and the capacitance between the lower wire and the circle is $5\ \varepsilon_0\ 11\ \text{mm}^2 / 200\ \mu\text{m} \approx 2.4$ pF. So the series capacitance between the two wires is 1.62 pF, about four times larger than the capacitance without the conductive thing.

Detecting an extra 1.2 pF in a circuit usually requires a relatively carefully built measurement circuit to keep stray capacitances from drifting, but it's entirely feasible. A thinner glaze or one with higher permittivity, or wider ribbons, would provide larger capacitances which would be even easier to measure; but wider ribbons would also reduce the relative difference in capacitance between the

finger-present and finger-absent states. A thicker glaze would probably necessitate either high-permittivity ingredients or wider ribbons.

At 10 MHz, which is a convenient frequency, 1.62 pF gives a capacitive reactance of $(2\pi fC)^{-1} = 9.8\ k\Omega$, while 0.44 pF gives 36 kΩ. These numbers are large compared to the expected resistance of the wires in the plate: according to Paper/foil relays (p. 3273), gold's resistivity is $2.44\times10^{-8}\ \Omega m$, so a 1 mm × 100 nm × 100 mm wire is 24.4 Ω. (See Spark particulate sieve (p. 2047) for information on metal thicknesses.) But they are small compared to the input impedance of any random op-amp.

Other conductors

Gold leaf has the advantages of being cheap and of not oxidizing, even when ceramic is being fired. (Most of the ceramic I've done was fired at 1020° (see file `ceramics-notes`) but porcelain can be fired at up to 1400°; some earthenware made from ferrous clay can be fired as low as 600°.) I have the impression that the usual pottery gilding technique is somewhat more complicated than merely gluing gold leaf to the greenware or bisqueware before a firing, but I'm not sure why.

Some oxides are also conductive; indium tin oxide is the one commonly used for transparent conductors in LCDs, but zinc oxide and some heavy metal titanates are also semiconductors. Yttria-stabilized zirconia is conductive at high enough temperatures, and was formerly used as an incandescent element in Nernst lamps, and silicon carbide, though not an oxide, oxidizes slowly enough to survive ceramic firing. Some of these might be practical to deposit on the surface as part of a glaze, but I suspect that others would dissolve in the common glazes.

If you're firing in a sufficiently reducing atmosphere, you may be able to use powdered copper or copper oxide to get copper traces rather than gold leaf; for uses like printed circuits, this could offer immensely higher conductivity. (Gold is very nearly as conductive as copper, but gold's cost advantage in this context would come from being able to use a very thin layer of it, which gives it substantial resistance.) Carbon and many different metals might work in a reducing atmosphere.

Applications other than touchpads and ROMs

Of course you can solder other components to the traces thus produced and make a more or less conventional printed-circuit board, though perhaps at somewhat higher cost.

The HP 9100 used a ROM design somewhat similar to the above, but using inductive coupling rather than capacitive coupling, and using a now-traditional multilayer printed-circuit board rather than layers of ceramic glaze. Such an inductive ROM could totally work, and inductive sensing could also be used to detect the proximity of ferromagnetic materials rather than conductive ones.

The traditional reason for applying gold leaf to thin sheets of vitrified silicon dioxide was for a burglar alarm: the gold leaf running around the outside of a glass window was part of a Wheatstone

bridge, and either breaking it along with the glass, or shorting it out to avoid breaking it, would change its resistance enough to trigger the alarm. So you could use such embedded conductors for detecting not only touch but also breakage.

The resistance calculated above for the gold-leaf traces is in a range that would be usable for resistance heating, although common pottery is somewhat fragile to thermal shock, so the heating would need to be fairly slow. (I suspect this can be improved in the body of the pottery by foaming, but I don't know how to improve it in the glaze.) The gold itself will withstand fairly high temperatures. Silicon carbide and zirconia are other promising materials for printed heaters.

An induction coil, for example for a pancake motor, can be printed on a ceramic surface in this way. If it's to be used for high-power applications like motors, cooking, or other bulk heating, rather than sensing, you probably want the lower resistances achievable more cheaply with copper.

Gold leaf's extreme thinness could conceivably be used to fabricate extremely fine circuit details, if you can pattern it finely enough. For example, you could imagine traces of 100 nm thickness, as above, but only 100 nm width rather than 1 mm, spaced 100 nm apart. This would provide 5 parallel conductors per micron, so a 10-mm-diameter circle could contain a spiral pancake coil of 50'000 turns, which would be a fairly sensitive detector of varying magnetic fields.

The (presumably) high resistance and high dielectric strength of the glaze could enable electrostatic-like fields, perhaps capable of moving macroscopic objects such as bits of aluminum foil. According to How would you maximize the energy density of a capacitor? (p. 42), fused quartz doesn't break down until 30 MV/m; if the glaze is 10 MV/m, you could safely use voltages of up to about 1000 V with the 100-μm glaze thickness I guessed at above.

By exposing spark gaps on the surface of the plate, not placing glaze over them, you could fabricate a kind of multiplexed matrix display. Probably you could only illuminate one spark gap at a time, and you'd probably need a few hundred volts to reliably get a spark going in air. Unlike conventional printed-circuit boards, glazed ceramic will not become conductive when you arc across it (there's nothing to char) and the gases produced by the spark will not attack it; however, the plasma will gradually vaporize it, and perhaps more importantly, vaporize the spark-gap contacts, especially if they are thin gold leaf. Also, the device produces nitrogen oxides and ozone.

Since arcing produces substantial UV, you could possibly get pixels of different colors by depositing different fluorescent colored dyes under different spark gaps.

Selectively generating arcs as you sweep paper across the ceramic could be used to selectively char the paper, thus printing on it, though not in an archival-safe manner.

Glow/corona discharge in air might permit longer life than arcing. Such spatially selective application of glow discharge in air could be useful not only for making visible images with light but also for selectively activating chemically-inert surfaces such as glass or polyethylene, or selectively initiating polymerization or ozone bleaching.

Glow discharge in a low-pressure inert-gas atmosphere, of course,

gives you a neon bulb. I think that's also how gas-plasma displays work.

Such exposed pairs of contacts are also, of course, how rubber-dome keyboards work, at lower voltages than those needed for spark gaps; these can usually be operated by touching them with fingers as well.

Air-quartz flash bulbs are the gold standard for short non-laser light pulses for high-speed photography; such a spark gap over (largely-quartz) fired clay amounts to something similar to an air-quartz flash bulb. Aside from the uses in stroboscopy, an array of such spark gaps firing in a raster sequence produces short flashes of light emanating from different points; by focusing these through a lens onto an object, they scan a spot of light across it in a fashion similar to a flying-spot TV camera, but without moving parts. The resulting waveform of reflected light detected at some point P produces an image of the scene as seen through the lens, as if illuminated from point P. (See Flying spot reilluminatable stage (p. 2358) for more thoughts on this.)

Topics

Improving LZ77 compression with a RET bytecode

Kragen Javier Sitaker, 2016-04-05 (updated 2016-04-06) (3 minutes)

LZ77 compression uses backreferences of the form (distanceback, length). You could think of this as a call to a subroutine consisting of a sequence of previously produced characters: the distanceback is like a PC-relative jump destination address. But the length is curious: it tells how long the subroutine is! In ordinary machine code, several different subroutines can share a common tail by having different entry points, each of which falls through into the next; but in LZ77-land, several different "subroutines" can share a common *head* as well, by being "called" from "callsites" with different lengths.

This means, for example, that the word "compression" in the first paragraph here can serve as a source both for the prefix "com" in the word "common" later in the paragraph, and for the word "compression" in this paragraph.

However, is it worth it? It imposes the cost of indicating the length on the "caller", which seems like it might be dumb. Consider if instead we had

```
'LZ77 '
X { Y { 'com' } 'pression' }
' uses ... nes can'
Z { ' share a '; call Y; 'mon' }
'tail by having...nes" can'
call Z
'\n*head* as well'
```

In this approach, there's still a count, but it's in the "callee" rather than the "caller", so the expense of storing it is amortized across callers. Here I've written it as a "}" terminator rather than a count. And you can see that we don't lose the ability to use common prefixes, but we don't save anything unless a particular backreference is used more than once.

But what is the tradeoff? It depends on how exactly we do the storage. If we actually use a "RET" terminator, then we can share tails for free, just as in machine code. But nesting isn't free: either we need a matching start marker, or we need to replace nesting with an explicit call. X above can't be represented just as 'com' RET 'pression' RET; otherwise when you called it, it would return too soon!

• It could be represented as { { 'com' } 'pression' }, which gives you suffixes like "om" and "ession" for free, though not "ompression", at the cost of needing both begin and end markers.
• Or X could be represented as Y 'pression' RET, with Y defined earlier outside of the main flow, but the extra invocation of Y is probably larger than the begin marker.
• Or, instead of using a terminator, we could just put the count into the begin marker: "9:3:compression", counting the initial 3-letter string as a single element of a 9-element sequence. This loses the

ability to share tails for free.

With any of these three possibilities, you also now have the possibility of representing the distanceback with an index into a list of defined subroutines, rather than a number of bytes back. This should save a number of bits from the distanceback, effectively enabling much larger windows.

This is a relatively obvious tweak to LZ77, so it's probably been tried before.

Topics

The Bleep ultrasonic modem for local data communication

Kragen Javier Sitaker, 2018-12-10 (updated 2018-12-11) (8 minutes)

Suppose you want to transmit data a short distance between cellphones and Cortex-M0 devices like the STM32F0 family over ultrasound. What's the simplest thing that could possibly work?

You can probably generate and even analyze audio on the cellphone in a webpage, using the Web Audio API. This suggests that maybe you want to see if you can press biquad filters into service.

Probably FSK for the encoding, and probably the sweet spot for audibility by humans but not cellphones is around 18kHz. And the further apart you have your frequencies, the faster your tone-detection filters can work. If they're at 17.5kHz and 18.5kHz, for example, your Q could be only about 18 and still separate them by 3dB. A 17.5kHz square wave will be just a sine wave after it gets through a standard audio speaker — its second harmonic is amplitude 0, and its third harmonic is 52.5kHz, which is not going to get reproduced by the speaker.

A Q of 18 means you get about 1kbps, which is adequate for some applications. (?! where does this number come from)

You probably want some kind of hefty error-detection code so that you don't turn random noise into garbage data.

I was thinking that the Goertzel algorithm is the standard algorithm for tone detection, but if your microphone isn't picking up substantial harmonics, it's probably fairly harmless to just chop the signal with a square wave or modified square wave, and chopping with two square waves in quadrature should give you I and Q signals. However, 17.5kHz and 18.5kHz are 2.52 and 2.38 samples per cycle, respectively, at 44.1ksps, so quadrature chopping would be somewhat tricky! You'd have to resample to a higher sample rate first. You can definitely use Goertzel though.

You probably want to window the signals both on output and on input, both to avoid spectral leakage down to the audible spectrum and to improve discrimination between them. Empirically, a first-order moving average is not good enough here, unless it's sized specifically for the beat frequency.

I think a negative-weight feedforward comb filter on input, plus a bit more frequency margin, might help with frequency discrimination. Setting the 1/0 threshold requires some amount of adaptation, empirically, which suggests that a constant-weight code would be helpful; Manchester encoding is the simplest, though it discards half the bandwidth.

I'm thinking maybe 17500 Hz and 18970 Hz, which differ by 1470 Hz, which is a period of 30 samples. Halfway through that period, they're ½τ out of phase, and ¼ or ¾ of the way through (7.5 and 22.5 samples), they're 90° out of phase, so when one is at its maximum, the other is at its zero. So, for example, if $y(n) = x(n-7) - x(n)$, the 17500Hz signal is attenuated by a factor of 0.985 (-0.13 dB, hardly at all), while the 18970Hz signal is attenuated by -0.070 (-23 dB).

Perhaps better, if y(n) = x(n) - x(n-5), the 17500Hz signal is attenuated by -0.100 (-20dB) and the 18970Hz signal is attenuated by 0.812 (-1.8dB). Similarly, if y(n) = x(n-34) - x(n), the 17500Hz signal is attenuated by -0.050 (-26dB) and the 18970Hz signal is attenuated by 0.709 (-3dB).

So we could maybe filter each signal with a four-stage cascade. For 17500Hz:

```
y(n) = x(n-7) - x(n)
a(n) = y(n) * cis(17500 τ n/fs)
b(n) = a(n) + b(n-1)
c(n) = b(n) - b(n-30)
```

For 18970Hz:

```
y(n) = x(n-5) - x(n)
a(n) = y(n) * cis(18970 τ n/fs)
b(n) = a(n) + b(n-1)
c(n) = b(n) - b(n-30)
```

And in practice a(n) and b(n) could be combined into a single Goertzel stage, and the final c(n) stage can be decimated.

The initial comb-filtering stage, in addition to attenuating the opposing signal relatively by 23dB, also works as a single-pole high-pass filter. The 7-sample window has its lowest resonance at 3150 Hz and other resonance peaks at 9450 Hz, 15750 Hz (the one whose sidelobe we're using (?)), and 22050 Hz.

XXX Oh wait this is totally wrong. I was calculating those attenuations based on the value of $|\sin(\omega t)|$ at those points, but actually the attenuation we want is $|\cos(\omega t) - \cos(0)|$.

Thinking about how to *emit* the levels inexpensively, it occurred that Don Lancaster's Magic Sinewave approach might work well if the baud time is a common multiple of the carrier periods (i.e. a multiple of 30 samples at 44100 Hz), which is to say that the baud rate should be a factor of 1470 Hz. 490 baud and 735 baud (90 and 60 cycles respectively at 44100 Hz) suggest themselves. The idea is sort of like wavetable diphone synthesis: you divide time into 30-sample grains (≈0.68 ms) and during each grain time you play an appropriate precomputed grain according to the current bit, the following bit, and your position within the bit. So in the 735-baud case, for example, you have two frame positions and four situations (00, 01, 10, 11), so 8 precomputed grains. For machines with an audio DAC, in 16 bits at 44100 Hz, each of these are 30 samples and 60 bytes, so the total wavetable is 480 bytes. The 490-baud case would need 12 precomputed grains.

(This is assuming that we're using a window with support of less than two transition intervals.)

Machines without an audio DAC probably need to compensate by using a higher sample rate; this will be limited by their CPU load, CPU speed, and memory. Supposing they can afford 4 KiB of program memory to store the wavetables, that's 16384 samples, or 2048 samples per grain, which works out to 3.01 megahertz, which is a low enough speed that you don't need fancy signal integrity stuff, and maybe it's within reach of even things like AVRs (which would

probably need an external ADC to receive signals, anyway). This gives you about 68 bits per 44100sps sample; $\sqrt{68} \approx 8$, so I think you'll have a signal-to- quantization-noise amplitude of about 8, or 18.3 dB. That's not a wonderful noise budget but it's probably adequate. Possibly we can dither that noise up to very high frequencies where it won't even get digitized on the receiving end.

You might be able to win a bit more by just repeating the same grain during the transition for the 00 and 11 cases, taking advantage of the fact that those grains themselves are time-reversal invariant, and time-reversing the 10 and 01 cases.

XXX wait, 17500 isn't divisible by 1470.

I did try the comb filter thing. I'd forgotten that feedforward combs have sharp nulls and very broad peaks. It helps a lot.

Better: 17640 Hz and 19110 Hz. Those *are* divisible by 1470.

Thanks to Nick Johnson, Ezequiel Alfie, Alastair Porter, Florian Pignol, Arthur Sittler, Kia, Colby Kraybill, Eugene Jercinovic, Andrea Shepard, John Bognar, and Rick Bartells for all the things I've learned from them that enabled me to get this to work. I haven't even mentioned it to most of them, so don't blame them if it's bogus.

Topics

Generalizing my RPN calculator to support refactoring

Kragen Javier Sitaker, 2016-10-17 (12 minutes)

My regular netbook isn't working because I left the charger at work. I've managed to get this old Thinkpad to boot, though installing Ubuntu seems to be not working (this ten-year-old CD-R is probably corrupt, judging from the kernel log messages), and I can't remember the disk encryption password I set on this machine probably nine years ago, which leads me to think about how much hassle is involved in even the most basic computing tasks.

If only I had an easily bootable operating system I could fire up from this pendrive in a matter of seconds! If only I could add notes to it from anywhere anonymously, later accepting them!

I feel like my RPN calculator app may be a step in the right direction. In it, keys 0–9 add to the number at the current position in the RPN program (space works as a separator between adjacent numbers), and the program is executed after each edit to build up a stack of expressions that are then executed. Operator keys are used to combine expressions into larger expressions. The operator "," concatenates numbers into a vector, with the usual kind of APL broadcasting operations. The "l" and "e" operators are ln and exp, respectively, enabling relatively easy powers and roots. The "i" operator is the APL iota, but zero-based, allowing the instant construction of sequences covering some range.

Vectors are automatically plotted, as well as being displayed numerically. A somewhat generous estimated calculated precision is maintained for each number, and its display is limited to that precision, in order to improve the signal-to-noise ratio of the display.

Alt-left and alt-right (or, for phones, $ and &) perform structural editing of the RPN program, moving an entire subexpression left or right over other instructions. Left and right (or the parentheses) navigate the RPN program, showing you the intermediate value calculated at each node.

At present it lacks even the minimal abstraction ability to use the same value twice. For example, to compute the first six triangular numbers, you can use the sequence `6i6i1+*2/`, resulting in the expression `i 6 * (i 6 + 1) / 2`. But to change this to the first 10 numbers, for example, you must go back and edit both of the sixes.

It also lacks any kind of aggregate calculation, even summing or array indexing, any kind of mouse interaction, and decimal points.

Abstraction

I'm increasingly coming to the conclusion that stacks are good for expression evaluation, but too confusing when you try to use them for general-purpose data storage; the position of any given value relative to the top of the stack is constantly changing. So probably to reuse values, it's better to use registers, i.e. variables, rather than providing stack manipulation operators, at least in the context of interactive calculations.

However, a set of interactions has occurred to me that seem like

they should make abstraction by refactoring quite simple:

• # to fetch a value recently computed, repeated to refer to less recent such values; in effect this is the "introduce local variable" refactoring, but with a user interface based on the Mill CPU's belt, and no need to specify ahead of time which values will be reused. However, once you are done wrapping a computation in a definition in this way, later # operations will skip over its internal structure.
• : when within such a definition to turn the subexpression you're currently looking at into a parameter, pushing that subexpression out into every place where it's invoked, turning it into a function. If you do this with the entire contents of the definition, you have reversed the refactoring, and the subexpression evaporates.

So, for example, in the case above about the triangular numbers, after having typed "6i", you could type "#", which would put the "i 6" into a subexpression, used twice. Maybe this would be displayed like this:

```
x = i 6 = 0, 1, 2, 3, 4, 5
x = 0, 1, 2, 3, 4, 5
x = 0, 1, 2, 3, 4, 5
```

Then, on typing "1+*2/", you would see something like this:

```
x = i 6 = 0, 1, 2, 3, 4, 5
x * (x + 1) / 2 = 0, 1, 3, 6, 10
```

If you move the pointer back to the 6 and type ":", you are pushing the 6 up to the level where x is invoked, making x a function; the result would be something like this:

```
x(y) = i y
x(6) * (x(6) + 1) / 2 = 0, 1, 3, 6, 10
```

At this point, the two 6es have become independent (although that may not have been the right default). To make them dependent again, you can put the cursor on the second one and type ## to fetch the first 6, then move right and delete the second one. The result would be something like this:

```
x(y) = i y
z = 6
x(z) * (x(z) + 1) / 2 = 0, 1, 3, 6, 10
```

If you go to the end and "#" it to turn the triangular-numbers calculation into a local subexpression, you get something like this:

```
x(y) = i y
z = 6
a = x(z) * (x(z) + 1) / 2 = 0, 1, 3, 6, 10
a = 0, 1, 3, 6, 10
a = 0, 1, 3, 6, 10
```

Now it may be desirable to make z a parameter of a. If you put the cursor on the first reference to z and use ":", you get this:

```
x(y) = i y
z = 6
a(b) = x(b) * (x(z) + 1) / 2
a(z) = 0, 1, 3, 6, 10
a(z) = 0, 1, 3, 6, 10
```

Then you can move over to the second z and use "##" to turn it into another reference to b, then delete the z:

```
x(y) = i y
z = 6
a(b) = x(b) * (x(b) + 1) / 2
a(z) = 0, 1, 3, 6, 10
a(z) = 0, 1, 3, 6, 10
```

If you now move the cursor onto the 6 in z and use ":", that will push it out into the invocations of z. That leaves z with nothing left to do, so it evaporates:

```
x(y) = i y
a(b) = x(b) * (x(b) + 1) / 2
a(6) = 0, 1, 3, 6, 10
a(6) = 0, 1, 3, 6, 10
```

(I'd also like to be able to manipulate programs the way rpn-calc manipulates algebraic expressions, building them up step by step with example values.)

During the course of these edits, there are times when a function will compute multiple values. For example, consider this definition:

```
a(b) = x(b) * (x(z) + 1) / 2
```

The RPN program is something like this:

```
local b  b x  z x  1 +  *  2 /
```

Upon introducing the second reference to b, but before deleting the z reference, it looks like this:

```
local b  b x  b z x  1 +  *  2 /
```

That works out to these expressions:

```
x(b)
b * (x(z) + 1) / 2
```

The question then is whether a, at that point, should be considered to be returning two values or merely computing x(b) and discarding the result.

Vectorization

To a great extent, not just loops but also nested functions can be eliminated entirely by sufficient vectorization, so to some extent this is an alternative to the previous item. Vectorization is less flexible but also more comprehensible.

The basic idea is that variables have values that depend on circumstances, and you can represent pretty much any variable as a scalar variable that depends on circumstances. For example, you could think of the altitude of land as a number, but one that depends on the latitude and longitude, and maybe time if you are modeling that. The textual content of an editor buffer is a character-valued variable that depends on the position within the buffer. The country of land is a categorical measurement which also depends on latitude and longitude. It is a sensible question what is the maximum altitude for each country, ranging across all the latitude/longitude pairs within that country.

You could reasonably display such vectors in tables, with one table for each set of circumstances that a vector's values depend on. Vectors depending on the same set of circumstances would be displayed in the same table. The traditional way to lay out such tables is with one attribute per column and one row per instance, but the reverse is probably better in this case, with one row per formula and one column per instance. As you calculated, rows would appear and disappear, with the formula displayed on their left followed by a sparkline.

It's not totally clear to me how to mix the display of the stack results with table-style display. Multiple hierarchical levels of circumstances are a reasonable thing to have; you could imagine using colspan cells within the same table to display values that depended on less than the whole set of circumstances, in particular including the empty set of circumstances: a scalar or constant.

The objective is to be able to add new circumstances later, as in the example above in which the altitude comes to depend on time as well as latitude and longitude; you could also imagine it depending on the reference spheroid (WGS 84?) and the data source being consulted. This suggests that aggregation operations (such as, for example, max) should specify a list of circumstances to range over all possible values of (and thus eliminate from the dependency list), rather than a list of circumstances to retain in the dependency list.

Constraints

Often the calculations I'm doing are in terms of the interrelated values of some mathematical model. The simplest interesting example is perhaps a sphere, which has a radius, a diameter, a cross-sectional area, a surface area, and a volume, any one of which determines all of the others. More complex models may involve conditionals, piecewise approximations from empirical data, and more parameters --- a cylinder, for example, has a volume, a radius, and a length, any two of which determine the third, as well as other properties, of course. It is desirable to express those relationships once for a given model and then derive an effective calculation procedure from that expression.

Units

It's very common for me to do calculations including measurement dimensions, and I wish my calculators were better at this. I often use units(1) to do the calculations, but it has some shortcomings:

- There's no way to define variables or functions or vectorize

calculations;
• Its output display defaults to expressing things in terms of fundamental units, which is often fairly confusing --- joules or volts, let alone farads, are difficult to recognize in that format;
• Often the input interpretation is surprising, and you may not notice an unexpected interpretation.

Topics

• Programming (p. 3658) (286 notes)
• Human–computer interaction (p. 3493) (76 notes)
• Stacks (p. 3730) (21 notes)
• Constraint satisfaction (p. 3387) (9 notes)
• Mill (p. 3584) (7 notes)
• Refactoring

Electroluminescent matrix

Kragen Javier Sitaker, 2016-07-27 (2 minutes)

The first gas plasma screens for Plato used two grids of parallel wires which created plasma at their intersections; you could do the same with an electroluminescent material like ZnS:Cu or, maybe, $SrAl_2O_4$ (LumiNova).

For example, on a hard, flat insulating substrate such as glass, you could lay down a bunch of horizontal 100µm-diameter copper wires at a spacing of 250µm; on top of that, you deposit a 200µm-thick layer of small ZnS:Cu crystals, maybe dispersed in a binder ("electroluminescent paint"); and on top of that, you lay down another bunch of 100µm-diameter copper wires with a spacing of 250µm, this time vertical.

For example, *980 pixels horizontally (720 vertical wires) at this resolution would be 183 mm, and 6615* pixels vertically (990 horizontal wires) would be 251 mm. So you could get a very readable full-page monochrome display with 712800 pixels by controlling only 1710 wires, an average of 417 pixels per wire.

If you ground one of the horizontal wires, leave the others floating, and put an high-frequency AC voltage on some, but not all, of the vertical wires, you should get glowing spots where those vertical wires cross the wire you have grounded. The intensity of each spot should be jointly proportional to the voltage and the frequency. By switching from one horizontal wire to the next, you can draw an arbitrary pattern of pixels.

Potential advantages of this approach include very low cost, inexpensive fabrication technology, low power consumption, and very low duty cycle — ZnS:Cu, as used on analog oscilloscope screens, has a time constant of around nine minutes (???), while $SrAl_2O_4$ has a time constant of about 15 minutes. This would allow you to use a refresh time of several seconds or even minutes to hours rather than tens of milliseconds.

(I've been thinking for a while that if you have a laser you can rapidly deflect with mirrors, you could use that to paint an image on a glow-in-the-dark screen, thus avoiding the need for the vacuum and X-ray shielding of a CRT, and also the hundreds or thousands of wires used by this design.)

Topics

Memory safe virtual machines

Kragen Javier Sitaker, 2019-12-04 (14 minutes)

Pointer arithmetic gives your languages freedom to implement different kinds of storage, including stack-allocated, heap-allocated, structs, arrays, struct extension with new fields on the end, linked lists, and whatnot. But it poses difficulties for safety.

The Forth virtual machine is fairly simple; as described in Notes on reading eForth (p. 1398), Bill Muench's eForth model, about 176 machine-code instructions for the 8086, consists of EXIT, EXECUTE, _LIT, _ELSE, _IF, C!, C@, !, @, RP@, RP!, >R, R@, R>, SP@, SP!, DROP, SWAP, DUP, OVER, CHAR-, CHAR+, CHARS, CELL-, CELL+, CELLS, 0<, AND, OR, XOR, UM+, REDIRECT, !IO, ?RX, TX!, and BYE, plus the machine-code routines RESET, LIST1, and VCOLD. Of these, the ones that access linear memory are just C!, C@, !, and @, about 10% of the total. Everything that accesses memory does so by way of these four primitives.

However, when developing software and sometimes even when running it, it's very convenient to get crashes and exceptions rather than wrong answers, ideally as close to the bug as possible. Using such a simple memory model sacrifices that possibility, since there's no way to distinguish out-of-array-bounds accesses, or integers wrongly interpreted as pointers, from valid pointers. Moreover, this makes garbage collection impossible. Can we define a virtual machine that is *almost* as simple and flexible, but provides better safety properties?

The rest of the Forth model

We can categorize the eForth code into control flow --- EXIT (i.e., return), EXECUTE, _IF, _ELSE, and LIST1; stack manipulation --- >R, R@, R>, DROP, SWAP, DUP, OVER; meta-stack manipulation --- RP@, RP!, SP@, and SP!; I/O operations --- REDIRECT, !IO, ?RX, TX!; startup and shutdown --- RESET, VCOLD, and BYE; ALU operations --- _LIT, 0<, AND, OR, XOR, and UM+; and portability helpers --- CHAR-, CHAR+, CHARS, CELL-, CELL+, and CELLS. This is a good approximation of a minimal usable virtual machine, although probably subtraction, multiplication, and division would be welcome additions.

My StoneKnifeForth, inspired by eForth, has a different set of primitives, some of which are things eForth implements in interpreted Forth rather than in machine code, such as comments. SKF is about 1400 instructions. Its memory operations are @, !, and store, which last is C!.

The C pointer approach

Suppose we define an untyped virtual machine whose memory supports the four operations fetch word, store word, fetch byte, and store byte, with register arguments to indicate the memory location to access, and an allocate operation to allocate N bytes of new memory. How can we implement it efficiently with some degree of memory safety?

Maybe we can codify more or less the C pointer rules: make pointers be (segment, offset) pairs, say 32 bits for each; mere integers have a distinguished invalid segment value for the segment part, such as 0. Subtraction of two pointers produces an integer if the segments are the same or crashes your program if not. Addition or subtraction of a mere integer to a pointer produces another pointer within the same segment. Pointer comparisons for equality compare both the segment and the offset. Pointer comparisons for ordering crash if the segment differs. No other pointer arithmetic is valid. The virtual machine checks dereferences against an upper bound it stores for the segment.

The allocate operation creates a new segment and returns a pointer to its start.

None of this stops programs from storing pointers in memory with the store-word operation and then altering their segment bits; for example, the XOR one-pointer double-linked-list hack can be implemented in this way. That means that garbage collection is not safe.

This approach allows, for example, moving a struct that mixes pointers and non-pointers to a different part of memory, in the same or a different segment, merely using memcpy. Note, though, that the situation where this is most advantageous --- persistence to files or transmission across a network --- can't take advantage of this, because the segment bits will not be valid in the other process, whether separated by space or by time.

The KeyKOS approach

Suppose we want to be sure that a subroutine we invoke cannot forge pointers to random memory, but only access data it has been given segments for. To prevent pointer forgery, we must strictly segregate segment identifiers from character data and, for example, ordinary integers. It is okay for offsets into a segment to be freely intermixed with character data, though.

One way to do this is to have separate byte segments ("segments" in KeyKOS) and descriptor segments ("nodes" in KeyKOS). Descriptor segments contain only descriptors; byte segments contain only bytes. The virtual CPU contains both descriptor registers and integer registers. Memory access instructions take an address consisting of a descriptor and an integer offset; there are six of them --- load descriptor, store descriptor, load integer, store integer, load byte, and store byte. Descriptors can only be loaded from and stored to descriptor segments, while integers and bytes can only be loaded from and stored to byte segments. The only operation on descriptors, other than storing them or using them in a memory access, is comparison for equality.

There are a few variants of this approach. Rather than having separate segments, you could have a "data fork" (of bytes) and a "resource fork" (of descriptors) for each segment; this avoids the dynamic check, but means that instead of having separate `allocate_byte_segment` and `allocate_descriptor_segment` calls, you'd have one call that takes two arguments. This way, a data structure that contains both pointers (to, potentially, other segments) and byte data can be a single segment, rather than a descriptor segment that points to a byte segment.

Or the virtual machine could maintain a bit for each byte in the segment, indicating whether it currently contains descriptor bytes or non-descriptor bytes; loading it with the wrong operation would crash your program. Alternatively, only attempting to load a descriptor register from non-descriptor bytes would crash the program, while loading descriptor bytes into a data register would be fine.

This approach is not very compatible with the C or Forth view of the world, and like varying-sized inline objects, it leads to a certain amount of duplication in machine code --- you can't write generic virtual machine code that agnostically handles either pointers or byte data without caring which, even if you pass in a size, as you do with qsort(). But it does seem like it would be workable, and it permits garbage collection and prevents pointer forgery.

The Unix approach

Suppose that a "process" identifies descriptors with integers, like Unix programs identify files, when it makes "system calls"; we could call them "handles". It can never see the contents of the descriptors themselves, just the integers that refer to them in its own local namespace. (KeyKOS did this in practice too, but the integers were in a limited range, I think 0 to 15.) If a different process has access to the same descriptor, it is probably referred to using a different handle.

For accessing byte data, rather than using effectively pread(2) and pwrite(2) as in the proposals above, we can have an mmap(2) instruction which maps the descriptor's byte data into the process's linear memory space. But what about accessing descriptor data, as in `SCM_RIGHTS`, so that one descriptor can point to another? Well, I suppose you want an instruction something like openat(2), but taking an offset rather than a filename.

So this gives us something like the following interface:

• call(code1, handle1): starts a "new process" running code1 and waits for it to terminate. Code1 runs in a new linear memory space with access only to the resource identified by handle1; a handle to that resource is passed to code1 at startup.
• ret(): terminates the current "process".
• open(handle1, offset): returns a newly allocated handle to the offsetth descriptor in the directory identified by handle1.
• new(nbytes): returns a handle to a newly allocated segment of size nbytes.
• del(handle1): deallocates the resource identified by handle1, which may be a segment or a directory.
• mkdir(n): allocates a new directory with space for n descriptors in it.

• link(handle1, offset, handle2): sets the offsetth descriptor in the directory identified by handle1 to the descriptor identified by handle2.

• map(handle1): maps the segment identified by handle1 into the caller's linear memory space and returns the address where this happened.

Maybe not quite as clean as Unix's open, close, read, write, fork, exec, exit, wait, or Forth's C!, C@, !, and @, but it's manageable; and, unlike Unix, it provides full confinement. And it doesn't have a

way to prevent child processes from leaking memory; I thought about adding a "pool" parameter to "new" and "mkdir" and a "spawn" call that creates a child pool, and making "free" take a (handle to a) pool rather than a segment; this would allow limiting the resources used by child processes as well. But it does permit precise garbage collection, so in some sense pools are extraneous.

Of course, unlike in Unix, these operations are virtual machine instructions rather than system calls.

KeyKOS had an operation to weaken a regular key to a "sense key", a read-only capability, so that you could provide read-only access to a resource you had read-write access to.

This interface doesn't permit multithreading, since call() is synchronous, and so it can't be robust in this form against child processes that hang forever. KeyKOS handled this in part by requiring a "clock key" to run a process; if the referenced clock didn't have any time on it, the process couldn't run. The Unix approach is, rather alarmingly, to make subprocess invocation implicitly asynchronous, thus requiring the creation of a new task.

A transactional linear-logic approach

If you add any concurrency or crash recovery to the approach described above, there is a new class of serious potential bugs that the virtual machine cannot detect and signal. If a segment can be concurrently mapped by two different threads and is writable by at least one of them, they can have race conditions. If we were to take the Unix approach and make call() asynchronous, this would implicitly happen on every call(), since the parent process still has access to everything it's passed to the child.

If instead we *transfer ownership* of resources to the newly created child process, so that the parent cannot access them until and unless the child returns them, we can avoid this problem. But this means that, if recovery from failure is to be possible, the child must return them in case of failure and also in case of success.

Handling failures this way suggests that perhaps the child should be run in a separate transaction, with all of its writes held in abeyance until its successful completion. Handling successes this way suggests that perhaps freeing a resource should only be possible to someone who holds a descriptor to the pool the resource was allocated from. But, by itself, that will not prevent the child from linking its resources into a cycle that is inaccessible from outside. Something like the tree discipline of the Unix filesystem is needed to prevent that. See Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) section "Pointer-reversal tree traversal" about why I think approaches like this will tend to be insufficient.

A Rustier approach

If the same segment is mapped more than once by the same process, and one or more of the mappings is read-write, it may suffer aliasing bugs. The classic example of this is the trick for swapping two values without a temporary variable:

```
a ^= b;  // a == a0 ^ b0
b ^= a;  // b == a0 ^ b0 ^ b0 == a0
a ^= b;  // a == a0 ^ b0 ^ a0 == b0
```

which you would want to be a no-op if a and b were the same value, but which instead obliterates the value and replaces it with a 0.

Rust avoids this problem by "borrowing" references for a statically determined lifetime; although my Rust is pretty limited so far, if I understand correctly, the creation of mutable references and of read-only references is a prerequisite to accessing an object, no mutable reference to it can be created during the lifetime of any reference to it, and no reference to it can be created during the lifetime of any mutable reference to it.

You could imagine segments being treated in this way, dynamically rather than statically. To map a descriptor read/write into your memory space or to pass a mutable reference to it to another process or store it in a directory, there would need to exist no references or mappings to it anywhere; to map it read-only or to pass a read-only reference to it to another process or store it in a directory, there would need to exist no read/write mappings or mutable references to it anywhere.

I'm not sure if that approach is feasible, but it seems promising.

Topics

Why Thunderbird is inadequate for opening a 7-gigabyte mbox

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

I opened my mailbox in Thunderbird.

• importing the mailbox is a pain
• had to create a dummy account first
• takes a number of minutes to open the mailbox
• uses 800–950MB of VM and 400+MB of RPRVT memory to open the mailbox
• sorts by default by date message claims to be sent, although there seem to be anomalies; this clusters spam
• uses 7–9 minutes of CPU to render a threaded view, during which time the UI is unresponsive; when you sort by something else and re-enable the threaded view, it takes another 7–9 minutes
• is there a way to hide the junk messages?
• I marked a bunch of messages as spam and a bunch of other messages as read in order to train the junk filter. Then I (actually Beatrice) selected all messages with interesting-feature-A (which took less than a minute of CPU time, during which the UI was unresponsive) and selected "Run Junk Mail Controls on Folder" from the Tools menu. This marked a number of the messages I'd marked as read, but not as spam, as spam. There is no obvious way to distinguish the messages that I have marked as spam and the messages Thunderbird has marked as spam.
• on the plus side, Thunderbird doesn't seem to want to store its updates to message status flags as Status: headers in the messages.
• on the minus side, it seems to be storing that information in a 176MB Mork file, which could make it hard to get it into anything else.
• Despite the fact that Thunderbird isn't storing which-message-is-read information in Status: headers, it seems to believe Status: headers added by spammers.
• Thunderbird thinks I have about 500 000 messages in this mailbox. But I actually have about 800 000.
• Scrollbars are not useful on a 500 000-item list.

Topics

You can't construct optical systems with arbitrary light transfers, but you can do some awesome shit

Kragen Javier Sitaker, 2018-09-10 (11 minutes)

An optical system reversibly transforms a light input to a light output. In the geometrical-optics approximation, the light input (or output) is a function from $\mathbb{R}^4$ (four-tuples of real numbers) to spectra. If we reduce the spectra down to RGB, which is reasonable for some purposes, this is a function $\mathbb{R}^4 \to \mathbb{R}^3$; if we reduce it down to monochrome, which is reasonable for other purposes, it's a function $\mathbb{R}^4 \to \mathbb{R}$. We could think of this as a four-dimensional scalar field. One view of the four dimensions is that they are the X and Y coordinates where light enters (or exits) the system boundary and the θ and φ angles at which it enters. That is, the system can do different things with light that enters at the same angle at different points, or with different angles at the same point.

So, in monochrome, the overall system behavior is a function $(\mathbb{R}^4 \to \mathbb{R}) \to (\mathbb{R}^4 \to \mathbb{R})$. But this is still an extremely loose description, because there are many such functions that we cannot realize as an optical system, and there are others that we can realize only with great difficulty.

(In wave mechanics, the input and output are very much simpler; at a given wavelength, each point on the boundary of the system only has a single complex phase and amplitude. This fact allows holographic optics to achieve astonishing performance for single-wavelength systems. Unfortunately the situation with multiple wavelengths becomes complicated again, while geometric optics either depends not at all on wavelength or only depends on wavelength in a very simple way, with color filters and whatnot. So here I will focus on geometric optics.)

Conservation of energy

For reasons of convenience, I'm going to focus on optical systems that don't dissipate energy, so the amount of light coming out is the same as the amount of light going in. This seems like kind of a stupid focus, since all actual systems do absorb some light, often most of it. Then they convert that light into heat. But it turns out that, if we treat this as a sort of aberration, we can derive some very interesting properties of optical systems that don't have it, and then we can figure out how real systems behave by adding in light absorption as a sort of correction.

Reversibility

Another limitation, given conservation of energy, is that optical systems must be *reversible*. That is, if a certain beam of light going into the system produces a certain distribution of light coming back out of the system, if we send this second distribution of light back in,

it will come out as the first beam of light, just going the other way. This seems not to be true in our day-to-day experience, and this requires some examination. For example, a laser pointer shining on white paint produces a spot that can be seen from any direction, so we know it's throwing off light in every direction, in a way that's called Lambertian reflection, and yet light going into the spot from every direction doesn't go back into the laser pointer. We explain this by saying that the paint is full of many different microscopic facets, each of which throws off light in a particular direction when the laser hits it, and there are so many of them even in that little spot that the light seems to go in every direction at once! And if we could shoot a very, very thin beam of light at each facet, in just the reverse of the direction that the laser was making it shine before, all of those beams would be redirected perfectly back to where the laser pointer had been.

Now, in reality, these facets are often so tiny that the geometrical optics approximation breaks down, and we have to use wave mechanics to see what will happen. But it turns out that wave mechanics is reversible too; reversibility is not just a consequence of the geometrical-optics approximation, but a property of the wave-mechanical nature of light that survives in the geometrical-optics approximation.

But if our system consists entirely of macroscopic features — mirror-smooth surfaces that are perhaps curved or have edges, everything either polished metallic mirror-bright or transparent — then, indeed, any transformation that the system produces can in fact be time-reversed in this way. And you can do it in practice, not just in theory, because you don't need microscopically tiny slivers of light the way you do for the white paint.

This imposes some restrictions on the mathematical form of our system. It can't, for example, transform two different distributions of incoming light into the same distribution of outgoing light, because then if you time-reversed the outgoing light, it wouldn't know which of these two different distributions it should produce. The function must be bijective, invertible.

But is that the only restriction? Can we realize any arbitrary invertible $(\mathbb{R}^4 \to \mathbb{R}) \to (\mathbb{R}^4 \to \mathbb{R})$ function as an optical system? No, not even close.

One of the strongest restrictions is linearity.

Linearity

Most optical systems are linear, in the sense that different beams of light don't interact with each other. If you have some beam A and the system transforms it to f(A), and some other beam B and the system transforms it to f(B), then if you shoot both of those beams of light at the system at once, A + B, then the distribution of light that comes out will be exactly f(A) + f(B). You can have a lens, for example, bend one light beam a bit to the left, and the other a bit to the right, but you can't have it bend the first light beam to the right when the second one is present, or to the left otherwise.

Now, this is just an approximation, but under most circumstances, it's a very, very good approximation, and it takes very sensitive instruments to detect departures from linearity. It's actually a much better approximation than geometrical optics is, because you can see

the diffraction phenomena produced by wave mechanics very easily in everyday life, if you know where to look; they're quite strong whenever you have objects on the scale of a few microns involved, such as your eyelashes. They're rarely more than one or two orders of magnitude away from visibility. Departures from linearity, by contrast, are usually six or more orders of magnitude away from visibility. So nonlinear optical systems are substantially more difficult to build.

There are a few that are common, though. Fluorescence is usually pretty linear, but it often has a substantial time constant, which means that it departs from instantaneous linearity. Optically-pumped lasers, however, are a sort of nonlinear fluorescence phenomenon: you don't get a laser beam at all until the gain of the lasing medium rises past 1, as limited by the Q of your cavity. And the most common kind of green laser isn't a green laser at all; it's an infrared laser with a frequency-doubling crystal on the front of it, and that's a nonlinear phenomenon — it doesn't start happening until the light intensity is above a certain level.

Other nonlinear optical phenomena include phase-conjugating mirrors, Kerr cells, the self-focusing of intense laser beams, and soliton transmission, which is a sort of temporal analogue of spatial self-focusing. Any dielectric inevitably behaves nonlinearly to light passing through it, since its overunity refractive index is due to its response to the electric field of light being different from the response of the vacuum, and that's an effect that inevitably reaches a limit at some field strength. Normally, though, light's electric field is far too weak for us to notice this nonlinearity.

But, in the geometrical optics approximation, we invariably ignore these nonlinearities, because they are tiny in everyday life. So our transfer function is, in effect, transferring every separate light beam that could enter our apparatus into some distribution of light at the output. So our transformation function can be computed from a sort of point spread function of the form $\mathbb{R}^4 \to (\mathbb{R}^4 \to \mathbb{R})$.

However, the requirement that the function be *reversible* means that as the input light beam shrinks toward a perfectly collimated beam entering at a single point†, the output light beam must *also* shrink toward being such a thing, except perhaps at discontinuities. So it's actually even simpler, and this is a simple case of a more general principle called "conservation of étendue".

XXX is this really correct?

† For wave-mechanical reasons you can't actually make a perfectly collimated beam entering at a single point — there's a diffraction limit on the divergence — but here we're talking about properties of the geometric-optics approximation.

Conservation of étendue

Étendue is a quantity that

Translation-invariance

Electroforming and Electropolishing

Electric current passes through a battery electrolyte not as free electrons, as in a metal, but as positive metal ions, and this is true

whether you're charging or discharging the battery. The positive ions are formed from the metal at the surface of the positive electrode, which has electrons running away from it down a wire, through a circuit, and back around to the negative electrode, where they travel to the surface of the metal and neutralize arriving positive ions, thus transmuting them back into insoluble metal.

You can use this process to coat some random conductive thing with a layer of metal, which is called galvanizing or electroplating — or electroforming, if you do it long enough — or to remove a thin layer from the surface of a piece of metal, which is called electropolishing, or cathodic corrosion if you do it by accident, like on a metal ship hull.

Because electric fields are strongest around edges and sharp points, electropolishing tends to remove those, leaving a mirror-like finish on initially rough metal. Also, since it doesn't

At the currents typically used, this process typically deposits around a nanometer per second of metal on one electrode and removes around a nanometer per second from the other. Much lower or higher currents don't work as well.

Fresnel electropolishing

Holographic electropolishing

https://en.wikipedia.org/wiki/View_factor
https://en.wikipedia.org/wiki/Lagrange_invariant
https://en.wikipedia.org/wiki/Etendue

Topics

• Physics (p. 3632) (119 notes)
• Optics (p. 3609) (34 notes)
• Caustics (p. 3368) (6 notes)

Cristina Fernández de Kirchner tweets about the attempt to kidnap Assange

Kragen Javier Sitaker, 2014-04-24 (3 minutes)

Volví de la Rosada. Olivos, 21:46 hs. Me avisan, Presidente Correa al teléfono. "Rafael?. Pasámelo".

"Hola Rafa, cómo estás?". Me contesta entre enojado y angustiado. "No sabés que está pasando?"

"No, que pasa?". Yo en babia. Raro, porque siempre estoy atenta… y vigilante. Pero recién había finalizado una reunión.

"Cristina. Lo han detenido a Evo con su avión, y no lo dejan salir de Europa".

"Qué? Evo? Evo Morales detenido?" Inmediatamente me viene a la mente su última fotografía, en Rusia…

Junto a Putín, Nicolás Maduro y otros Jefes de Estado. "Pero que pasó Rafael?"

"Varios países le revocaron el permiso de vuelo y está en Viena", me contesta.

Definitivamente están todos locos. Jefe de Estado y su avión tiene inmunidad total. No puede ser este grado de impunidad.

Rafael me dice que va a llamar urgente a Ollanta Humala para reunión urgente UNASUR.

Llamo a Evo. Del otro lado de la línea, su voz me responde tranquila: "Hola compañera, como está?". El me pregunta a mí como estoy!

Me lleva miles de años de civilización de ventaja. Me cuenta la situación. "Estoy aquí, en un saloncito en el aeropuerto…"

"Y no voy a permitir que revisen mi avión. No soy un ladrón". Simplemente perfecto. Fuerza Evo.

CFK: "Dejáme que llame a Cancillería. Quiero ver jurisdicción, Tratado y Tribunal al cual recurrir. Te vuelvo a llamar". "Gracias compañera"

"Hola, Susana". No querido, Susana Ruiz Cerruti. Nuestra experta en legales internacionales de Cancillería...

Me confirma inmunidad absoluta por derecho consuetudinario, receptado por Convención de 2004 y Tribunal de La Haya.

Si Austria no lo deja salir o quiere revisar su avión, puede presentarse ante la Corte Internacional de La Haya y pedir…

Siiii!, UNA MEDIDA CAUTELAR. No se si ponerme a reír o llorar. Te dás cuenta para que son las medidas cautelares.

Bueno, sino le podemos mandar algún juez de acá. Madre de Dios! Qué mundo!

Lo llamo a Evo nuevamente. Su Ministro de Defensa toma nota. En Austria son las 3AM. Van a intentar comunicarse con las autoridades.

Hablo con Pepe (Mujica). Está indignado. Tiene razón. Es todo muy humillante. Me vuelve a hablar Rafa.

Me avisa que Ollanta va a convocar a reunión de UNASUR. Son las 00:25 AM. Mañana va a ser un día largo y difícil. Calma. No van a poder.

Topics

- History (p. 3500) (71 notes)
- Español (6 notes)
- Wikileaks (p. 3775) (2 notes)

A simple virtual machine for vector math?

Kragen Javier Sitaker, 2018-11-06 (updated 2018-11-09) (15 minutes)

Could you design a simple virtual machine for vector math that a wide range of existing and future vector hardware (SIMD instruction sets, GPUs, FPGAs) could execute efficiently? A sort of equivalent for C's virtual machine (viewing C as a virtual machine!) but aimed at things that can execute efficiently across a wide range of today's computers, not 1984's — a lowest common denominator that's low enough to get substantial parallelism on almost anything, but not so low that typical code is as slow as typical C code.

I think it's feasible.

Of course, nonportable code will always be faster, but the idea of the vector VM is to be fast enough to use portable code most of the time.

I was originally thinking of this approach as part of an archival virtual machine proposal, a successor to Chifir sufficiently improved to be workable, whose raison d'être was archaeological in nature — the objective is for an archaeologist finding the artifact to be able to implement the specified virtual machine as a fun afternoon hack, then load the ancient data into it and breathe life back into it. I thought maybe a pipelined vector design could work, but upon further consideration, I decided it wasn't well suited to compatible reimplementations without reference to an existing working implementation.

Still, though, I think it might be interesting for less demanding purposes, the sort of thing OpenCL and Vulkan are aimed at. It's sort of a microscopic SPIR-V with about 30 operations, or a VM for Wadge's Lucid.

Unfinished notes

Old Cray machines weren't parallel-SIMD, where you could add, subtract, multiply, take the square root, or whatever of each of 4 or 8 or 16 values in a single cycle; instead they had some 8 "vector registers" of, say, 64 single-precision floats, and vector operations on them which operated elementwise. But the machines didn't have 64 floating-point execution units; they might have 6. So a vector-to-vector operation necessarily took place one float at a time, but pipelined, with one result per cycle. Moreover, you could "chain" operations from one register to the next, and even to main memory. So, for example, you might compute mx + b, for vector values of the variables, as follows:

```
vectorload m_addr, V1
vectorload x_addr, V2
vectorload b_addr, V4
vectormul V1, V2, V3
vectoradd V3, V4, V5
vectorstore V5, y_addr
```

So with six instructions you would start the computation of the values, and then it might take 64 more clock cycles to finish. This is how the Cray-1 got 138 megaflops, bursting to 240 megaflops, in 1977, on an 80MHz clock.

It's also very similar to how Numpy gets reasonable performance for numerical code despite being an extension for dog-slow CPython. This, plus Python's "iterable" interface (now adopted by JS and Java), suggests the approach of computing abstract sequences of values (of some arbitrary length) rather than individual values or vector values of some fixed length, and including a stride argument in the vector memory fetch instruction.

Consider a matrix-matrix multiply C = AB. Here, in C-style row-major form (which is backwards from standard mathematical notation) C[i][j] = Σ_k A[i][k] B[k][j], or $C_{ij} = \Sigma_k A_{ik} B_{kj}$; writing that as nested loops:

```
C = {0}; // everywhere (this is not valid C syntax)
for (int i = 0; i < A_rows; i++) {
    for (int j = 0; j < B_rows; j++) {
        for (int k = 0; k < A_cols; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Thus written, the code is precisely equivalent if we swap inner and outer loops into any of the 6 permutations. As written, it totals each item of C separately, iterating over a column of A and a row of B for each one; but we could also, for example, multiply a cell of A by a row of B, sending the results to the corresponding row of C, merely by swapping the inner two loops:

```
C = {0};
for (int i = 0; i < A_rows; i++) {
    for (int k = 0; k < A_cols; k++) {
        for (int j = 0; j < B_rows; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

That is, $C_i = \Sigma_k A_{ik} B_k$.

We could turn the inner loop here into a program that reads a vector from memory, multiplies it by a scalar (previously loaded from A), reads another vector from memory (the row of C), adds the product to it, and writes it back where it came from.

In the case where C has many more rows than columns, though, it would be more efficient to write a column at a time. This is the same kind of simple transformation:

```
C = {0};
for (int j = 0; j < B_rows; j++) {
    for (int k = 0; k < A_cols; k++) {
        for (int i = 0; i < A_rows; i++) {
            C[i][j] += A[i][k] * B[k][j];
```

```
      }
    }
}
```

That is, C[...][j] = Σ_k B[k][j]·A[...][k]. Here we are updating a whole column of C by adding to it the product of a scalar from B and a column of A.

This is of course not optimally cache-friendly (something the Crays didn't have to worry about) but it still might be a win if C has a sufficiently small number of columns.

Building a dataflow network on the stack

So suppose that we have a stack machine equipped with an additional stack where the items are not numbers but streams of some finite but possibly large length. By manipulating these streams, we can set into motion highly parallelizable computations which may involve orders of magnitude more computation than that required to interpret the instruction stream. The virtual machine does not know about multidimensional arrays, but it can efficiently execute some computations on them.

The crucial thing to note here is that the stream operations here treat different indices in the stream as completely separate; there is no way for anything that happens at index 0 in the stream to affect anything that happens at index 1 in the stream, or vice versa. So the execution of the entire stream is embarrassingly parallel. Whether that's useful or not depends on what you're trying to compute.

Sources

There are two operations which create a new sequence: `load(address, mode, stride, count)` and `iota(start, stride, count)`. (They can take their arguments from, for example, a different stack.) `load` produces a sequence of `count` items loaded from memory using `mode` (e.g. `uint32_t` or `float*8`) located `stride` bytes apart; if `stride` is 0, then all of them will be copies of the same value. `iota` similarly produces a series of `count` items; the first will be the number `start`, and succeeding items will increase by `stride`, which may be 0.

The span of addresses that will be accessed by `load` can be bounds-checked as soon as the `load` instruction executes, allowing the bounds check to be amortized over a potentially very large number of data accesses.

Sinks

There are correspondingly a few different final destinations we can send these streams to.

`store(address, mode, stride, count)` is the counterpart of `load`; it stores the values of a stream into memory. With `load` and `store` we can copy blocks of data around memory, translate it between different numeric formats, and transpose matrices by using different strides.

`argmax` and `argmin` each take two streams as arguments: an `index` stream and a `data` stream, which should be of the same length. When the streams are exhausted, they yield the `index` value corresponding to the largest or smallest datum from the `data` stream. If the `index` and `data` streams are the same, they thus produce the max or min of the stream.

`sum`, similarly, computes the sum of a stream.

Stream transformations

However, there are also a large collection of operations that operate elementwise on streams. For example, the `~` operation computes the bitwise NOT of the values in a stream, consuming the stream and producing a new stream of the altered values. The `+` operation consumes two streams (which should be of the same length) and produces a new stream of the sums of corresponding values from those streams.

The collection of stream transformations consists of the following 16 operations:

- `+`;
- unary `-`, which computes the numeric negation of each item in the stream;
- binary `-`, which subtracts streams elementwise;
- `*`, which multiplies streams elementwise;
- `divmod`, which consumes a stream of dividends and a stream of divisors, and produces a stream of floored quotients and a stream of nonnegative remainders;
- `<<`, which consumes a stream of quantities and a stream of bit shift distances, performing an arithmetic left shift on each quantity;
- `>>`, which is the equivalent right-shift — a logical right shift for unsigned quantities, an arithmetic right shift for signed;
- `|`, which computes the bitwise OR of two streams;
- `&`, which computes the bitwise AND;
- `^`, which computes the bitwise XOR;
- `~`;
- `\`, which computes the bitwise abjunction `(a & ~b)`;
- `<`, which consumes two streams and produces a stream of boolean values which is true when the value from the first stream is less than the corresponding value from the second;
- `=`, the same except that it tests for equality;
- `?:`, which consumes a stream of booleans and two streams of quantities, producing a single output stream; its values are quantities from the first stream when the booleans are true, and from the second stream when the booleans are false;
- `⌜`, which combines two streams to produce a stream of values that come from the first stream when those are greater, or from the second stream when those are.

Stream stack manipulation

You also have the usual collection of simple stack operations for the stream stack — `dup`, `drop`, `swap`, `over`, `rot`, `>R`, `R@`, and `R>`.

These are particularly important here because of their effect on the stream execution engine. While references to a dataflow network are live on the stack, the engine doesn't yet know what use will be made of them; it is only once the last reference has been removed from the stack that it can actually execute the requested computation. If they are `drop`ped, perhaps it need not do any computation at all.

XXX what happens if you request the sum or argmax of a stream while other references are still live on the stack? Maybe just eliminate `dup` and `over`, because the alternative seems to be unbounded buffering.

Why a stream stack

Why use a stack machine for building the streams instead of, say, a register-based virtual machine? Don't register-based bytecodes run faster?

The advantage of a stack machine is that the number of uses of each value is explicit, while in a register machine, it is implicit. If you have no `drop` operation or other way to discard a value, you know that all values are used; if you additionally have no `dup` or `over` or `R@` operation or other way to duplicate a value, you know that each value is used exactly once. The point at which no more references to a particular tree exist on the stack is precisely the point when it will not be used in the future.

Perhaps you could design a two-operand register machine with the same property, if each register has an "empty"/"full" bit like the memory words of the Tera MTA, such that using a register as a "source" operand optionally "empties" it, and reading from "empty" registers or moving to "full" registers is not permitted.

But a stack machine seems like a much simpler route to the same goal.

Okay actually fuck all this

So what you actually want is to construct a dataflow graph from a sequence of inputs to an equal-size sequence of outputs. And you want to do that while ensuring that your dataflow graph is acyclic, not only in the directed form but in the undirected form as well. (There's no good reason for that restriction, but I did talk about imposing it, at least possibly.) And then you want to launch it to go process stuff and then maybe get results.

In that case, wouldn't it be best to separate the graph construction and graph execution steps? And in that case, wouldn't it be best to be able to execute the same graph more than once? In my matrix multiplication example, for example, you want to run your inner loop m×n times (e.g. `A_rows` × `A_cols` times). There's no benefit to reconstructing the graph again every time through the inner loop!

Separating them also eliminates the problem of making sure all the lengths are equal — you supply the length, along with the input and output data addresses and strides, when you invoke the graph. And it eliminates the question of what to do when you invoke the `sum` instruction but there are still things left on the stack, because the sum won't be delivered at graph construction time in any case, but at graph invocation time.

(And in that case maybe when you instantiate the sum node, you give it a destination register, so that you can have several of them in the same graph.)

Getting back to the question of "a sort of equivalent for C" — is a stack-based virtual machine really the same level of programming as C? Wouldn't you want infix syntax for programming at C level? Why do you need some kind of bytecode representation of your vector dataflow graph? Wouldn't you be better off with a language extension for C, or maybe even a C library? And in that case, wouldn't you be better off with explicit node labels of some kind instead of this dopey stack-machine interface? I mean, it would be a lot easier to read your code, probably, even if infix is too hard to provide because C doesn't do operator overloading.

Topics

Vitruvius could have taken photographs

Kragen Javier Sitaker, 2016-07-30 (1 minute)

It turns out that in classical Rome (and possibly earlier in Lydia) the means for refining gold from electrum was to smelt the electrum with salt to oxidize the silver to silver chloride. (The modern nitric-acid process wasn't possible until nitric acid was produced; it was available by the 13th century, but wasn't available in classical Rome.)

Silver chloride is one of the silver halides used in photography; it can be used by itself. Silver chloride plus gelatin is by itself adequate to produce negative prints on paper (or papyrus) given enough light and time.

The Romans had gelatin, too; the widespread use of hide glue dates from Middle Kingdom Egypt. So they had the materials technology necessary for photography. We could be looking at Julius Caesar's hobby pinhole landscape photographs today if they had known about the properties of the materials they had at hand.

(Actually, I'm not 100% sure that the collagen in hide glue is sufficiently similar to gelatin; gelatin is collagen hydrolyzed either simply by boiling or more quickly with acids, bases, or proteolytic enzymes.)

Topics

Fencepost cognitive interface errors in text editing

Kragen Javier Sitaker, 2019-04-24 (24 minutes)

A substantial part of the work of editing natural-language text amounts to rearranging small pieces of it, and these pieces most frequently correspond to natural units, such as words, clauses, prepositional phrases, sentences, and paragraphs. Existing text editors fall far short of the optimum in this.

Emacs has commands for moving by and deleting characters, words, sentences, lines, S-expressions, paragraphs, and in some cases, pages. You would think that this would make it easy to do sequences like "go back five words, delete two words, and then insert them at the end (suitably defined) of the sentence." But if you attempt to apply that sentence to the end of itself, this is what you get:

"go back five words, delete two words, and then insert them at the end () of the sentence."suitably defined

Rather than, as you might hope:

"go back five words, delete two words, and then insert them at the end of the sentence (suitably defined)."

Similarly, you might hope that backward-word (M-b in Emacs parlance) would take you to the same position as backward-word backward-word forward-word (M-b M-b M-f), at least if there was a second word to move backwards over. But, as far as I know, it never does; the first sequence leaves you at the end of a word separator, while the second sequence leaves you at its beginning. Consequently, after the second sequence, you can use kill-word (M-d) to delete one or more words including any leading separators, which you can then insert again coherently at a position before another word separator. For example, starting from

can |then insert again coherently at

you can use M-f M-d M-d M-f C-y to get to this desirable state:

can then coherently insert again| at

You could make the same edit from this different starting position:

can then insert again coherently| at

by using C-f M-DEL M-b M-b C-y:

can then coherently |insert again at

M-DEL is backward-kill-word, which, unlike M-d, includes any *following* separators; thus the necessity for the initial C-f to avoid this mess:

can then coherently|insert again at

This requires slightly nontrivial cleanup, and the situation is worse in programming languages. Transient-mark mode, M-@, M-h, and C-M-SPC can help significantly, but they don't really solve the problem.

The underlying difficulty here is that in Emacs, a sentence is not a sequence of words; it is a sequence of alternating words and word separators, which contains nearly twice as many elements as words. Since it would take twice as long to get anywhere if the movement commands moved by such elements, they move past the large

elements, and you can use character-movement commands to fine-tune. Unfortunately, this leads to a lot of fine-tuning, user errors when you get the fine-tuning wrong, and unnecessary round trips to make sure the fine-tuning is right.

The same difficulty attends Emacs's commands for sentences and S-expressions, though typically not lines and paragraphs.

lowriter

LibreOffice Writer clones Microsoft Word's text editing command set, which in its turn is copied from early Macintosh, and thence from Bravo and Smalltalk. Its word movement commands (Ctrl-← and Ctrl-→) take you to the beginnings of words or the beginnings of non-space strings between words, and they are the same positions in both directions. Its word deletion command, Ctrl-Backspace, deletes back to these same positions, except that it also has stopping positions at the beginnings of paragraphs, even empty ones. (Also, lowriter considers embedded apostrophes, though not trailing apostrophes, to be parts of words; similarly, embedded commas are parts of words if they have digits on both sides of them, though periods are not.) It doesn't seem to have commands to move by sentences, although it does have movement by paragraphs (Ctrl-↑ and Ctrl-↓). This works better than Emacs's word-movement approach in most cases, but not all. The first example above, commanded as as Ctrl-← Ctrl-← Ctrl-← Ctrl-Shift-← Ctrl-Shift-← Ctrl-Shift-← Ctrl-Shift-← Ctrl-x Ctrl-→ Ctrl-→ Ctrl-→ Ctrl-v, yields this:

"go back five words, delete two words, and then insert them at the end of the sentence(suitably defined) ."

But the second example:

commanded as Ctrl-→ Ctrl-Shift-→ Ctrl-Shift-→ Ctrl-x Ctrl-→ Ctrl-v, comes out fine, though slightly different from Emacs:

can then coherently insert again |at

And the third example, which logically should have the same problem as Emacs, when commanded as Ctrl-Shift-← Ctrl-x Ctrl-← Ctrl-← Ctrl-v:

can then insert again coherently| at

instead also comes out fine; the space to the left of "coherently" was deleted immediately after the word itself, and upon being inserted at the beginning of "insert", a space is appended to preserve the pre-cut-and-paste word boundaries:

can then coherently |insert again at

If you paste into the middle of a word, spaces are inserted both before and after; if you paste into the middle of whitespace, spaces are inserted neither before or after. If you paste between a sentence-ending word and the sentence-ending punctuation, space is inserted before the pasted word, but not after; no corresponding cleanup seems to exist for pasting to the beginning of a sentence. However, if the string you deleted was "coherently ", with the trailing space, as it would be if you used the word movement commands to find both ends of the selection, or if it was "coherentl", not touching the final word boundary, none of this magic DWIM whitespace behavior happens.

I suspect that the reason for the DWIM behavior is that, in lowriter, if you select words by double-clicking with the mouse,

instead of using Ctrl-Shift-← and Ctrl-Shift-→, the space after the word is not included in your selection!

Also, in lowriter, triple-click selects by sentences, but doesn't include whitespace surrounding the sentence; this means that if you rearrange a sequence of sentences by triple-click and drag-and-drop, you have to clean up whitespace afterward, in a way exactly analogous to the Emacs word-separator problem described above. Quadruple-click (!!) selects paragraphs, but only single paragraphs; dragging outside the paragraph reverts to selecting by characters.

Vim

Vim's word movement commands (w and b, not e and ge) seem to be entirely consistent with lowriter, except that they treat apostrophes and commas like any other punctuation; and deleting, cutting, and pasting is consistent with movement, though of course without DWIM.

When programming in Emacs, I've often been frustrated by its word-movement commands skipping over long sequences of punctuation and whitespace as if they weren't even there. Vim's command structure is significantly better in this regard.

Eclipse

Eclipse's word movement commands have the same problem as Emacs's.

New editor design thoughts

What should the command set for a new editor look like?

Established movement-command convention

Assuming that MacOS and Microsoft Word are consistent with lowriter, violating their word movement and deletion command convention is costly; Vim's behavior is also consistent with it, and the behavior seems to be both more predictable and more frequently what is desired without DWIM magic. But the fact that lowriter does resort to DWIM magic behavior under fairly normal circumstances suggests that perhaps a better alternative is possible.

Selection-verb versus Emacs or Vim commands

The Smalltalk and Bravo selection-verb convention, imitated in Macintosh and in nearly all subsequent GUIs, of first selecting a region of text visibly and then applying subsequent commands to that region of text, is substantially more predictable than the Emacs and Vim approach of applying commands to regions computed after the command starts. Both the selection-verb convention and the Vim approach enjoy substantially better orthogonality than the Emacs command set, although by the same token the Emacs commands are often shorter:

```
|          | forward  | backward | mark              | delete/kill    | backw
ard delete/kill    |

| char     | C-f      | C-b      |                   | C-d            | DEL
                |
```

```
| word      | M-f      | M-b      | M-@                  | M-d           | M-DEL
, C-backspace      |

| sentence  | M-e      | M-a      | mark-end-of-sentence | M-k           | C-x D
EL                 |

| line      | C-p      | C-n      |                      | C-k (XXX),    |
                   |

|           |          |          |                      | C-S-backspace |
                   |

| paragraph | M-}, C-↓ | M-{, C-↑ | M-h                  | kill-paragraph | backw
ard-kill-paragraph |

| sexp      | C-M-f    | C-M-b    | C-M-SPC              | C-M-k         | ESC C
-backspace         |
```

Of these 27 different keystroke commands, I think only 16 are in my own subconscious repertoire, and that's after 30 years of this body using one or another Emacs. Some of those work in readline, or GTK (by default), while others don't, adding to the difficulty.

Ctrl-backspace

Ctrl-backspace is extremely valuable, more valuable even than backspace, because it eliminates not only keystrokes but also feedback round trips. (I mistyped that "trips" as "tirps", for example, and then used Ctrl-backspace to delete it and type the correct word.) Replacing Ctrl-backspace with a selection-verb sequence like Ctrl-Shift-← Ctrl-x (which does work in lowriter, for example) would be intolerable, unless both the selection and the verb were a single keystroke, could be released in an indeterminate order, and supported repeating without fiddly key alternation.

Multilevel cut and paste

Emacs's killing and C-y and M-y behavior is also extremely valuable: it means you don't have to decide ahead of time which deleted text you are going to want to paste later, and you don't have to worry about losing text on the clipboard by cutting something else. Every mainstream UI has supported multilevel undo for over 20 years, eliminating the danger of losing your undo information with a following command (the famous modal UI "edit" problem, in which "e" selected "everything", "d" deleted it, "i" went into insert mode, and "t" was inserted, losing the undo info.) It's long since time they supported multilevel cut and paste!

The Back button

Emacs's C-u C-SPC command, which pops the mark stack, is extremely useful for returning to editing what you were just editing before your last search; it's like a WWW browser "back" button, but inside a single editor buffer. It really needs a more convenient keybinding, one that can be repeated without key alternation. Browsers bind it to Alt-← and, historically, Backspace, although that's going away, but something you can type without taking your

fingers off the home row would be superb.

Text movement

Eclipse's Alt-↑ and Alt-↓ keybindings for moving a block of lines (either the current line, or the selected block, expanded to full lines) are one of the few code-editing features in Eclipse that I miss in Emacs and Vim. They don't work as well for text in paragraphs, since it tends to have logical boundaries that don't coincide with line boundaries, but you could imagine a similar command set that did work well; lowriter, for example, permits drag-and-drop rearrangement of text fragments with the mouse, although it doesn't provide ongoing feedback about the final result.

Emacs has a set of "transpose" commands which can in theory be used for this, but they are awkward to use: they only move single units (of whatever size: lines, sentences, sexps, words, chars — but not blocks of two lines or words or whatever), only forward, and only two of them have default keybindings that can be repeated without alternating keys (C-x C-t is transpose-lines, which doesn't take advantage of the transient map feature in recent versions of Emacs that, for example, allows C-x e e e to run the last keyboard macro three times.)

Movement via incremental search

Incremental-search, which originated in Emacs in the 1970s is extremely valuable, both for the usual search purposes and for cursor movement, as Jef Raskin showed in his work (on the Canon Cat, SwyftWare, The Humane Environment, and Archy). It's an even faster means of cursor movement than using the mouse. Unfortunately, Emacs's implementation of incremental-search is crippled for cursor movement in a few different ways:

• When searching *forward*, Emacs incremental-search leaves the cursor at the *end* of the search string, not its beginning. That means that if you want to move the cursor from "extremely" above to the start of "beginning" in this paragraph, you can't simply search for "beginning"; you'll end up somewhere in the middle of the word or at its end. Instead you must search for "its ", and the search is no longer effectively incremental — even though "it" is sufficient to uniquely identify the string you're searching for, you have to keep typing to get to the place you want. Alternatively, you can search for "beginning" and then issue an extra command to get from the middle to the beginning of "beginning"; this requires typing only up to "be".

• Emacs's incremental-search is modal, requiring an extra keystroke to terminate it. This can result in mode errors, where later keystrokes are erroneously used as part of the search key, but the larger problem is that it is a significant amount of overhead. The Canon Cat, for example, used its quasimodal "LEAP" to move by paragraphs by searching for paragraph breaks, or by "sentences" by searching for periods; releasing "LEAP" ended the search. Thus "search for the next newline" was "LEAP-Return", releasing "LEAP" after "Return", the same number of keystrokes as Emacs's M-a or C-e; it's a sequence of three events. Using C-s . RET in Emacs is, by contrast, a sequence of six events: Ctrl down, S down, release all, period

down, RET down, release all. A benefit of being modal in this way is that the same C-s and C-r keystrokes can repeat the search in different directions.

• As an additional problem, the particular keystroke Emacs chose to terminate incremental-search is RET or Enter. This makes it unintuitive and inconvenient to search for newlines (you can do it by typing Ctrl-J, although I don't know if I knew that until I tried it just now).

Vim's incremental search avoids problem #1, but still has problems #2 and #3, and actually problem #3 is even worse: /^V^J RET in Vim searches for NUL rather than LF. Searching for an actual linefeed in Vim requires /\n RET.

A separate tweak to text incremental search is to display further search hits in context in a separate pane once they're infrequent enough, allowing a smooth transition from in-context navigation to menu navigation.

Multiple cursors

The proprietary Sublime Text editor's multiple-cursor feature seems like a better alternative to keyboard macros under most circumstances, because it allows you to see the effects of all the "iterations" of your "macro" incrementally as you are composing it. But I'm getting off on a tangent.

Selection command design

Most of that is preamble to say that I think the selection-verb model is generally better, but you need sufficiently powerful selection commands to keep that model from being too clumsy. For rearranging text by word, sentence, or paragraph, perhaps you could use a command like Emacs's M-@, which extends the selection to the end of the next word, thus allowing you to select multiple words by repeating the command — but with semantics more like Emacs's M-h, which also extends the beginning of the selection to the previous paragraph boundary, if it isn't already there (except that it does the wrong thing if there's already a selection which wasn't created by M-h). So, in Emacs, M-h selects one paragraph, M-h M-h selects two paragraphs, and so on; a new editor could have keybindings that operate similarly.

Three such keybindings — select-word, select-sentence, and select-paragraph — would allow you to quickly select the span of text you want to operate on, if it's less than, say, four logical units long. Making longer selections would be faster with the mouse or using incremental search, requiring the user to strategize as to how to make the selection, and potentially causing long decision times around the break-even point due to the problem of Buridan's Ass; to the extent that we can avoid imposing the need to strategize on the user, we can shorten the path to the user's desire. (The kfitzat haderech is the ultimate user interface!)

S-expression selection is somewhat trickier because there are at least two potential directions the user might want to expand their selection: forward to the following expression, and up the S-expression tree. So a single keybinding for marking sexps will not suffice. Thus Emacs provides forward-sexp (C-M-f), backward-sexp (C-M-b), mark-sexp (C-M-SPC, which extends the region

forward), down-list (C-M-d), backward-up-list (C-M-u), up-list, kill-sexp (C-M-k), backward-kill-sexp (ESC C-backspace), and kill-backward-up-list. Maybe the right solution is to repurpose the select-word and select-sentence keybindings in programming-language modes.

Perhaps mouse selection can give an indication of the desired granularity of selection by its speed: if the mouse is moving too fast for the user to have intended to select at the granularity of a single character or even a single word, perhaps sentence or even paragraph granularity was intended. If the mouse selection starts out with the incorrect granularity, the user can correct it by moving the mouse faster or slower as they drag out the rest of the selection. This mechanism could smoothly incorporate probabilities from stochastic models of natural language to permit sub-sentence selections, including things like dependent clauses and prepositional phrases.

Delimiter behavior

But you still have the problem of the fenceposts or delimiters: when are they included in the selection and when are they excluded?

If there's an improved model of text that permits a better user interface for rearranging bits of it, I suspect it looks something like, "Paragraphs are sequences of sentences. Sentences are sequences of words." This avoids the fencepost problems described earlier, where you move by words, cut by words, and then paste, and yet you end up merging words or creating extra spaces. This is not too far from the model implicit in the Vim command set, where w and b take you to the beginnings of words or punctuation strings.

The sequence-of-words model does potentially have the problem lowriter's DWIM is trying to patch up: a sequence of words in a different context might need different word separators around them. For example, if I were to replace "around them" in the previous sentence by moving "in a different context" to the end of the sentence, including a space after that phrase would be inappropriate.

A possible solution to this problem might be something like the following data model:

- Words are nonempty sequences of alphanumeric characters, with possible exceptions for things like apostrophes and commas in numbers.
- Words are always terminated by word terminators, which are inserted automatically if not present, as Vim does with trailing newlines in a text file. All word terminators terminate nonempty words.
- Word terminators followed by words are displayed (and, in plain text, serialized) as spaces.
- Word terminators followed by punctuation or paragraph breaks are not displayed (or serialized in plain text).
- Spaces following other spaces or punctuation — anything that isn't a word — represent real spaces, not word terminators.
- Paragraph breaks are always preceded by a word terminator or real space, which is not displayed (or serialized in plain text).

This provides a bijective mapping between displayed character sequences and internal character sequences. In one direction, spaces following alphanumerics and preceding either spaces or

alphanumerics are converted to word terminators, and word terminators are inserted before punctuation that follows alphanumerics; in the other direction, word terminators are converted to spaces unless they precede punctuation.

In terms of commands:

- Word movement commands move to points which include the beginnings of words, beginnings of paragraphs, and beginnings of punctuation strings.
- Mouse selection of words selects between the same points word movement commands move to, and thus includes the fucking word terminators.
- Movement by sentences moves to the beginnings of sentences. The sentence terminator pattern
- Selection by sentences includes the sentence-ending punctuation, any following punctuation before space, and any following spaces, including the invisible space preceding a paragraph break, but not the space that often precedes the beginning of the sentence; that space belongs to the previous sentence. It also includes any punctuation at the beginning of the sentence following spaces that belonged to a previous sentence, such as an open parenthesis or open quote.

Like lowriter's approach, this provides the right behavior for cutting and pasting sequences of words from the ends of sentences to the middles and from the middles to the ends; however, it does so in a way that is predictable because it is consistent with its data model, rather than being a DWIM special case. Unlike lowriter's approach, it also provides the right behavior for cutting and pasting sentences from anywhere in a paragraph to any other sentence boundary in a paragraph. Like lowriter's approach, it still requires manual cleanup for cutting and pasting sequences of words to or from the beginnings of sentences. It differs from lowriter's approach when you select a word (or several words) with the mouse and either copy or move it to the interior or end of another word: in that case, it will paste with a space after it but not before, as lowriter does when you make the same selection using keyboard selection commands.

How well will these rules apply to programming languages, should you be editing a program in natural-language mode? Since the mapping between the internal representation and the serialized one is bijective, it shouldn't pose any particularly serious problems.

Scoring

I think the editor should use game mechanics to teach you to use its command set in the way we have Tayloristically determined to be the most effective.

Autosuggest

The editor should use a neural network pretrained on a corpus of existing open-source projects and natural-language text, then later trained on the things you actually write in it, to order and possibly even add autocompletion suggestions. It can also adjust the frequency of autosuggest suggestions to the circumstances: if the human frequently accepts a suggestion or peruses the options, it should offer suggestions more frequently.

Topics

High-precision control of low-stiffness sytems with bounded-Q resonances

Kragen Javier Sitaker, 2017-05-29 (updated 2017-06-01) (4 minutes)

By inverse-filtering the control signal applied to a plant by the estimated OTF of the control function, we can compensate for arbitrarily poor stiffness, up to limits imposed by the control-output bounds and the Q factor of series resonances in the system which impede our ability to impose rapid changes on it. The estimated OTF can be updated moment by moment from incoming sensor data, which permits compensation for mild nonlinearities in the plant; for example, the resonant frequencies of a robot arm may change as it is being extended. High-Q-factor series resonances impose notch filters on the spectrum of the OTF, making it poorly conditioned, thus requiring large components in the inverse filter. These large components can easily cause the inverse-filtered signal to have a very poor signal-to-noise ratio or to exceed the limits of the control actuators, requiring for example very large forces, velocities, or displacements. This problem can be ameliorated somewhat by using nonlinear optimization algorithms, rather than simply solving a linear system, in the control loop. However, in many cases, it may be better to change the design of the plant to damp the high-Q resonances. For example, in a mechanical system, these high-Q resonances can be damped and broadened by adding dashpots or other dissipative elements, thus trading off efficiency for precision control. Counterintuitively, the common approach of increasing rigidity can worsen the controllability of the system when using such an adaptive control algorithm, as it increases the Q factor of the system's vibrational modes, even as it moves them to higher frequencies. As more data is gathered about the system, it becomes possible to empirically estimate the variation of the control-feedback OTF over the plant's parameter space, thus enabling compensation for OTF nonlinearities in parts of the parameter space we expect to visit in the near future. If the control system can learn an OTF that is locally nonlinear or stateful, then nonlinear optimization algorithms in the control loop could potentially compensate even for such phenomena as gear backlash. Finally, using a self-validating analysis system such as reduced affine arithmetic, the control system can optimize not only to reduce the expected deviation between the plant's state and the commanded result, but even for the uncertainty in the plant's state.

All of the above is, to me at least, still somewhat speculative. I have a strong intuition that all of it is true, but a lot of work is needed to verify it in practice. It's about four times too long for a paper abstract!

How to investigate? Well, one thing to try is to simulate a simple physical system, like maybe a linear one-dimensional mass-spring-dashpot system connected to another mass-spring-dashpot system plus a little bit of measurement noise, and try to command it to make some movements with some different

kinds of control systems:

• Maybe a simple bang-bang control system;
• Simple proportional control;
• A tuned PID control system;
• The inverse-filtering-based system I describe above, in its simplest form;
• maybe more advanced forms of it if that seems like it would be better.

That should provide some kind of evidence that this is a good idea, before I get into more complicated plants.

The obvious way to do the simulation at this point is with SVG and JS in Chrome, using direct-mode solutions with maybe a second-degree approximation for the integrals, which will also make it easy to demonstrate to other people. Maybe I can make it game-like, which should make it easy to draw a desired motion with the mouse or a touchscreen in order to see how different control systems respond.

Maybe I can try some experiments first with Jupyter notebooks with Numpy to see what inverse-filtering a noisy signal looks like. Maybe I need to review how to compute the OTF of a linear mass-spring- dashpot system, because I'm pretty sure that's like a closed-form kind of thing.

Topics

• Mathematical optimization (p. 3611) (29 notes)
• Interval and affine arithmetic (p. 3528) (24 notes)
• Robots (p. 3688) (9 notes)
• Control (p. 3390) (9 notes)

Approaches to limiting self-replication

Kragen Javier Sitaker, 2016-11-30 (7 minutes)

Since self-replicating automata entered the fictional literature in Samuel Butler's Erehwon, humans have been concerned that their uncontrolled replication could be dangerous; notable examples include Karel Čapek's "R. U. R.", Lem's "Invincible", Dick's "Second Variety", Drexler's "gray goo", Star Control 2's "Slylandro Probe", and, in a way, Goethe's sorcerer's apprentice's broom. And, of course, our experience with biological self-replicating systems includes numerous troublesome examples of exponential self-replication, including locust plagues, cancer, all kinds of infectious diseases, toxic algal blooms, mold infestations, ant and other insect infestations, rats in cities, and cane toads in Australia. In computers, self-replicating code and related phenomena have caused many problems, from the TFTP "Sorcerer's Apprentice Syndrome" and the accidental fork bomb to the helminthiasis of the internet (the "Morris worm"); nowadays, self-replicating code is a mainstay both of computer security attacks (where it is often called a worm) and defense (where it is often called a security patch or security update).

Our experience with biological systems, however, is misleading when it comes to mechanical systems. If you want to design a self-replicating mechanical system to have a high degree of assurance that it won't continue running on its own, consuming more resources than anyone wanted to produce more replicas of the system than anyone wanted, there are a variety of strategies you can employ without unduly limiting the system's intentional uses.

• Large size.

One reason that cancer is such a problem in animals is that we are made from trillions of essentially autonomous units, each capable of self-replication, most of which (except for red blood cells) do in fact self-replicate in the regular course of events. This provides hundreds of trillions of opportunities for exponential self-replication to arise during the lifetime of a single organism.

By contrast, we can design the self-replication process of a robot to use convergent assembly, in which the minimal self-replicating unit is very large — rather than the tens of microns of a typical eukaryotic cell, it could be hundreds of millimeters, or even more than a meter, in diameter. This lowers whatever risk of exponential replication may exist by a factor of about a quadrillion.

By using convergent assembly, in which the robot contains many small manipulators producing parts to be assembled by asmaller number of larger manipulators, it is possible to obtain this desirable large size without paying an excessive cost in replication time, though this is not intuitive from examining biological models such as elephants, humans, or whales, with their perilously low fertility rates.

• Broadcast architecture.

Rather than keeping a copy of the full program to build a new

robot inside of each robot, as cells do, it has been suggested (originally by Laing, I think, as a way to reduce the size of the robot) to store the program in a centralized transmitter, broadcasting the subprogram for each stage in the process to all of the robots at once. In this way, no robot ever contains the full construction program at once; if the central transmitter ceases to transmit, perhaps because a human has hit the red EMERGENCY STOP button, the entire replication process will cease.

Concern about the space used for the program might seem quaint today when the system-on-a-chip the Raspberry Pi is built around has a gigabyte of RAM. However, this may be purely an artifact of our primitive macroscopic fabrication technology — semiconductor fabs are optimized for efficiently producing consumer products, so the wire-sawing process used to dice wafers has unacceptable waste below a scale of a millimeter or so. With self-replication, we may be able to usefully reduce robot size below the level where each autonomous unit contains space for hundreds of thousands of memory bits.

From a certain point of view, a convergent-assembly desktop factory is a broadcast-architecture machine — after all, what distinguishes the smallest manipulators from autonomous replicators is precisely that they are fixed in place in a larger assembly governed by a larger program they do not have access to.

• Manual process steps.

If the replication process turns a bucket of dirt and rocks into a bunch of robots, a simple way to prevent the process from running amok is to only fill the bucket through human intervention. For example, if the bucket is mounted on top of the replicator, a person could shovel dirt and rocks into it with a shovel. At a larger scale, the person could use a backhoe or a manually operated hydraulic excavator or power shovel capable of depositing hundreds of tons of material in a single operation.

The manual process step need not be the initial raw-material handling step; having a person manually pick up a bucket of finished parts from an autonomous digging part-fabricating robot and dump them into the parts hopper of an assembly robot would work too. If the human stopped carrying parts buckets, no further replicas would be produced. At the extreme, you could require a single human-executed final assembly step, such as installing a fuse or a magic word behind the robot's teeth, in order to bring the fully-assembled robot to life.

However, the raw-material extraction step of the process is the step most likely to cause damage to nearby objects such as buildings, other machinery, or humans, so it is the step most desirable to automate.

• Alternation of generations.

If robot type A is well-suited to producing robot type B, and vice versa, but neither is well-suited to produce others of its own type, then it is possible to employ either type of robots to produce an arbitrary quantity of the other type of robot, or other articles, without ever enabling exponential growth. Exponential growth is only possible if replication of both robots is possible at once — both construction programs and all the necessary raw materials must be present.

Alternation of generations may be desirable for material-processing reasons as well. RepRaps and similar FDM machines cannot replicate

themselves from raw materials because, among other things, they cannot manufacture their hotends (extrusion nozzles), because the hotends necessarily remain solid at temperatures that melt the thermoplastic extruded through them.

Moses, Yamaguchi, and Chirikjian refer to this kind of alternation of generations as a "cyclic fabrication system", saying, "It is cyclic in the way the game 'rock-paper-scissors' is cyclic: tools, materials, and fabrication processes are chosen such that one process creates tools used in the next process…" They suggest prototyping using a hard two-part polyurethane resin which can be cast in wax molds, which in turn can easily be machined by fully-hardened polyurethane; or using a low-melting metal alloy to make mandrels on which to electrodeposit softer but higher-melting metals such as copper and nickel.

Topics

• Self-replication (p. 3703) (24 notes)
• Safety (p. 3693) (9 notes)

The coolest bug in Ur-Scheme

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

One of the fun things about writing graphics code is that you get better bugs. In normal programming, bugs are mostly frustrating: they get in your way, make things harder, corrupt data you painstakingly created, or crash the program and interrupt what you're doing. But a lot of bugs in graphics code either look really cool or have no real visible effect.

At the moment I'm not writing graphics code. I'm writing an almost-Scheme compiler in itself. But I just created a really bizarre bug.

Here's a little bit of the assembly output from the compiler:

```
epacse__4:
        # compute desired %esp on return in %ebx and push it
        lea 4(%esp,%edx,4), %ebx
...
        movl (htgnel_gnirts__2), %eax
```

Where did `htgnel_gnirts__2` come from? Well, it's `_string_length` spelled backwards, followed by `_2`. And `epacse` is `escape` spelled backwards. I accidentally created a bug that *spells names backwards*. That's almost as funny as some of my graphics-code bugs.

How this happened requires a little bit of explanation. In Scheme, as in most Lisps, adding items to the beginning of a list is fast and safe, but adding onto the end of a list is either slow and bug-prone, safe but extremely slow, or very verbose and therefore bug-prone and hard to maintain. But reversing a list is relatively fast. So I wrote a function that looked like this:

```
(define (stringlist->string stringlist)
  (list->string (reverse (stringlist->string-2 stringlist 0))))
```

Because I thought stringlist->string-2 was going to have to build up a list of all the characters *backwards*, and then I was going to have to reverse it.

When stringlist->string-2 turned out to be able to build up the list of characters in the right order --- by adding them backwards --- I forgot to take out the `reverse`.

Topics

Incremental persistent binary array sets

Kragen Javier Sitaker, 2017-04-10 (4 minutes)

Nayuki wrote a very clear description of a data structure she calls a binary array set, or BAS. It's a linked list of sorted arrays in power-of-2 increasing sizes, merged upon overflow; insertion is θ(1) amortized (Θ(N) worst case), and membership testing is average-case and worst-case Θ((log N)²).

I asserted to a friend of mine that incrementalizing it to get Θ(1) worst-case insertion time was straightforward. The transformation also gives you an FP-persistent† version of the BAS data structure. Here I sketch the details.

Merging strategies

With power-of-2 array sizes, if you happen to be at a state with 1024 items, the immediately previous state will have had array sizes of 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512. The item in the array of size 1 will have been merged zero times, the two items in the array of size 2 will have been merged one time, the size-4 array will have been merged from two size-2 arrays and thus each item in it has been merged twice, and so on; the 512-item array contains items that have all been merged 9 times. That is, the items in an array of size 2^n have been merged *n* times. So, in fact, under this merging policy, amortized insertion time per item is not constant; it is Θ(log N).

I thought we could do better using a multi-way merge. All the items in the 1024-item array have been through the same number of merges (10) because the behavior upon inserting the 1024th item is to merge it to produce an array of 2 items, which then must be merged and discarded to produce an array of 4 items, etc, doing a total of $10 \cdot 1 + 9 \cdot 2 + 8 \cdot 4 + \ldots\ 2 \cdot 256 + 1 \cdot 512 = 2036$ work, where each unit is a comparison and a couple of pointer copies. If we instead did a single 9-way merge using a 9-item binary heap, the average work per item is about 3.17, so we end up doing about 3200+1024 work instead. Memory locality is surely better; not counting the accesses to the tiny heap, we only have to do 1024 work this way.

This is not obviously better, and although I haven't done the analysis for real, I don't see a strong reason to expect it to get better asymptotically. So it's probably best just to use the simple strategy.

Incrementalizing

The basic idea is that every time you perform an insertion, you also do enough merge work to ensure that the merging stays ahead of insertion. This is somewhat tricky, in that we can't simply stop merging newly inserted data until the completion of a large merge further down the chain; that would kill our search-time guarantee. So at times a large merge must be paused while a small merge is performed.

As noted above, the

An incremental BAS state is either a merged BAS (one with no

† "Persistent" unfortunately has two conflicting meanings when it

comes to data structures; functional programmers use it to mean that modifying the data structure does not make its previous states inaccessible. By "FP-persistent" I mean this meaning, not the more common meaning of "retrievable after rebooting your computer".

Topics

Spring energy density

Kragen Javier Sitaker, 2016-09-05 (updated 2019-04-20) (3 minutes)

What's the maximum energy density of a spring? What material is best?

As you put more and more energy into a spring, eventually you will break it. The properties of the spring material put a limit on how much energy you can store in the material before you break it.

Typically the relevant numbers are yield stress (or elastic limit) and some kind of modulus of stiffness. Some springs are stressed in shear (like a torsion bar in torsion, or a coil spring in compression or tension), while others are stressed in tension or compression, like the surfaces of a leaf spring or a bow.

Common geometries, like the torsion bar, coil spring, or leaf spring, have a zero-stress region in the middle of the material, with stress increasing linearly up to the surface, where it is at a maximum. This geometry reduces the spring's energy density to half of the theoretical maximum for the material. Other geometries, such as tubular torsion bars or I-beam leaf springs, can reduce the wasted weight. Springs stressed in pure compression, like a squeezed block of rubber, or in pure tension, like a stretched wire, do not have this problem.

https://en.wikipedia.org/wiki/Shear_modulus say:

The shear modulus is one of several quantities for measuring the stiffness of materials. All of them arise in the generalized Hooke's law:

• Young's modulus describes the material's response to uniaxial stress (like pulling on the ends of a wire or putting a weight on top of a column),
• the bulk modulus describes the material's response to uniform pressure (like the pressure at the bottom of the ocean or a deep swimming pool)
• the shear modulus describes the material's response to shear stress (like cutting it with dull scissors).

It also gives a table:

Material	Typical values for shear modulus (GPa) (at room temperature)
Diamond[2]	478.0
Steel[3]	79.3
Copper[4]	44.7
Titanium[3]	41.4
Glass[3]	26.2
Aluminium[3]	25.5
Polyethylene[3]	0.117
Rubber[5]	0.0006

http://www.freebase.com/base/materials/solid_material/shear_modulus?instances= has 13 values for the shear modulus that are totally useless because they have no attribution and no units. (Also, they have no precision.)

http://www.engineeringtoolbox.com/modulus-rigidity-d_946.html gives these values:

Material	Shear Modulus - G - (10^6 psi)	(GPa)
Aluminum Alloys	3.9	27
Aluminum, 6061-T6	3.8	24
Aluminum, 2024-T4	4.0	28
Beryllium Copper	6.9	48
Brass	5.8	40
Bronze	6.5	44.8
Cadmium		19
Carbon Steel	11.2	77
Cast Iron	5.9	41
Chromium		115
Concrete	3.0	21
Copper	6.5	45
Glass		26.2
Glass, 96% silica	2.8	19
Inconel	11.5	79
Iron, Ductile	9.1 - 9.6	63 - 66
Iron, Malleable	9.3	64
Kevlar	2.8	19
Lead	1.9	13.1
Magnesium	2.4	16.5
Molybdenum	17.1	118
Monel metal	9.6	66
Nickel Silver	6.9	48
Nickel Steel	11.0	76
Nylon	0.59	4.1
Phosphor Bronze	5.9	41
Plywood	0.09	0.62
Polycarbonate	0.33	2.3
Polyethylene		0.12
Rubber		0.0003
Structural Steel	11.5	79.3
Stainless Steel	11.2	77.2
Steel, Cast	11.3	78
Steel, Cold-rolled	10.9	75
Tin		18
Titanium, Grade 2	5.9	41
Titanium, Grade 5	5.9	41
Titanium, 10% Vanadium	6.1	42
Tungsten		161
Wood, Douglas Fir	1.9	13
Zinc		43
Z-nickel	11	76

(See also You can stuff a UHMWPE hammock in your wallet (p. 799).)

The number I was looking for is called the “specific energy” or “elastic potential energy per unit volume”.

Topics

Making a mechanical state machine via sheet cutting

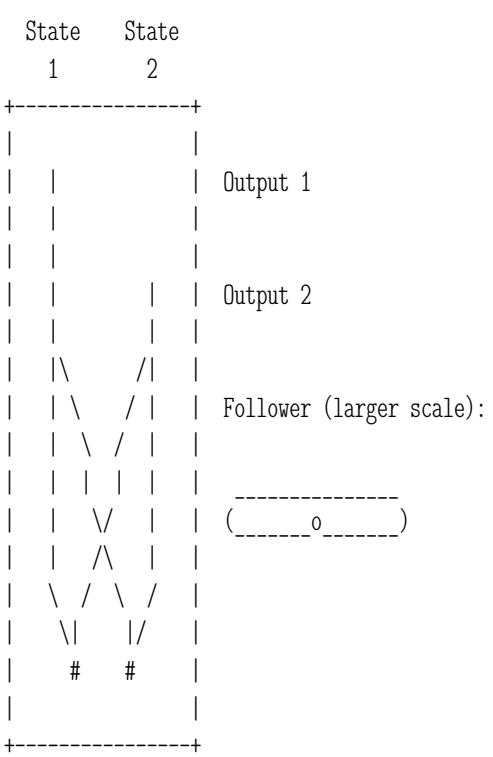
Kragen Javier Sitaker, 2014-04-24 (updated 2015-09-03) (7 minutes)

It occurred to me that there is a simple way to build lookup-table (LUT)-based mechanical finite state machines with planar fabrication. I think this is considerably less dense than than the heightfield- based LUTs I've written about previously. It represents the transition matrix as a maze of channels cut into a surface, with a single follower traversing the maze under the influence of springs and friction.

Press the button on the back of a pushbutton ballpoint pen, and the tip extends; press it again and it retracts. It's a mechanical state machine with two states that takes no inputs. It works by having a rotating cylindrical thingy that slots into guides in the outer barrel of the pen; they force it to rotate when it moves longitudinally.

We can extend this mechanism to more general state machines, that take N discrete inputs and transition arbitrarily between M discrete states, each of which has one of P discrete outputs. The input is encoded as a distance to which you push the thing corresponding to the ballpoint pen button against the spring; the state is encoded in the rotation of the barrel; and the output is encoded as a distance to which the spring is able to push back once the input is withdrawn. And the mechanism is changed somewhat to be able to handle general state machines.

For simplicity of diagramming I will unroll the cylindrical thingy to be planar. Here's a way to realize our ballpoint pen with my new scheme:



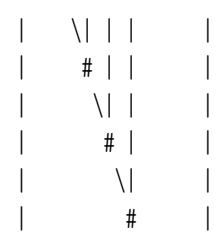
The paths here are smoothly curved channels in the plane. An oblong "follower" with rounded ends begins in either the State 1, Output 1 or State 2, Output 2 position at the top of the diagram, and

is then pushed down to the bottom. It's free to move horizontally, but has to overcome some friction to do so. Where the channels branch, the branch points are wide enough that it is kinematically capable of following either branch; but the horizontal friction ensures that it simply continues straight down; it follows the curves down to one of the transition points marked #.

When the input pressure is removed, it begins to return back upwards, but because it's at a branch point with an available vertical path it follows that branch rather than returning by the same path by which it came — and thus it ends up going to the other state. At the point that the two return paths cross, the intersection is not wide enough to allow the follower to turn to follow the wrong return path, like the shuttles in a trammel of Archimedes.

(If the plane were a cylinder, the two paths would not have to cross over each other, and that is in fact the approach taken by actual ballpoint pens. Also, they typically have a larger number of states, such as 8 or 12, but with only 2 outputs. And commonly they have a separate "nudge" in the thing propelling the follower vertically to push it into the return path, but that isn't strictly necessary.)

When there are multiple possible inputs, there will be multiple return branch points on each path down, branching off at different heights:



These different branch points are independently routable to arbitrary different next states. This lets you use any arbitrary transition function for your state machine. However, in the general case, all these separate M*N return paths need to be able to pass in parallel and cross over before getting back to the states, which puts a limit on how complex our state machine can be in a given space.

This mechanism provides fan-out, but not fan-in: unlike with the heightfield LUT, combining two separate inputs, such as two summands or multiplicands, must be done with some other mechanism. But such mechanisms can be fairly simple.

In addition to wrapping the transition maze around a cylinder, you can also wrap it around on a disc, as shown by the spiral cams in " Basic Mechanisms in a Fire Control Computer (1953)", using rotation rather than translation as one of your dimensions of motion; and indeed, if you distort the other dimension into a series of arcs, you can use rotation rather than translation for both of them, while retaining the compactness of planarity.

Heightfield LUTs are not novel

This video also shows that my heightfield LUT was already invented, in analog form for continuous functions in 1953; apparently it's called a "barrel cam". The Equation of Time Cam, from the first prototype of the Clock of the Long Now, is an example that I had seen. It compensates for the changing time of solar noon throughout

the year.

Investigating further, it turns out that, before 1953, the Jaquet-Droz automaton known as The Writer used a heightfield LUT, in the form of a stack of cams, to encode a vector font; this automaton is a pen plotter with a non-ASCII character set, in the shape of a small boy with curly hair. Three cam followers control the X, Y, and Z motions of the pen, and the axial displacement of the camshaft determines which character is produced. Wikipedia explains that the automaton was constructed between 1768 and 1774. It still works, more or less, and is demonstrated once a month at the Musée d'Art et d'Histoire de Neuchâtel.

In between the Writer and the Cold War barrel cams, the rotational dimension of the cam was generalized to be any continuous input variable, rather than just time; and the axial displacement dimension was generalized to be possibly continuous rather than discrete. I don't know when between 1774 and 1953 these two generalizations occurred, nor whether they might have been present in other machines even before 1774.

This is interesting to me because, as I wrote in 2010, LUTs have major advantages for mechanical digital computation: they're much easier to design than combinations of universal gates like NANDs or NORs, and they also involve many fewer moving parts. So it seems like you could have built a general-purpose digital computer in the 1700s, at least in Switzerland, if you had the idea that it was an interesting thing to want to build.

Topics

Bistable magnetic electromechanical display

Kragen Javier Sitaker, 2019-10-24 (16 minutes)

I recently watched a YouTube video by "GreatScott!" demonstrating a bistable magnetic electromechanical 7-segment display; each segment is a slot through which a moving part is visible, one part of which is white, while another part is black. There are 7 electromagnets on the back of the display; by passing a pulse of current through them in one direction or the other, a permanent magnet attached to this moving part is given a kick big enough to swivel the part and change the color of the segment. The permanent magnet has two stable positions in which it is attracted to a ferromagnetic core, so the display remains stable without applied power.

The display is multiplexed with a per-digit common ground, so that, for example, an 8-digit display with 56 segments requires only 15 wires — but 7 of them need to be bipolarity.

Digital electromechanical decoding

It occurred to me that in some sense this was excessive; 7 bipolarity signals (with distinct +, -, and 0 states) can encode $3^7 = 2187$ commands rather than the 14 needed to switch 7 segments or even the 112 needed to switch 56 segments. 7 *unipolarity* signals are enough to encode 128 commands, enough to switch 64 segments on or off. Moreover it should be feasible to route the magnetic flux in such a way that the decoding is done passively, by the magnets.

The crucial tricks are:

- You can route magnetic flux to an area by providing a low-reluctance path from a coil to that area; for example, a sheet of electrical steel, or more prosaically, a mild steel wire. (At kHz and higher frequencies, ferrites might be useful.)
- You can sum flux from different coils by terminating low-reluctance paths in the same area, but with enough of an air gap between them that there isn't a low-reluctance path between the different coils. It's sufficient that the flux they produce impinge on a magnet or magnets attached to the same movable part.
- You can route magnetic flux from either end of a coil, so you can get both subtraction and addition.
- By including fixed permanent magnets in the mix, you can include a constant term in these sums.

This allows you to set the flux in a given area to an arbitrary affine function of the currents in the different coils. Consider an area you want a movable permanent magnet to be attracted to only when the code 0001101 is present on the coil control wires. You route the positive ends of coils 0, 2, and 3 to that area, and the negative ends of coils 1, 4, 5, and 6, and include a fixed permanent magnet powerful enough to counteract the flux from just over 2 coils, but not more.

In this area, if coils 0 and 2 only are energized, they are not sufficient to overcome the fixed permanent magnet, and the movable

magnet continues to be repelled from the area. If coil 1 is additionally energized, it partially cancels the flux from coils 0 and 2, repelling the movable magnet even more strongly. Now if coil 3 is energized, we have 0, 2, and 3 fighting against 1 and the permanent magnet, not quite enough to overcome it; but if coil 1 is then de-energized, the balance flips, and the area becomes attractive rather than repulsive.

In practice this probably means that one such "balance point" is needed for each pixel — a position which can be made a stable equilibrium with the right combination of energized coils, but becomes an unstable equilibrium when power is removed — and once the movable part has been brought to this balance point, one of two additional coils is energized to tip the equilibrium in one direction or the other, while the other coils are de-energized.

(Slightly tweaking this, instead of using two additional coils, you could use one additional coil and a permanent magnet; this means that the balance point is not quite an unstable equilibrium when power is removed.)

So, simply by choosing the polarities with which each coil is coupled to each pixel, we can make a unique combination of coil activations the strongest for that pixel, then provide a permanent magnet strong enough to cancel any combination other than that one. In essence this is an electromechanical McCulloch–Pitts neuron.

Stable, high-coercivity rare-earth or even ferrite permanent magnets will work much better for this than unstable alnico magnets, because one of the magnets needs to have a strength that stably discriminates between the case where, say, 6 coils are activated in concert, and the case where 7 are.

A trick not needed: by using different thicknesses of ferromagnetic material, you can get different amounts of flux from the same amount of electrical current. This allows you to compute weighted sums and differences. However, though this trick is not needed, it is an alternative to using varying strengths of fixed permanent magnets in the different cells; it would allow them all to be the same strength.

PWM electromechanical decoding

A hard disk drive's head is positioned with a voice-coil actuator by running a precisely controlled current through a "voice coil", producing a precisely controlled magnetic field which moves the head to a precisely controlled position within a few milliseconds. Dynamic speakers work on the same principle, moving the speaker cone to what is in principle a precisely controlled position by producing a precisely controlled magnetic field with a precisely controlled ac voltage. Class-D audio amplifiers generate that voltage by, essentially, reactively low-pass filtering a PWM signal.

A very simple way of decoding PWM would put a floating magnetic compass globe, like those people used to have on their car dashboards in the 1980s, in an enclosure with a small transparent window through which a single digit could be seen, out of ten printed on the globe in different positions; a permanent magnet would align the globe to display "0" in its equilibrium position, and a coil producing a field at perhaps 120° from that of the permanent magnet could swivel the globe to any desired position 1–9 by applying an appropriate strength of field. A second coil producing a vertical magnetic field could provide a magnetic dip to counteract the globe's

tendency to return to a default horizontal position; this could be used, for example, to select from a larger repertoire of characters, or to engage mechanical interlocks that kept the globe from turning when power was removed.

(For some reason, the traditional way of doing this, the galvanometer, uses a mechanical hairspring rather than a permanent magnet to return the needle to its zero position when power is removed.)

If you have some array of magnetically-responsive pixels — for example, Dapper-Dan-style magnetic whiskers in tiny mostly-transparent plastic boxes, parts of which are opaque white — you can use a similar approach to scan a needle in a four-bar linkage involving two galvanometers back and forth over this array of pixels, activating a magnetic field at its tip to change the color of a pixel when appropriate. I think we can expect this to be slow and scale down poorly, but it would work.

Tiny permanent magnets behind white paper, or better still boundaries between magnetic poles behind white paper, could potentially make the pixels bistable in the absence of friction — the black filings would remain stably stuck to them in the absence of any applied magnetic field, even in the face of slight vibrations, but could be persuaded to leap to a different attractive spot by a temporary cancellation of the magnetic field with the needle tip.

An advantage of using magnetic-pole boundaries is that the field projected from the magnetic tip wouldn't have to be perfectly calibrated — any amplitude large enough to temporarily more than cancel one of the poles would cause all the pole boundaries around that pole to temporarily disappear, encouraging the filings to migrate to a different still-existing boundary between poles. By alternating the field a number of times, filings in the area could perhaps be vibrated loose from any frictional moorings that prevent them from vacating the area.

Even without any PWM signals, scanning one or more needle tips over a two-dimensional area could be effected by purely mechanical means, for example in a VCR-like helical pattern, or a Spirograph-like family of circles of the same radius rotated around a center, or a Lissajous pattern created by two elastic resonant modes of different frequencies. Then, a persistent image could be produced simply from a time-varying magnetic field at the needle tip.

Non-magnetic equivalents using electrets

Suppose that instead of magnetic fields we use electric fields, and instead of permanent magnets we use electrets, which have the potential advantage of being monopole-capable. (As far as we know, magnetism is not monopole-capable, but all the electrical particles we know of are electrical "monopoles", and so too are chunks of charged electret.) As described in Paper/foil relays (p. 3273), this should scale down rather well. This is more or less how e-ink displays work, but without the in-display decoding.

To be concrete about one possible realization, suppose we have some negatively-charged black electret particles suspended in oil in a tiny linear capsule; one end of the capsule is transparent, while the other end is opaque. We have some more negatively-charged electret embedded in the wall of the capsule near its center, slightly toward

the opaque end, so the particles tend to drift toward the ends of the capsule when no voltage is applied, and in particular if they start out precisely in the center, they will tend to drift toward the transparent end, making the capsule look black. Lines 0, 2, and 3 are connected through capacitors to electrodes wrapped around places near the center of the capsule, but not overlapping, so that their mutual capacitance is low. Lines 1, 4, and 5 are connected through somewhat larger capacitors to electrodes at each end of the capsule, and line 6 is connected through a capacitor to an electrode on the opaque end of the capsule.

If +5V is applied to all of lines 0, 2, and 3, this pushes a certain amount of charge through the capacitors onto the electrodes around the center of the capsule, calibrated to be sufficient to shift the equilibrium such that the suspended electret particles will tend to drift from the ends to the center of the capsule. If only two of these lines are energized, this will not push enough charge onto those electrodes to cancel the wall-embedded electret. If all three of them are energized, but also one or more of lines 1, 4, and 5, there will be a net positive charge in the center of the capsule, but a larger net positive charge at the ends, so electret particles will remain at whichever end they are. But if none of these inhibitory lines are energized, the particles will move to the center, or rather, into a cloud near the center but slightly toward the transparent end.

If lines 0, 2, and 3 are then grounded, the wall-embedded electret will repel the cloud back to the transparent end. But if first line 6 is brought high, it will move the cloud past the wall-embedded electret, and then when lines 0, 2, and 3 are grounded, the cloud will migrate to the opaque end instead.

Of course, the same seven lines can control 63 such capsule pixels in this way, with lines 0–5 varying between inhibitory and activatory roles on different pixels, and line 6 always controlling which way the equilibrium falls when the decoding lines are released. Different capsules may require different amounts of wall-embedded electret to cancel their varying numbers of activatory lines, or perhaps the series capacitances that set the charge could instead be varied.

This is substantially more complex than the current schemes of e-ink displays, and it requires fairly high precision of manufacture to precisely calibrate the varying amounts of electret in each capsule, as well as precision of design to distribute the electrical fields properly.

Non-magnetic equivalents using other kinds of actuators

We can easily go rather far afield with these ideas.

A scanning needle tip (whether raster, Spirograph, Lissajous, or otherwise controlled) of course can be activated in other ways. For example, mechanical actuation — in machining this is called a dot-peening machine and is used for alphanumeric part marking of malleable surfaces, and I've used a handheld "Vibro-Graver" version of the same process to mark my hand tools. In an electrolyte, voltage on a scanning needle tip can produce an image on a surface by selective electroplating or electrochemical machining (depending on polarity), and in air it can produce a corona discharge, which can selectively functionalize passivated surfaces (see Cold plasma

oxidation (p. 2406)) or produce light. If scanned over the same surface in an electrolyte for a long period of time, it can be used to 3-D print by electrodeposition or to cut an almost arbitrary cavity by electrochemical machining. On a few metals, like silver, such electrolytic processes can be used to induce a reversible, localized, dramatic color change; as mentioned in Electrolytic anodizing, with a small movable electrode (p. 3059), anodizing of titanium can produce quite brilliant colors through iridescence, and this can be done selectively to produce color images.

Some of the summing-and-differencing approaches discussed above might be usable to select individual "pixels" in such processes as an alternative to moving a needle around; for example, anodic dissolution of a metal workpiece will happen only in areas where there's a *net* current of positively-charged cations from the workpiece to the tool pixels, while it's possible to prevent anodic dissolution of tool pixels by making them out of carbon, and electroplating of the tool pixels with a suitable electrolyte. So if all but one of the tool pixels have a positive current flowing from them to the workpiece because of summing negative and positive currents, it should be possible to do selective electrochemical machining at the one pixel that is sucking up cations instead.

Relays

Most of the above methods can be adapted to activate electrical switches rather than optical pixels.

Topics

- Electronics (p. 3430) (138 notes)
- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Manufacturing (p. 3558) (50 notes)
- Mechanical things (p. 3569) (45 notes)
- Optics (p. 3609) (34 notes)
- Displays (p. 3414) (13 notes)
- Relays (p. 3681) (3 notes)

Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups

Kragen Javier Sitaker, 2012-12-04 (5 minutes)

(Probably a duplicate of a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190).)

I don't have much time to write this, so I may end up sending out a version that's somewhat telegraphic.

There's an alternative to summed-area tables with a small, linear space cost and linear construction time providing worst-case logarithmic-time reduction over arbitrary intervals over arbitrary semigroups.

Explanation

Summed-area tables

Franklin Crow's 1984 paper, "Summed-area tables for texture mapping" calls them "summed-area tables", and Graphics Gems called them "sum tables". More recently, they're known as "integral images". In the one-dimensional case, they allow you to calculate the sum of values in an arbitrary interval in constant time by subtracting the values from the summed-area table at the ends of the interval: sum(f[m:n]) = -sat(f)[m] + sat(f)[n], where sat(f)[i] = sum(f[0:i]), for a suitably low value of 0.

Decimation

As an extension, you can use a decimated summed-area table, with values only present every (e.g.) 16th or 32nd index, without losing the constant-time property. You may have to consult the original array, but only up to 2*(16-1) or 2*(32-1) values of it, which is constant. This dramatically reduces the space cost of the technique.

Generalization over operations

You can generalize the sum-table idea beyond integer addition? Clearly they work fine for mod-N integer addition, vector addition, and the combination of the two (e.g. XOR). I think your operation only needs the properties of associativity and left inverse, and to preserve the constant-time property, you need the constraint that the operation must be computable in constant time.

(For the N-dimensional case, I think you may also need commutativity.)

A logarithmic-time alternative to sum tables for semigroups

But what do you do if you're interested in an operation that doesn't have a left inverse? For example, the "minimum" operation (or in general the meet operation of a meet-semilattice) can't have inverses of elements, because it's idempotent, so you can't compute it with a sum table.

But you *can* compute it in logarithmic time with a tree. Let

```
mint(f, m, n) = nil if m == n
              = (m, n, min(f[m:n]), mint(f, m, floor((m+n)/2)),
                                    mint(f, floor((m+n)/2), n)) otherwise
```

Now if you precompute mint(f, 0, f.length), which is a balanced binary tree with 2*f.length - 1 nodes, not counting the nils, and which can be computed in linear time, you can compute min(f[m:n]) for arbitrary m, n in logarithmic time given that tree. That algorithm is straightforward.

This algorithm applies to any semigroup over the elements; it can be used to calculate sums as easily as minima, although more slowly than using a sum table.

Space reduction: decimation

Analogously to sum tables, if your leaf nodes represent spans of some 16 or 32 elements instead of 1, you get a dramatic space reduction without losing the logarithmic-time asymptotic performance.

Space reduction: array storage

The contents of the tree produced by the mint() function depends only on m and n, except for the min(f[m:n]); and if f.length is a power of 2, it is a full binary tree. A full binary tree can be stored, as in the classic binary heap, in an array a such that the children of the element at a[i] are at a[2i+1] and a[2i+2] (zero-based). So you can store the minima for the tree in an array (without decimation, of 2*f.length - 1 elements) rather than allocating numerous nodes on the heap.

This requires a slight enhancement to the lookup algorithm to recompute the same (m, n) as the construction algorithm, rather than looking them up in the tree.

Constant-space bottom-up construction

If you construct the tree recursively, in addition to the O(N) space for the results, you need O(log N) stack space. But that is not necessary. If you're using the array storage suggested in the previous section, you can fill the array starting from the end, so that the only auxiliary storage you need for the construction process is a simple counter.

Enhancement: indices

In the case where the semigroup operation is exactly minimum or maximum over a totally ordered set, the value stored in each treenode will be the value of one of the items in the original array. In this case it is strictly more powerful to store the index of that item rather than its value. This may be useful if you have some other data that are indexed the same way.

N-dimensional case

This generalizes easily to k-d trees, although the efficiency guarantees are not as good.

Topics

Viral wiki

Kragen Javier Sitaker, 2015-10-15 (3 minutes)

Suppose that you wanted to preserve some textual library and ensure that it remained available. Perhaps you could ensure that by enticing people to comment and contribute, but requiring them to publicly self-host a replica in order to do so.

In particular, you could do this by requiring them to comment by adding the comment in their own local fork of the library and sending a pull request. They could self-host privately, or read without commenting, and remain private; but in order to comment, they would need to host a copy themselves and keep it online long enough for the author to pull from it.

This would ensure that many copies of this library would exist.

Ideally, downloading a replica of the library wouldn't take any more bandwidth than downloading a regular web page. But how much is that?

Right now, 2015-10-15, loading without cache:

what	how much	reqs
Hacker News front page	7.5kB	5
a Hacker News comment page	31kB	1
"Life" on English Wikipedia	570kB	47
Alexa.com's "top 500 global sites" page	1.1MB	69
the "videos" page for HolaSoyGerman on YouTube (without a video)	1.4MB	
my timeline on Twitter	1.9MB	80
a NY Times editorial from March	2.6MB	179
consumer electronics salesman's LinkedIn page	3.1MB	101
my TL within a few minutes	4.6MB	466

| Amazon page for a deep-cycle lead-acid battery | 5.0MB | o o193 |

Watching a high-resolution YouTube video can easily use 12 megabytes per minute.

So a median web page now is about 2MB of data in the initial load, and another several megs per minute. So it probably isn't useful any more to keep the initial load below, say, ¼MB, and you can probably use 16 to 64 megs of data transfer to do the whole replication if the user doesn't have to wait for all of that, but you probably don't want to cost them 128 or 256 megs. I say this even though finding all of the above sites frustratingly slow, except for Hacker News, because apparently people use them anyway.

(Chris Zacharias's famous "Page Weight Matters" tells about how he made YouTube much more useful in 2009 by reducing the Youtube watch page's weight from 1.2MB to 98kB, enabling its use in areas in Siberia and Southeast Asia where 98kB still took two minutes to load. I think that probably in 2015 the number has gotten substantially higher.)

Topics

- Performance (p. 3621) (149 notes)
- Systems architecture (p. 3691) (48 notes)
- Archival (p. 3322) (34 notes)
- Gossip (p. 3478) (6 notes)

Urban autarkic network

Kragen Javier Sitaker, 2018-04-27 (1 minute)

The ESP8266 uses 200 mA at 3.3V when operating, which is 660 mW. With a duty cycle of 4.5%, that's 30 mW; in sleep mode it takes a bit under 3 mW. People report success in getting it to average 18 mW. It costs about US$6. Brian Benchoff reports a range of 366 m with the antenna that's built into the printed circuit board. The IXYS SLMD121H04L solar cell costs US$6.20 in quantity 1 and yields 89 mW in, presumably, full sun; it's 43 mm x 14 mm (150 μW/mm²). So you could run an ESP8266 off it during the daytime with a fairly reasonable duty cycle, given reasonable power circuits. A 10000 μF 35V TDK aluminum electrolytic capacitor costs US$2.60 and is 22 mm diameter × 52 mm long; it can store, in theory, 6 joules, which is 9 seconds of full-duty-cycle operation.

You could set up a mesh network of solar-powered ESP8266s 250 m apart in order to provide a communication line. You could drop them in trees, on top of smokestacks, on the roofs of industrial buildings, painted black. Each board might cost US$25, so you would need to spend US$100 per kilometer.

Topics

Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times?

Kragen Javier Sitaker, 2016-09-06 (3 minutes)

Create and destroy population inversions to make lasing gain positive or negative, thus providing high-gain amplification. Pipeline bits through optical fiber with length comparable to fiber diameter (≈8μm or 30fs, thus enabling switching speeds in the tens of THz). Doped fiber sections pumped from the side by another laser produce an AND gate in the obvious interpretation, but you can do better still with Manchester encoding: the stimulated emission from one pulse leaves behind a lack of population inversion which will attenuate following pulses for a while. Wideband-excitable materials like trivalent-erbium-doped fiber amplifiers may permit wavelength-division multiplexing in computation; the inhomogeneous-broadening-induced effect known as "spectral hole burning", usually considered a nuisance, thus provides a way for pulses at one wavelength to *suppress* pulses at nearby wavelengths. Dynamic memory can also be constructed by using a metastable population inversion to store each bit, as long as the spontaneous-emission half-life is long enough to permit a reasonably low-frequency refresh cycle.

Taking full advantage of these effects will require micron-precision optical-path-length matching.

Induced emission from metastable nuclei

Increasing characteristic operating frequencies with these approaches past the tens of THz requires stimulated emission or at least induced emission at shorter wavelengths. Induced or stimulated emission from nuclear isomers provides a plausible route to six to nine orders of magnitude faster operation; note that these timescales are so short that even deep-sub-nanosecond-half-life nuclear isomers could be useful, dramatically broadening the possible range of possibly useful substrate materials, which should reduce potential conflicts between this computational technology and proliferation concerns. Aside from the present difficulties of inducing metastable nuclear state decay, this approach has another serious difficulty: controlling the flow of high-energy gamma rays is more difficult than simply using optical fibers.

Neutron-induced-fission logic

Aside from the possibility of induced emission from majority-metastable nucleus populations, neutron-induced fission has been known since the 1930s and a practical energy source since the 1940s, and produces prompt neutrons within 10fs of the fission, and they can be channeled to some extent by neutron reflectors. However, it is not entirely clear to me how to use this effect for

computation, since I don't see how to combine two different neutron signals in anything other than a sort of logical OR. Moreover, aside from proliferation concerns, computational devices built using this approach would need the fissile nuclei physically replaced after firing, a process that will surely take at least microseconds if not entire seconds; so although a nuclear prompt-fission chain-reaction logic device could compute with propagation delays in the femtosecond range (if we figure out how to combine signals usefully), its clock rate would be very low.

Topics

Affine arithmetic has quadratic convergence when interval arithmetic has linear convergence

Kragen Javier Sitaker, 2016-08-24 (updated 2017-01-18) (10 minutes)

The standard interval-arithmetic approximation is wonderful for ensuring correctness, but it doesn't work very well in regions where the derivative is high. Functions can be perfectly regular in the analytic sense and still have an arbitrarily large derivative. As a result, if you are computing a function over an interval and you chop the interval in half repeatedly, you may end up merely chopping the function's result in half repeatedly. (Or not even that, if it has a singularity.)

Can we do better?

As an alternative, you might think we could conservatively approximate the function's value over that interval $x \in [a, b)$ with a triple (m, d, e) such that $x \in [a, b) \Rightarrow f(x) \in [mx + d, mx + e)$. Just as with simple intervals, there are many possible such triples that could be used; it may not be immediately clear which is the best choice, but I think it's the one with the smallest e - d value. (For a function given as an enumeration of points, there is a usually efficient algorithm beginning with its convex hull.)

The identity function on any interval is represented by (1, 0, 0); the constant k, similarly, is (0, k, k).

Addition and subtraction of such values is fairly easy and reminiscent of that with intervals. Multiplication may or may not have a local maximum which must be taken into account; division, as with interval arithmetic, may or may not have a singularity, as well as perhaps local maxima.

For a regular function, intuitively I expect that, when I divide a small enough interval in half, its derivative should vary by half as much over the interval, because on a small enough interval, its third and higher derivatives create too little change to matter, so it looks like a parabola.

I intuitively guess that the best approximation of a parabolic segment (in the e - d sense) is tangent to the parabola at the midpoint of the interval. Adding a linear function doesn't affect the e - d error, so for analysis we can add a linear function that brings both endpoints of the interval to the X-axis, with the midpoint horizontal. Now if we divide this into two sub-parabolas and do the same with them, their second derivative is the same as the original, but with only half as far to affect the parabola from the horizontal midpoint to the endpoint, it can reach only a fourth as high.

Therefore, I intuitively expect this approach to quadruple its accuracy when you chop an interval in half, so it will need half as many evaluations to reach the same accuracy of some arbitrary regular function as the simple interval-arithmetic method. As a bonus, it provides an approximation of the first derivative over that same interval, but it is not a conservative approximation.

I intuitively expect this accuracy improvement to be crucially

important for a composition of functions, since it means that the output of your approximation is in some sense more precise than its input — so you might not suffer progressive degradation as you get further from the input.

If you extended this approach to use a higher-order approximation than linear, you could perhaps tighten the bounds further; but this isn't necessary in order to get the precision-improvement property mentioned above.

When extended to multiple independent variables, this linear approximation approach requires linear extra work per independent variable; instead of a single variable m, you have a vector of coefficients $[m_0, m_1, ...m_n]$, and the dot product of this vector with the vector of independent variables $[x_0, x_1, ...x_n]$ gives you the approximation correction. Higher-order approximations would necessarily involve sets of coefficients at least quadratic in the number of independent variables.

Bisection methods potentially take time exponential in n in an n-dimensional space.

Arithmetic operations

As I said before, the identity function $i(x) = x$ is represented as $I = (1, 0, 0)$, and the constant function $k_n(x) = n$ is represented as $K_n = (0, n, n)$. But where do we go from there?

Given

- $x \in [a, b) \Rightarrow j(x) \in [m_j x + d_j, m_j x + e_j)$ $(J(a, b) = (m_j, d_j, e_j))$
- $x \in [a, b) \Rightarrow k(x) \in [m_k x + d_k, m_k x + e_k)$ $(K(a, b) = (m_k, d_k, e_k))$

Then what can we say about pointwise operations on these functions?

Addition is fairly simple. Clearly $j(x) + k(x) \in [(m_j + m_k)x + d_j + d_k, (m_j + m_k)x + e_j + e_k)$, so $(J + K)(a, b) = (m_j + m_k, d_j + d_k, e_j + e_k)$.

Subtraction is only slightly trickier; $j(x) - k(x) \in [(m_j - m_k)x + d_j - e_k, (m_j - m_k)x + e_j - d_k)$, so $(J - K)(a, b) = (m_j - m_k, d_j - e_k, e_j - d_k)$.

Multiplication starts to get hairy.

The product $j(x)k(x)$ is guaranteed to be in the interval

```
[min((mⱼx + dⱼ)(mₖx + dₖ),
     (mⱼx + dⱼ)(mₖx + eₖ),
     (mⱼx + eⱼ)(mₖx + dₖ),
     (mⱼx + eⱼ)(mₖx + eₖ)), min((mⱼx + dⱼ)(mₖx + dₖ),
                                (mⱼx + dⱼ)(mₖx + eₖ),
                                (mⱼx + eⱼ)(mₖx + dₖ),
                                (mⱼx + eⱼ)(mₖx + eₖ)))
```

However, which of the four alternatives is the minimum and which is the maximum might vary according to x. The first one expands out to $m_j m_k x^2 + (m_j d_k + m_k d_j)x + d_j d_k$, which can clearly change sign twice over some interval and possibly have a local maximum in the middle.

The quadratic term is the same across all four alternatives, but the linear term varies, and the combination of those two means that the location of the parabola's extremum can vary between the four; each of the four might be the minimum at some point in the interval!

In most cases, both j(x) and k(x) will have known sign on the interval — they are known to be either positive everywhere in [a, b) or negative everywhere in [a, b). If *either* of them meets this criterion, the solution is simple; if j(x) is known to be positive (i.e. $m_j a + e_j > 0$ and $m_j a + d_j > 0$), then XXX

Hmm, the answer to this isn't clear to me right now.

Coming back later to rethink this:

So the idea is that you represent a value y_i computed for some interval $\{x_0 \in [p_0, q0), x_1 \in [p_1, q_1), \ldots\ x_n \in [p_n, q_n)\}$ as some function $\Sigma_i x_i m_i + [d, e)$ associated with that interval, where "[b, c)" means "some unknown number e such that $b \le e < c$"; the representation of that value then is

$(d, e, [(m_0, p_0, q_0), (m_1, p_1, q_1), \ldots (m_n, p_n, q_n)])$

Then, when values are valid over the same interval, we can do relatively straightforward things for arithmetic operations, although I haven't worked out the details in cases of indeterminate signs for multiplication and division. When they are valid over different intervals, we can intersect their ranges and apply a subdivision operation.

In effect here what we are computing with are piecewise-linear approximations of functions with error bounds per piece, rather than individual intervals.

To take a concrete example, in raytracing, we have x_0 and x_1, the pixel coordinates, and we want to compute colors (r, g, and b) as functions of those two values. An approximate solution here is perfectly fine as long as the approximation isn't too wildly wrong. (And in fact common rendering algorithms are wildly wrong for some pixels, the ones that are close to object boundaries.) So we can start with a conservative approximation for, for example, $x_0 \in [0, 1024) \wedge x_1 \in [0, 768)$ — we can calculate the ray directions as intervals, calculate which objects could possibly be intersected by those rays, and calculate what range of colors and illumination those objects could potentially result in, eventually coming up with some kind of conservative approximation for the color gradient of the whole scene.

Once we have this conservative approximation for the scene as a whole, we can compare its error bounds to the error bounds we want to accept for our colors. If it's too large, we break up the interval into subintervals and redo the computation for each subinterval. In the past, doing things like this, I found that dividing into three subintervals was better than dividing into two. Unfortunately the representation gives us no clue as to which dimension is most promising to subdivide. In this case that is quadratically bad in some cases, but with many dimensions it is exponentially bad.

While you're doing the calculation, though, you have some idea how much of the error comes from each independent variable. If you could somehow include that information, with something like forward-mode automatic differentiation, you would have a much better chance of choosing good subdivision dimensions to reduce the error.

However, if you have both many independent variables and many dependent variables, this will be quadratically large. One possible solution to this problem is to redefine the indices of the dependent

variables as separate independent variables, as in the raytracing example in which ultimately there are only three dependent variables, r, g, and b. This, I think, makes the *reverse-mode* automatic differentiation problem computationally tractable, which should help some with the problem of picking which dimension to subdivide.

Topics

Measuring the moisture content of coffee and other things with dielectric spectroscopy

Kragen Javier Sitaker, 2019-07-16 (updated 2019-07-17) (28 minutes)

How can you measure the humidity of coffee, rice, beans, yerba mate, flour, oatmeal, polenta, wood, clay, concrete, soil, plastics, and so on?

The measurement techniques described in Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) can be applied to humidity measurement, but there are additional issues.

The importance of humidity

On Earth, humidity is a crucially important property for many purposes.

Food

Dried food stored with too much moisture can grow mold, and in particular mold on improperly stored legumes is one of the major causes of human liver cancer in the world, by way of aflatoxin contamination in the food eaten by the humans. Also, extra moisture in food represents extra weight that must be stored and sometimes moved, and food is often sold by weight, so standards of marketability impose maximum humidity contents on foods.

Deep freeze (p. 1465) talks a bit about moisture content of marketed dry food (11% or less for marketed soybeans, for example). Food storage (p. 2706) talks a bit about moisture as a factor in food decay.

Wood and wood products

Wood expands and contracts considerably according to humidity, and so must be dried to roughly the proper humidity before being made into things, so that it doesn't expand or contract too much after being assembled; also, ever since the development of lignolytic enzymes such as lignin peroxidase (probably at the end of the Carboniferous), already-installed wood is subject to attack by fungus if it's humid.

Additionally, humidity in installed wood can indicate the presence of termites, which will destroy the wood, or a water leak, which may destroy things in contact with the wood, as well as permitting fungal growth, as above.

Wood products like fiberboard and particle board are similar to wood in this way, but some of them additionally suffer direct degradation from being wetted. Most varieties of MDF expand about 60% in the cross-grain direction when they get wet, losing most of their already rather pitiful strength in the process, and the adhesives used in in particle board are also often degraded by getting wet, even if the moisture doesn't last long enough to permit mold attack.

Dry wood and wood products also provide substantial insulation

values, while their wet versions do not.

A design sketch of an air conditioner powered by solar thermal power (p. 2233) talks a bit about various organic hygroscopic substances, including wood, and how much water they can absorb: dry wood contains about 12% moisture, while wood in equilibrium with a humid atmosphere can contain up to 25% to 30% humidity.

Clay

The physical properties of a clay body† prepared for pottery-making depend sensitively on its moisture content. In a couple of percent near 20% water by weight, it transitions from brittle "dry" clay, which is still cool to the touch because of the heat-pipe effect as water evaporates from near the humans' fingers, to flexible "leather-hard" clay, which can still be broken, to fully plastic clay which can be deformed arbitrarily as long as it's kept in compression. (Like ductile iron, it still eventually reaches brittle fracture under sufficient tensile deformation.) A fully plastic clay body is a fucking amazing material for forming: it requires very little force to deform it, and because its elastic deformation is so small as to be very difficult to measure, which means that once you form it into a shape, it has very little springback as you remove the forming tool.

There's a further humidity reduction from "dry" to "bone-dry", at which point the object no longer feels cool to the touch, and is ready for firing. Bone-dry clay is still brittle, but considerably stronger than merely "dry" clay.

The precise moisture percentages at which the clay body transitions between these states vary fairly widely depending on the types of clay involved and the other ingredients.

As the clay body dries, it contracts, which sets up stresses in the object, which can deform it, and provokes some dimensional imprecision — the contraction is typically anisotropic due to not only anisotropic orientation of clay grains but also because of external forces present during the contraction process, so the shrinkage is somewhat unpredictable. Nearly all of this contraction is between the "fully plastic" state and the "leather-hard" state; there is dramatically less contraction from "leather-hard" to "dry", and none from "dry" to "bone-dry", and almost none from "bone-dry" to bisque-fired (sintered) ceramic. (Glaze-firing densifies the ceramic further, producing further contraction.)

In the plastic state, the clay is sticky and tends to adhere to whatever you use to form it, pulling it out of shape as the forming contact ends; once it is leather-hard, it is no longer sticky. Plastic clay in a dry atmosphere forms a thin leather-hard layer at the surface which can serve to ameliorate this stickiness.

So, if you form clay in the fully plastic state, you get substantial contraction and consequent imprecision upon drying to leather-dry. If you form the clay in the leather-hard state, you can get near-net shapes, but you are very limited in the deformations you can achieve. You can also *cut* leather-hard clay with a blade, getting glassy-smooth surfaces, although these do not survive bisque firing. Once the clay is dry, it can be further cut to shape with abrasive processes, at the risk of shattering the brittle piece.

So, precise measurements of clay moisture content, down to a fraction of a percent, are very useful for controlling manufacturing

processes, particularly automated manufacturing processes. The humidity at each of these transition regions depends on the precise contents of the clay body, but if you're using a well-controlled clay body, you can calibrate your humidity levels to that clay body and get reproducible manufacturing results.

† For making pottery, we mix clays with other materials, including of course water, but also sand, other "tempers" such as grog (powdered fired clay), flocculants, deflocculants, and organic gums, in order to balance the properties of plasticity, green strength, contraction, firing temperature, and strength after firing; this mixture is called a "clay body", although sometimes in the above I've sloppily called it "clay". Pure clay contracts on drying considerably more than commonly-used clay bodies do, and its green strength and even fired strength are much lower.

Concrete

Concrete needs water to harden, but I think that if it's too wet, atmospheric carbon dioxide can't penetrate, which slows the hardening process. If it dries out before hardening fully, it can become crumbly and impossible to harden. So, during hardening, it's potentially beneficial to monitor the moisture.

Also, continued exposure to water can degrade concrete, particularly if the water contains high concentrations of, for example, chloride or hydronium ions. And, as with wood, moisture in concrete can be a sign of water leakage, which can eventually result in damage to other objects if it continues. If the moisture is sufficient to saturate the surface of the concrete, it usually becomes very visible to the humans by darkening the concrete, but if the surface is kept somewhat dry by exposure to air, moisture in concrete can be entirely invisible.

So, monitoring moisture in concrete is useful both during hardening and long afterwards.

Soil

Soil moisture is crucially important for plant growth, because if there isn't enough moisture in the soil, plants can't suck it out of the soil, so they stop growing and eventually die. Also, if the moisture levels are too high, you get two kinds of fungal problems and a bacterial problem: too much water can suffocate symbiotic mycorrhizal fungi, which are extremely beneficial to land plants (although some land plants can survive without mycorrhiza); too much moisture can help non-symbiotic fungi to eat the plants; and, without access to nitrogen from air, rhizobial bacteria cannot fix nitrogen.

However, the particular level of soil moisture needed for plant growth depends on the salinity of the soil; roots have a harder time pulling moisture out of saline soils due to higher osmotic pressure.

Plastic

Many common plastics, notably including PET, nylon 6, and PLA, are hygroscopic; they absorb water from the air. In ordinary use, this is rarely a problem, or is even beneficial, but it has a couple of important effects on melting or hot-forming the plastics.

First, the absorbed water affects the plastics' specific heat, generally increasing it, so the plastic heats up more slowly. Second, although

these plastics are relatively stable in the presence of water at ordinary temperatures, they hydrolyze at the temperatures used for forming or melting them. (Also, PLA in particular, if kept wet, hydrolyzes to lactate over several months at body temperature, and several years at room temperature.) So they must be dried before molding, which is done by heating them to a lower temperature for several hours.

Static permittivity

Electric charge produces an electric field according to Gauss's law, $\nabla \cdot E = \rho/\varepsilon_0$, where ρ is the charge density and ε_0 is about $8.8541878 \times 10^{-12}$ A² s⁴ / kg m³. But, here on Earth, the electric field we observe is always lower than this prediction — usually about 0.06% lower in air, almost 5 times lower in glass, and almost 90 times lower in pure water. That is, to get a given field, we need about 0.06% more charge than this law would predict, or about 5 times as much charge if the region of interest is mostly filled with glass, or almost 90 times as much charge underwater. (Of course, unless our water is very pure, the charge will leak away over time through electrolytic currents, but we can do this measurement pretty quickly, in much less than a nanosecond, so the leakage is small.)

We explain this by a phenomenon called "electric susceptibility": we suppose that the molecules of the substance have their own electric fields, and they interact with the applied field. For example, in water, the two hydrogen atoms are kind of on the same side of the oxygen, and they have a small positive charge, while the oxygen has a small negative charge, less than an electron. So if we put a negative charge on the left of it, it attracts the hydrogens and repels the oxygen. Because water is a liquid, the water molecule is free to turn around, it tends to turn so that its hydrogens are on the left and the oxygen is on the right ("dipole relaxation"). So then the water molecule's own tiny electric field is subtracted from the electric field of the negative charge we put to its left, and in fact it cancels almost 99% of it under ordinary conditions. So we need 90 times as much charge to get the same electric field as we would predict from Gauss's law.

There are a few different ways that charges can move around inside the substance ("polarize") in response to the applied electric field. For example, in addition to molecules turning around, ions can move around ("ionic conduction"); crystal structures can deform ("ionic polarization"); electrical charge can flow to different parts of a molecule, especially a conjugated compound; and so on. As a general principle, though, because opposite charges attract each other, all of these effects *cancel* the field somewhat; they never make it stronger. The cancellation is never complete, either, because as it approaches completeness, the leftover field's influence on the charges approaches zero, so other influences on them are more important, like thermal motion. So we would expect that usually heat would make this susceptibility go down, and in fact we do see this with water: at 100°, we only need about 55 times as much charge as Gauss's law predicts, instead of the 88 times as much charge we need at 0°.

So usually when we apply Gauss's law, we apply it in the form $\nabla \cdot E = \rho/\varepsilon$, where the ε is a "permittivity" which is ε_0 multiplied by a "relative permittivity" or "dielectric constant" that includes the susceptibility of the medium, as well as the inherent "vacuum

permittivity" ε_0. So, for water at 0°, we say the relative permittivity is 88, and for water at 100°, we say it is 55.

(This effect is also the main reason light travels at different speeds through different substances, which is why transparent substances both refract and reflect light at their surfaces. Rutile — titanium dioxide — refracts light so strongly because its permittivity is even higher than water's, even though it's solid.)

Porous, dry organic materials like paper, wood, coffee, beans, and rice have relative permittivities around 4, which is much smaller than water's permittivity of 88. By coincidence, quartz sand's relative permittivity is also about 4 (it's 3.9) and so is concrete's (it's 4.5). So, if these materials absorb water, their permittivity goes up substantially.

Static relative permittivity is just an approximation

Consider the polarization of water again, though. It happens by turning the water molecules around so that their hydrogens are predominantly on the side toward the negative charge and their oxygens are on the side toward the positive charge. It happens that this effect is close to linear under normal circumstances: slowly applied fields that are quite small compared to the enormous electric fields inside the molecule. But, as you can imagine, this linearity breaks down under other circumstances.

One aspect of this is that it takes a certain amount of time for the molecules to come into this alignment, and some of the energy of the applied field is lost in the process — the molecules in liquid water are shaking around under the influences of one another's fields, and so if you apply a rapid enough jolt of electrical field, they won't respond fast enough. So as the time we're considering goes to zero, or frequency goes to infinity, the susceptibility also goes to zero.

(In optics, this variation of permittivity with frequency is called "chromatic aberration" or more generally "material dispersion".)

So you could imagine rapidly bringing some charge into proximity with some water, over a short time period, like an attosecond. At first you would observe the whole field predicted by Gauss's law in its pure form, the ε_0 form, but then if you left the charge there for a while, gradually the field would decay down to its usual level of about 1% of its original value. Most of the energy has gone out of the field. Where did it go?

And of course the answer is that it went into heating up the water molecules: as they rotated around to come into alignment with this sudden jolt of electrical field, they jostled against each other and gained some kinetic energy. And this is how we heat up water in a microwave oven, by applying an electrical field that goes the other direction every 210 picoseconds or so. This loss of electrical energy to heat is called a "dielectric loss". The dielectric loss is often combined with the dielectric constant into a single complex number called the "complex permittivity".

A thing to note there is that the different susceptibility mechanisms operate on different time scales. Ionic conduction is a great deal slower than dipole relaxation, for example, and water always contains some ions; water absorbed by organic matter or soil usually contains

an enormous quantity of ions. It happens that ions become more mobile when the temperature is higher, so the low-frequency permittivity of moist organic matter is dominated by ionic conduction, and so its permittivity goes *up* when it gets hot, instead of down like pure water's, at least until it's close to boiling. But, for the same reason, the permittivity of water with lots of ions drops sharply with frequency, much more sharply than pure water's, and above a few tens of megahertz, its permittivity becomes dominated by dipole relaxation and drops with temperature, for the reason explained above.

So, for example, carrots contain so many solvated ions that, at low frequencies, they have a relative permittivity of about 600 at 25°, which increases to about 850 at 45°, but around 100 MHz the permittivity of carrots is nearly invariant with temperature. Navel oranges, which contain many fewer ions, only have a relative permittivity of about 200 at 25°, which increases to 250 at 45°, and the point at which their permittivity becomes insensitive to temperature is about 40 MHz. (All of this is according to Nelson's 2006 paper, "Agricultural applications of dielectric measurements.")

(I am somewhat skeptical of the precision of these numbers; theoretical considerations suggest that they come from Maxwell–Wagner–Sillars polarization, which can give you arbitrarily large permittivities because the charges can be separated by arbitrarily large distances.)

Not only the permittivity but also the dielectric loss varies with frequency; the dielectric loss, too, falls with frequency in the limit, but may locally rise over some frequency range.

Another way this linear approximation can fail is with very strong fields. At the macroscopic field strengths we usually observe, the linear approximation is very good, but you can easily see that once, for example, all the water molecules are all pretty well lined up with the applied field, they can't align themselves any further to cancel an even stronger applied field; any further electrical susceptibility must be due to other effects such as the molecules bending, or charge moving around on them, or ions moving through the water, which are much weaker effects. So at high enough field strengths, the relative permittivity of any substance drops back to almost 1, just like its relative magnetic permeability. In optics these deviations from linearity are called the "Kerr effect", and they are one of a few ways to get nonlinear interactions between light beams or to electrically control light beams at subnanosecond timescales. (At even higher field strengths, though, the applied field is stronger than the fields that hold the molecules together and the substance will ionize. This alters its electrical and optical properties more radically.)

This nonlinearity is also very important in practice with ceramic capacitors; among the highest-dielectric-constant materials available are the piezoelectric ceramics barium titanate and lead zirconate titanate, which can have relative permittivities (dielectric constants) in the thousands, and they make possible high-capacitance chip capacitors. But when fully charged to their rated voltage, the capacitance of these capacitors can drop by almost half — the field is almost high enough to cause avalanche breakdown of the perovskite structure, and the permittivity of the dielectric drops dramatically at such high fields.

However, I don't think these strong-field effects are important to the moisture-measurement problem.

Another direction in which the linear approximation fails — for some substances — is in the limit of long time periods. The energy stored in a capacitor at a given field strength is proportional to the permittivity of the capacitor's dielectric; by using a higher-permittivity dielectric, you can store more energy in the same space. So why don't we use water-dielectric capacitors for everything except cases where miniaturization is paramount, since water is so much cheaper than lead zirconate titanate? It turns out that, in the limit of large times, water will always break down in a constant electric field, although the time it takes extends exponentially as the field is reduced. So water-dielectric capacitors do work, and they are used for some systems that need to release an enormous amount of energy very quickly, but they cannot hold their charge for a long time.

A third direction in which this approximation fails has to do with anisotropic materials, which can have greater permittivity in some directions than in others.

Applying permittivity variation to moisture measurement

So, suppose you have some ground coffee, and you want to know how much moisture it contains. The most straightforward thing to do is to place an insulated metal plate in contact with the coffee — for example, a copper pour on a printed circuit board, covered with solder mask — and charge the plate up to a given voltage, like 3.3 volts. The amount of charge needed for this will depend on the permittivity of the coffee. To measure the charge, you can allow the plate to discharge through a known, or at least constant, resistance, and measure the time constant of the decay curve. This gives you a measurement which depends on the moisture content of the coffee.

(For a sensor that isn't connected to earth ground, you might actually need two equal-size metal plates, one treated as "ground", to reduce variation due to the floating voltage of the instrument.)

The problem with this is that, as explained above, the measurement depends not only on the moisture, but also on the temperature. It also depends on how tightly packed the coffee is, because if there's more air mixed in with the coffee, that will lower the measured permittivity. And in situations where the measurement is being taken by placing some kind of handheld sensor against the coffee (or wood or whatever), rather than dumping the coffee into the sensor, you have the additional variables of the size of the air gap between the sensor and the coffee, and the percentage of the plate that is in contact with the coffee.

(I suspect, but am not certain, that all three of density, contact area, and air gap size will have essentially the same effect, in which case we can lump them together into a single unknown, "quantity of material".)

In an alternative arrangement, you run a sinusoidal AC voltage at a controlled frequency into the sensor plate (or plates) and measure the AC current that ensues, thus giving you a measure of capacitance, which varies linearly with permittivity; the phase shift between the

voltage and the current tells you how large the dielectric losses are. If you know enough about the substance whose humidity you're measuring, you can choose the frequency where its permittivity doesn't vary with temperature, only with density and humidity, and then you can use the dielectric losses to determine how much of the substance you're measuring.

More generally, at that point, it's just a matter of estimating two unknowns (moisture content and quantity of material) from two measurements (capacitance and loss factor).

By sweeping the frequency over a wide range, you can obtain a whole spectrum of complex permittivities at different frequencies, which in theory can provide you with an arbitrarily large number of possibly independent measurements. This could allow you to estimate a larger number of unknowns, such as temperature, moisture content, density, air gap size, and contact area, and perhaps even to distinguish between, for example, coffee, human flesh, and rice.

Another possibility is to use an array of smaller "pixel" contacts; for example, a 16×16 array of 4 mm × 4 mm contact areas separated by 1-mm gaps would enable an independent permittivity measurement on each pixel. This would reduce the problem of unknown contact area (where an unknown fraction of the sensor plate is in contact with the material being measured) and also permit the generation of images.

In the cases of soil and concrete, an important additional unknown is salinity. In soil in particular, higher salinity increases permittivity and conductivity, but makes water less available to plants, while higher moisture increases permittivity and conductivity, while making water more available to plants. So if we want to measure the soil's need for irrigation, but don't know the salinity, a simple permittivity measurement is insufficient — we need to estimate both the moisture content and the salinity, and probably the density and temperature as well. For irrigation measurements in particular, it may be feasible to supplement the permittivity measurements with resistivity measurements and thermometer measurements, if you're leaving the sensor embedded in the soil long enough to measure its temperature to some degree of precision.

An alternative approach to controlling for temperature effects is to intentionally heat the material being measured during the sensing process. The dielectric losses being measured will deposit a small and approximately known amount of heat into the substance being measured; since some of the temperature coefficients of permittivity we're talking about above are as high as 2% per degree (20000 ppm/K), even small temperature shiffts might be detectable, and might provide a much-needed additional dimension of variation. That is, you can map the complex permittivity of the sample not just at many frequencies but at many frequencies at many temperatures.

Harmonics and analog electronics

If you're generating the stimulus signal from something like an AVR Arduino, the signals you can generate are somewhat limited — the AVR can easily generate square waves of up to about 4 MHz, and I think that by hacking the SPI you can generate a somewhat noisy 8-MHz square wave. More modern cheap microcontrollers like the STM32L0 and STMF0 lines (see Notes on

the STM32 microcontroller family (p. 3176)) can easily generate square waves and pulse trains at a few tens of MHz. However, as mentioned earlier, the permittivity of moist organic material has important variations up to and above 100 MHz — in particular, you often have to go that high for the dipole-relaxation mechanism to be the dominant contributor to the dielectric constant.

However, an 8-MHz square wave has significant harmonics at 24 MHz, 40 MHz, 56 MHz, and 72 MHz. Perhaps a small amount of analog circuitry connected to the outside of the microcontroller could filter out a particular harmonic for later conversion.

However, if you're doing *that*, maybe you should just offload the whole oscillation and demodulation task onto analog electronics, reserving the microcontroller for just control.

Topics

- Electronics (p. 3430) (138 notes)
- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Metrology (p. 3579) (18 notes)
- Ceramic (p. 3371) (17 notes)
- Drying (p. 3417) (7 notes)

Assigning consistent order IDs

Kragen Javier Sitaker, 2015-09-03 (3 minutes)

In a Java system I'm working on where we simulate trading, we end up with order IDs in the logfiles. Unfortunately, Java's default is to use randomized memory addresses or something for object hash codes, which vary randomly from run to run. This means we can't usefully diff our logfiles. Also, the IDs are 6 hexadecimal bytes.

I think a more useful way to assign order IDs would be with dates: represent the most useful aspects of the date in a short string. In particular, I'm thinking the last digit of the year (since our backtesting simulations cross years), some uniquish ID for the day inside the year (a bit over 8 bits of data, so we can expect a minimum of two characters for this), and then a serial number inside the day.

There's a standard notation for something close to this in the futures markets: ESZ4 is the ES contract with delivery in December 2014, with the letter Z identifying the month, in some sense identifying the fortnight within the year. If we use letters for fortnights in this way, we have 14 or 15 days to discriminate among with an additional letter or number to get to the day, which is entirely doable. Ideally this second letter will be lowercase, and avoid easily misread letters like l, i, and o. Then we can number the orders inside the day with digits.

This gives us IDs like Z4d3, for the third order on 2014-12-04. This is a huge improvement over the current situation: it's a third shorter, human-readable, and consistent from run to run.

From http://www.cmegroup.com/product-codes-listing/month-codes.html:

January	F
February	G
March	H
April	J
May	K
June	M
July	N
August	Q
September	U
October	V
November	X
December	Z

If we wanted to assign two letters per month, sequentially, consistent with the above, we could almost do it:

January	E F
February	G ???
March	H I
April	J ???
May	K L
June	M ???
July	N P

```
August      Q R
September   S T
October     U V
November    W X
December    Y Z
```

If we add B for the second half of February, A for the second have of April, and C for the second half of July, then we can do it, at some cost to consistency. If we're willing to accept that kind of inconsistency, then at the cost of a little more of it, we could eliminate another one: in the above chart, the CME letter is sometimes the first half of the month and sometimes the second. If we make it always the first half, and use out-of-sequence letters for the others, then we end up with this (the six out-of-sequence letters marked with *):

```
January     F B *
February    G C *
March       H I
April       J D *
May         K L
June        M E *
July        N O
August      Q R
September   U P *
October     V W
November    X Y
December    Z A *
```

Topics

Convolution with intervals

Kragen Javier Sitaker, 2015-09-07 (1 minute)

If we know that on $[x_0, x_0 + \Delta x)$ function f is always inside [a, b), and that on $[x_1, x_1 + \Delta x)$ function g is always inside [c, d), what can we say about that interval's contribution to their convolution (f * g) on $[x_0 - x_1 - \Delta x, x_0 - x_1)$? (I'm assuming here that this is the right interval, and I might have gotten it a bit wrong.)

I think that the contribution must necessarily be in $[ac\Delta x, bd\Delta x)$, if we assume that all of (a, b, c, d) are positive. (The other cases are important to cover in practice but conceptually don't matter much.) This gives us both an upper and a lower bound. We can extend this to the case of infinite Δx specifically and only in the case where either $a = b = 0$ or $c = d = 0$. Applying this approach to all possible intervals allows us to bound the convolution of two functions by working from an interval-wise decomposition of them.

It might be useful also to know the *integrals* of the functions over the relevant intervals, or at least bounds on them. The contribution to the convolution cannot be more than the product of the integrals (again, assuming all positive).

Topics

drag-and-drop calculator for touch devices

Kragen Javier Sitaker, 2015-09-03 (5 minutes)

I wrote an RPN calculator in DHTML, and it runs on my iPhone, but it isn't very usable. One problem is that it's misidentifying some keypresses (apparently shiftKey is false when you press the * on the on-screen keyboard on both Android and iPhone, but the keyCode is the same as 8, and so * comes out as 8, because I was too dumb to use a library) but it has bigger problems. The keyboard eats up a lot of the screen real estate; predictive text like Swype is totally incompatible with instant feedback; you have to switch keyboard modes to get to different operators; there are no arrow keys, much less a Ctrl key to use with the arrow keys; and pressing the wrong key is common, so undoing an error needs to be extra easy.

Fundamentally the issue is that command keys are a bad idea on these touchscreens. They have the usual problems with command keys (they're not discoverable, not selectively available and thus invisibly modal) but also the touchscreen offers very poor precision for keypressing. Instead, it's optimized for dragging, and pointing at things. You could use keys (ideally transparent ones!) for data input, but you probably shouldn't use them for editing.

I think you can usually get about 6 rows of about 4 buttons on a cellphone touchscreen, so a keypress can convey about 5 bits of information. But a drag stop can typically indicate a position to about 8 bits of horizontal precision and 8 bits of vertical precision, or 7 bits each easily — with the caveat that you can't see what the fuck is under your finger.

The ideal drag-and-drop behavior of mouse-driven systems is that, when you start dragging a thing that can be dropped in places, the places it can be dropped will light up, and in some cases emerge from invisibility, in order for the user to be able to discover them. Then you can carefully position the dragged item over the drop target and release it. If a given on-screen object wants to support several different actions from having the same thing dropped on it, those different things need to be different drop targets, perhaps positioned in active zones around it, which can become active only when a drag is in progress. Your drop action is the utterance of an RDF triple: the subject, perhaps, is the dragged item; the object is the object from which the drop target emerged; and your choice of drop target — a delta vector between the drop position and the object of your utterance — provides the verb.

(Naked Objects pioneered this approach to method invocation for business data processing UIs.)

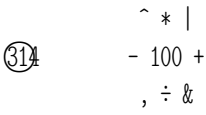
A problem with this on a small screen is that your drop targets need to be damned big for you to be able to see them around your finger, which means you can't have very many of them. Maybe you can dynamically zoom drawers or things as you drag around in order to effectively get more screen real estate, but man. So much loss.

One possible fix to this would be to have the "drop targets" emerge from the *object being dragged* rather than the stationary object; they can

protrude in all directions from under your finger, sort of like a pie menu.

In the calculator case, perhaps you have a number 314 and a number 100 that you would like to combine:

If you start dragging the 100 around, binary operators can begin to protrude from it, while the 314 lights up as a potential target:



and if you position one of those operators over the 314, it will light up; if you then release your touch, the two numbers can combine into a new formula. You can take advantage of the precision of dragging in order to unambiguously select the 314 as the target; if it is embedded in a compound formula, that formula might magnify as you approach it in order to reduce collisions.

Of course, if you could manage to drop the 100 directly on the 314, you could then pop up a pie menu of binary operators, but this seems both more cumbersome and less discoverable.

A different approach, taking advantage of the possibilities for multitouch interaction afforded by modern touchscreens, would be to use two separate fingers for different things, instead of a single drag-and-drop gesture.

Topics

• Human–computer interaction (p. 3493) (76 notes)
• Multitouch (p. 3591) (12 notes)
• Calculators (p. 3362) (11 notes)

Inflatable stool

Kragen Javier Sitaker, 2014-04-24 (6 minutes)

I was at a bus stop today and sat on a discarded inkjet printer. It occurred to me that it would be nice to have brought a stool to sit on while awaiting the bus, but stools are typically heavy and bulky; I'd want something that would fit into my purse (about a liter) without weighing much (the purse weighs about 1kg).

Generally stiff things that can resist compression are kind of heavy, but materials can resist a lot of tension even while being quite light. A simple rope of webbing, like a seatbelt, that could hook onto the roof of the bus stop in two places would make a practical, relatively comfortable, and quite strong hanging seat.

Another way of avoiding the need to carry around materials that can resist compression is to use air to support the compression, as in a balloon. The balloon skin only needs to resist tension, so you can make a pretty large balloon that doesn't weigh much. This couch, almost two meters long, weighs under 3kg: http://www.amazon.com/Blow-Inflatable-Furniture-Sized-Couch/odp/B00245MIAY. A comparably-sized couch made from foam might weigh 30kg.

But how little material could you get away with? I weigh about 100kg (980N) and sit on an area that's about 40cm×40cm, or 0.16m², so an air pressure of 6100 N/m² (6.1 kPa) is sufficient to hold me up. Ambient air pressure is about 101 kPa, so you only need to compress air very slightly to get it up to 101+6 = 107 kPa, say, by sitting on it.

How much material would you need in the skin of the inflatable stool to resist that air pressure? Very little. You can estimate hoop stress for a thin-walled cylinder as Pr/t, where t is the thickness. So the hoop stress of a 25-cm-radius cylinder containing 6 kPa(g) will be 6 kN/m² · 0.25 m / t, or 1.5 kN/m / t.

The biaxially-oriented polyethylene terephthalate film used for mylar balloons and potato-chip bags has a tensile strength of about 200 MPa, almost as strong as low-end aluminum alloys and a lot lighter and more flexible. It's commonly available in 10μm to 100μm thicknesses. To find the thickness needed to contain this pressure, we solve

```
200 MPa = 200 MN / m² = 1.5 kN/m / t
t · 200 MN / m = 1.5kN
t = 1.5 kN / (200 MN / m) = (1.5 / 200 000) m = 7.5 μm
```

If we figure on using a film of several times that thickness, say 20 μm, we should be pretty safe from bursting if we don't totally jump on the thing. How much would that weigh? $2\pi r^2 + 2\pi rh$, figuring 50cm height and 25cm radius, gives us a total area of 1.2m²; so 20 μm would be 24 cc. (7.5 μm would be 9 cc.) PET weighs about 1.38g/cc, so this would be about 33 g of plastic. This is acceptably small.

That's fine as far as it goes, but trying to sit on a paper-thin mylar balloon at bus stops will have some serious practical problems. In the summer, if you're wearing shorts, it will stick to your thighs in an

unpleasant way; but also, the part of it pressing against the rough cement with 6kPa and sliding around as you move will get cut up.

You can solve both of these problems by gluing woven cloth disks to the top and bottom of the plastic, probably the coarser the better. Coarser cloth will weigh more but allow better air circulation and protect more fully against abrasion; as the cloth gets sufficiently coarse, you might want an additional somewhat finer cloth layer between your skin and the burlap or whatever, or between the bottom of the balloon and the ground burlap. This introduces an undesirable tradeoff between weight and performance. Adopting a pattern imitating the reticulate venation of dicotyledon leaves, with a few thick veins separating areas of finer venation, could ease this tradeoff.

The cloth is likely to increase the weight to perhaps as much as 100 g, but we aren't done yet. Now we face the question of how to inflate the thing. It holds about 0.1 m^3 (100 liters) of air, which is around 30 lungfuls; inflating it by mouth would be a lot of work to have something to sit on for only, one hopes, a few minutes.

You could carry around a gas cylinder to inflate it with, but this is impractical; 100 liters of CO_2 is about 200 g. You'd need to empty three 88 g paintball-gun CO_2 cartridges to inflate it fully, and those cartridges weigh over 400 g each, so you'd be carrying around 1200 g of cartridges. Even the 200 g of CO_2 itself is unreasonably heavy.

The most promising solution I've seen so far for inflating things like this is the Windcatcher, which uses a sort of funnel with a low-pressure one-way valve to help you entrain air as you blow into the entrance. I don't know how much it weighs, but they were selling a bag using it for US$40 to their Kickstarter backers; a larger air mattress using it weighs 840g, but I hope that's mostly the bag.

Topics

- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Furniture (p. 3465) (2 notes)
- Inflatable

Does SAAS make it harder to ship? I doubt it.

Kragen Javier Sitaker, 2007 to 2009 (7 minutes)

From a comment on the Smoothspan blog, by Damon Edwards of dev2ops.org:

A troubling trend I've noticed is how the benefits of "rock star" software development teams (small, highly skilled, highly motivated) are increasingly neutralized by poor operations.

In an ops heavy SaaS and on-demand world, the software development phase becomes an increasingly smaller part of an application's overall lifecycle. Time and time again we see great code sitting behind the bottleneck of QA, staging, performance testing, and then production deployment.

On the project plan, the "rock star" teams repeatedly deliver great code in record time... but at all but the smallest of enterprises, their "delivery" of code is a long way from where the business is actually realizing the benefit.

I don't know whether this is true or false, but if it's true, it represents the reversal of a trend touted by Philip Greenspun in 1998:

When I graduated from MIT in 1982, my classmates and I had but one choice if we wanted to get an idea to market: Join a big organization. When products, even software, needed to be distributed physically, you needed people to design packaging, write and mail brochures, set up an assembly line, fill shelves in a warehouse, fulfill customer orders, etc. We went to work for big companies like IBM and Hewlett-Packard. Our first rude surprise was learning that even the best engineers earned a pittance compared with senior management. Moreover, because of the vast resources that were needed to turn a working design into an on-sale product, most finished designs never made it to market. "My project was killed" was the familiar refrain among Route 128 and Silicon Valley engineers in 1982.

How does the Web/db world circa 1998 look to a programmer? If Joe Programmer can grab an IP address on a computer already running a well-maintained relational database, he can build an interesting service in a matter of weeks. By himself. If built for fun, this service can be delivered free to the entire Internet at minimal cost. If built for a customer, this service can be launched without further effort. Either way, there is only a brief period of several weeks during which a project can be killed. That won't stop the site from being killed months or years down the road, but very seldom will a Web programmer build something that never sees the light of day (during my entire career of Web/db application development, 1994-1998, I have never wasted time on an application that failed to reach the public).

And by Paul Graham in 2001:

One of the most important changes in this new world is the way you do releases. In the desktop software business, doing a release is a huge trauma, in which the whole company sweats and strains to push out a single, giant piece of code. Obvious comparisons suggest themselves, both to the process and the resulting product.

With server-based software, you can make changes almost as you would in a program you were writing for yourself. You release software as a series of incremental changes instead of an occasional big explosion. A typical desktop software company might do one or two releases a year. At Viaweb we often did three to five releases a day.

I see four possible interpretations:

• Damon Edwards is wrong, and in fact the software development phase is not "becoming an increasingly smaller part of an application's overall lifecycle" as a result of "SaaS", which is what Graham called "server-based software" and what Philip Greenspun called "a service", and in fact the per-user cost to deploy software is continuing to

shrink.

This is a plausible answer; Moore's Law continues to grind away giving us more MIPS per watt and MIPS per CPU, the cost of bandwidth was still falling last I checked, and perhaps more importantly, EC2 and S3 and Hadoop and Puppet and MogileFS and `aptitude` and Xen and `monit` and Cacti and `backuppc` and `nginx` and `perlbal` and `memcached` and Nagios and Erlang and Varnish and Capistrano are reducing the amount of human effort it takes to administer a given number of CPUs and increasing the number of users each MIPS can support. Maybe Damon sees operations becoming more difficult because the internet is still growing, and so the biggest services have to deal with more users now than in 2001 or 1998.

• The effort required to deploy software on centralized servers really is growing out of proportion to the effort required to write it in the first place, and that's because of the dependence on centralized servers. If the software could run on the machines of its users, the way Firefox or BitTorrent or Skype or Emacs does, the users would be the ones deploying it. Of course, they might still find that difficult, but that wouldn't be visible to the software authors; they would just see that nobody was using their software, not the hours of frustration expended trying to install it. But a lot of that deployment effort can still be moved into software (that's the point of InstallShield, `aptitude`, `easy_install`, RubyGems, CPAN.pm, Fink, Darwin Ports, Xen, AppEngine, and so on, although the diversity of items in that list suggests that the job is far from over), and with the effort that can't be, people can avoid duplication of effort by sharing solutions online.

Just because the software runs on its users' machines doesn't mean it can't be providing a networked service; consider BitTorrent or Skype or, for that matter, Sendmail, ircd, or INN.

• The effort required is growing, but not because of centralized servers. But I do not know of any other plausible candidates.

• A post by Jesse Robbins on O'Reilly Radar 3 suggests that some startups get their operations highly automated early on, so they can easily deploy their changes, while others screw up and end up with a mess, and spend lots of time on operations. If this is correct, then Damon Edwards is wrong in thinking that operations *inevitably* consumes a greater and greater proportion of the resources available; he's just working with dumb groups who dig themselves into big holes.

Topics

• Programming (p. 3658) (286 notes)
• Devops

When and why exactly should your code “tell, not ask”? That is, use CPS?

Kragen Javier Sitaker, 2014-01-08 (4 minutes)

Here I explain "tell, don't ask" in terms that explain when and why it's valuable, without the resorts to metaphor and simplistic heuristics often used in the teaching of object-oriented design.

One of the widely respected principles of object-oriented programming is "Tell, don't ask." In the unhelpful metaphorical language often used to describe OO design principles, it is said that, rather than asking objects about their state, you should tell them what to do: send them a message, don't ask them a query. Especially don't ask them a query and then use the results of the query to decide how to modify their state; if you're doing that, you should instead move the logic into that object.

The problem with all of this is that, so stated, it doesn't have enough context. What are we trying to achieve with tell-don't-ask? When might the drawbacks of telling exceed the drawbacks of asking, if ever? How can you tell when you've missed an opportunity to tell instead of asking? What do you do when you think you want the benefits of tell-don't-ask but you can't figure out how to get telling to do what you want? That's what this post is about.

CPS

In the land of compiler implementation, especially the Scheme school, tell-don't-ask is known as programming in " continuation-passing style", or "CPS". It turns out that, if you have polymorphic method calls or closures, and you don't have a stack-depth limit (or you have tail-call elimination) there is actually no limit to the expressiveness of a program that never uses any return values. You can take a program written with return values and transform it, completely mechanically, into a program where no function has a return value.

In a sense, this is obvious: machine code doesn't, generally speaking, have functions or return values, just subroutine call and return instructions. So to execute a program, a compiler must transform it, mechanically, into a form that doesn't need return values. Typically, return addresses and function arguments are pushed onto a stack in RAM, or stuffed into registers and then later placed on the stack. There are various ways to think of the resulting process. CPS takes the position that the return-address and local-variable data on the stack for the calling function is essentially a closure, which is invoked as the last thing that the callee does.

This shows how to transform a function call into a CPS function call: you take the rest of the caller, after the function call is to return, and package it up into a closure that you pass to the function. For example, given this JavaScript function:

```
function add_numbering(parent, domnode) {
```

```
  parent.numbering[parent.numbering.length-1]++;
  var number = parent.numbering.join(".") + ") ";
  insert_before(text(number), domnode.firstChild);
}
```

Suppose we want to transform the concatenation of the trailing ") " into CPS style. It's an invocation of the built-in string-concatenation operation +, which yields the return value we are going to turn into a text node and insert into the document. A generic CPS version of + would take an additional argument that is a closure to which to pass the concatenation result:

```
function concat(a, b, k) { k(a+b) }
```

And we can make the transformation by packaging up the tail end of the function like so:

```
function add_numbering(parent, domnode) {
  parent.numbering[parent.numbering.length-1]++;
  concat(parent.numbering.join("."), ") ", function(number) {
    insert_before(text(number), domnode.firstChild);
  });
}
```

If you've used Node.js this should look really familiar; Node uses explicit continuation-passing style for I/O functions. It does this because one of the freedoms CPS gives you (to be listed later!) is the freedom to suspend a computation and resume it later.

Generalizing this, if you have some function call

Topics

- Programming (p. 3658) (286 notes)
- Systems architecture (p. 3691) (48 notes)
- Program design (p. 3654) (11 notes)
- Object-oriented programming (p. 3606) (10 notes)

Nobody has yet constructed a mechanical universal digital computer

Kragen Javier Sitaker, 2014-04-24 (6 minutes)

The FIBIAC demonstrates an electromechanical machine that calculates the Fibonacci sequence under control by punched cards; later, Chris Fenton, the author, built a purely mechanical version, called The Turbo Entabulator. It's a three-counter machine. He asserts that the machine is not capable of calculating anything more complex than the Fibonacci sequence, but I think it may be able to go a bit beyond this.

Fenton recommends The Mechanism of Weaving, which describes how mechanical weaving machines worked in the late 19th century, and how mechanical computers should have worked.

An artificial muscle computer is a four-page paper from November that claims to describe a general-purpose computer built from 13 artificial muscle relays and a sliding-block mechanical tape memory, implementing the (2, 3) Turing machine proven to be universal by A. Smith, previously conjectured by Wolfram. Artificial muscle relays are electromechanical devices in which an electric artificial muscle compresses a "piezoresistive dielectric elastomer switch". However, this Turing machine is only universal if you first initialize its memory to a certain repeating pattern, requiring machinery that the authors have not implemented. The authors admit they were not able to program it to perform even an addition. In my view, this machine falls short of implementing universal computation.

None of the above have a significant amount of memory. The FIBIAC has almost 30 bits; the Turbo Entabulator has almost 10 bits; and the artificial muscle computer has 8. Konrad Zuse's mechanical Z1 had considerably more, 1452 bits, organized into 64 general-purpose and two special-purpose registers of 22 bits each. However, the Z1 fell short of universal computation because of its lack of control flow, even aside from the finite memory size that has been an unavoidable limitation of all computers constructed so far.

As I wrote before (in the post about heightfields and string) I think the threshold where a stored-program computer becomes interesting, e.g. capable of interpretation or compilation, is around 2K or 4K of RAM, that is, 16 to 32 kibibits. The Z1 was interesting with fewer bits because it, like Fenton's machines, ran its program from a separate read-only memory; the same could be said of microcontrollers, which commonly have less than 256 bytes of RAM --- some as little as 32 --- but invariably have at least 2K of program, and of the Atari 2600, whose RAM was 128 bytes but whose ROM could be up to 64K.

So an interesting question here is how to make a mechanical computer with 16 kibibits of memory. The Z1 had a mechanical latch for each bit, but it might be more practical to use some largish quantity of homogeneous material, like a disc or drum memory, that can be reversibly transformed.

It's not the only option, though. Consider a horizontal 64×64 matrix with a thread hanging from a spring at each point. Below, the threads are clamped in 128 clamps: one clamp that clamps all the threads on each row, and one that clamps all the threads on each column. The clamps maintain thread tension against the springs, so that if a spring happens to be stretched to some position, the clamp prevents it from contracting. If you open a single row clamp and a single column clamp, then a single thread is released, and its spring is free to contract --- unless something is pulling the other end of the thread to a new position before allowing the clamps to reclose. If the machinery can distinguish 16 positions for a given thread, that thread can be said to store 4 bits, and so we have our 16 kibibits.

You don't actually need 128 clamps; if you route the threads through something that keeps them from catching or moving laterally, such as a smooth pipe, you can address an individual thread out of 4096 with only 24 clamps, each of which clamps half the strings. Each string thus must pass through 12 clamps. For a sufficiently large number of threads, ternary addressing would slightly reduce the number of clamps, but 4096 threads is not large enough; you would also need 24 clamps for base 3 or base 4 addressing of 4096 strings. Base-4 addressing would halve the number of clamps any individual string passed through (to 6), and by the same token halving the number of strings in any individual clamp (from 2048 to 1024).

Given the additional complexity of the pipes and the probable difficulties in clamping 1024 strings, the optimal number of clamps for this number of strings is probably somewhere in between 24 and 128. Base 16, which would mesh particularly well with the base-16 contents I suggested above, would run each string through three clamps instead of two, and thus need 48 clamps, each clamping one-sixteenth of the strings (256 of them).

It might turn out that 256 strings is too many, and we need some kind of "chip select" as well as row/column/plane selection.

Topics

Rarely are function-local variables in Forth justified

Kragen Javier Sitaker, 2018-04-27 (10 minutes)

This was a big epiphany for me in Forth: you usually shouldn't use function-local variables. Instead, use "global" variables. This is true to some extent in PostScript, too, though less strongly.

First, a disclaimer: don't take what I say about Forth too seriously, because I've never written a significant program in Forth, only exercises like a self-compiling compiler. I've never done anything more than a few hundred lines of code in PostScript, either.

Traditional Forth lacks function-local variables. Function-local variables are crucial to Smalltalk, Lisp, and Algol-family programming for three reasons: lexical locality, recursion, and closures. Forth solves these in different ways, so **it's okay to use non-function-local variables instead**, and this has a benefit for factorability of the code. I would say "it's okay to use global variables instead", but one of the reasons it's okay is that they aren't really global in Forth.

Lexical locality

In Algol-family languages like Pascal or C, if a variable isn't local to a function, it's global to the entire program, which means it be modified by any code at all, including not only other files in your project, but even library modules you don't have the source code to.

By contrast, in Forth, a variable's scope extends only from the point of its declaration over the code that lexically follows it, up to the point where you switch to a different wordlist (or, in traditional Forths, vocabulary) or define another variable with the same name. This is not as small a scope as a C or Pascal function, but it's a much smaller scope than a C or Pascal program, so the variable name collision problem is manageable.

The point about another variable with the same name bears repeating: if you declare another variable with the same name in Forth, the old declaration stops being visible, and each part of the code uses the version of the variable that was visible when it was being compiled.

Languages like Python or Common Lisp are somewhere in between: a global variable (defined with `defparameter` or `defvar` in CL) is not global to the entire program, but just a single module. This reduces the seriousness of the problem.

PostScript, with its odd hybrid of Forth and Lisp semantics, is closer to the Algol family in this sense — its symbols ("name objects") are not module-scoped like Common Lisp symbols, nor are their scopes lexical as in Forth. You can dynamically add and remove dictionaries from the dictionary stack, but this is clumsy (it must be done in every function) and error-prone.

Recursion

In languages like Pascal or C, any function is potentially recursive, which means that if its local variables are not stored in stack-allocated

memory, they could get overwritten by recursive calls. Moreover, local variables are the only language-native mechanism provided for stack-allocation of memory; without them, simple things like recursive-descent parsers become major feats of software engineering.

In languages like Smalltalk and Python, the problem is even worse, because nearly any infix operator in your method could result in a recursive call chain that includes the same method. So even methods that are not intended by their authors to be recursive are likely to need to be re-entrant. (The gradual introduction of pervasive multithreading in the modern C ecosystem has had a similar effect.)

Also, Smalltalk, Lisp, Python, and functional languages like ML strongly encourage you to use recursively-defined data types.

The net effect of all of this is that, in these environments, function-local variables are vastly preferable to statically allocated variables.

By contrast, in Forth, recursion is very much the exception; recursively-defined data types are unusual, and functions can only call functions that are defined textually earlier in the program, except using `RECURSE`, `DEFER`red words, or similar mechanisms, which are unlikely to pop up without the author noticing them. And, if you want to save and restore the value of a variable for a recursive or potentially recursive call, you can do so fairly easily using the operand stack; `A @ B @ RECURSE B ! A !` saves the values of `A` and `B` during a recursive self-call, doing explicitly what Perl 4 or a dynamically-scoped Lisp would save local variables implicitly.

In PostScript, again, the situation is intermediate; recursive function calls are just as easy as in Lisp, and it's easy to define recursive data structures, although at least the native list-like data structure is an array, not a linked list. But PostScript shares with Forth relative ease at explicit saving and restoring variables on the operand stack. PostScript also doesn't have the tricky ad-hoc polymorphism that can give rise to unexpected recursion in Smalltalk and Python; it does use first-class function values pretty often, but rarely in ways that lead to unexpected recursion.

So function-local variables are not necessary to permit recursion in stack languages, and recursion is typically less of a danger.

(It's worth pointing out that function-local variables are not sufficient to make recursion safe. Recursive code can easily get stack overflows or suffer re-entrancy bugs related to nonlocal data structures, and so is prohibited in things like MISRA C.)

Closures

Pascal has very limited closures, which are also present in GNU C, although little-used. In vanilla C, the only way to get the equivalent of a closure — for example, for `qsort` — is to store the data it needs in statically allocated variables, which breaks re-entrancy and thus causes multithreading problems. (glibc provides a `qsort_r` function that takes a userdata parameter to solve this problem.)

Languages like (modern) Smalltalk, Python, Common Lisp, Scheme, Ruby, and JavaScript have closures and use them extensively. So function-local variables become a crucial mechanism for encapsulating state in objects of indefinite extent.

In the Forths that have added local variables, local variables do not provide closures; neither does PostScript support closures with local

variables, since PostScript's dictionary stack amounts to purely dynamic scoping, like Lisps before Scheme. Forth, instead, provides closures with the `CREATE DOES>` mechanism, which is explicit rather than implicit about what state is being stored. I don't know what the PostScript equivalent would be, although I bet you could hack something together with runtime code generation.

So function-local variables do not provide closures to augment the expressive power of PostScript or Forth, the way they do in many modern programming languages.

It's okay to use non-function-local variables in PostScript and especially Forth

In summary, function-local variables in Forth aren't needed for lexical locality, recursion, or closures, and when they're available, they also don't provide closures. And function-local variables in PostScript aren't needed for recursion, and they don't provide closures. So the advantages that make them a no-brainer in other families of languages are weaker or absent. What about the disadvantages?

Function-local variables are more costly in Forth or, especially, PostScript, than in other languages. Consider this particularly egregious case of stack abuse in PostScript (from Heckballs):

```
% Calculate distance from x1 y1 to x2 y2
/dist { 3 2 roll sub  3 1 roll sub  dup mul exch  dup mul add  sqrt } bdef
```

Probably a better way to write this is as follows:

```
/dist { 4 dict begin  /y2 exch def  /x2 exch def  /y1 exch def  /x1 exch def
        x1 x2 sub dup mul  y1 y2 sub dup mul  add sqrt  end } def
```

There are two interesting things to note here:

• The new definition is almost twice as long, 32 rather than 19 tokens, and includes a new error-prone `end` at the end. Also, it isn't clear that it's more readable, as the parameters are necessarily listed in reverse order.
• The new definition isn't as safe to use with `bind def`, because that introduces the danger that the variables `x1` and so on might accidentally be bound to some definition in the enclosing environment, rather than being local variables as intended. (As it happens, in this case there are no such variables, and `bind def` would have worked fine.)

Suppose that instead we use non-function-local variables:

```
/dist { /y2 exch def  /x2 exch def  /y1 exch def  /x1 exch def
        x1 x2 sub dup mul  y1 y2 sub dup mul  add sqrt  } def
```

The size penalty is somewhat less, although we run an even worse variable-collision risk, since this will clobber any values of x1, y1, x2, and y2 that any other function is using at the time — a problem much less likely in Forth.

We could conceivably refactor this into smaller pieces:

```
/is-p1 { /y1 exch def  /x1 exch def } bdef
/is-p2 { /y2 exch def  /x2 exch def } bdef
/dx { x1 x2 sub } def  /dy { y1 y2 sub } def  /sq { dup mul } bdef
/dist { is-p2 is-p1  dx sq  dy sq  add sqrt } bdef
```

In PostScript, you can still do this with function-local variables:

```
/is-p1 { /y1 exch def  /x1 exch def } bdef
/is-p2 { /y2 exch def  /x2 exch def } bdef
/dx { x1 x2 sub } def  /dy { y1 y2 sub } def  /sq { dup mul } bdef
/dist { 4 dict begin  is-p2 is-p1  dx sq  dy sq  add sqrt  end } bdef
```

You can't do that in Forth, any more than you can in C, which makes using function-local variables in Forth very costly to both the flexibility and the predictability of your code. To my mind, predictability is key to its readability.

So using function-local variables, although it's a viable strategy in PostScript, isn't nearly the slam-dunk obvious win that it would be in more conventional languages. In Forth, often, it's actively counterproductive.

Topics

- Programming (p. 3658) (286 notes)
- Python (p. 3671) (27 notes)
- Forth (p. 3461) (19 notes)
- Smalltalk (p. 3716) (12 notes)
- The PostScript programming language

An 8080 opcode map in octal

Kragen Javier Sitaker, 2019-08-28 (updated 2019-11-24) (11 minutes)

The 8008, 8080, 8086, i386, and amd64 instruction sets are, unfortunately, usually given in hexadecimal; but they are dramatically more readable in octal. The 8080 opcode map in particular can be drawn rather neatly using octal.

i386 and amd64 examples

Consider this segment of amd64 machine code:

```
400575:     ba 02 00 00 00          mov    $0x2,%edx
40057a:     be 64 06 40 00          mov    $0x400664,%esi
40057f:     bf 01 00 00 00          mov    $0x1,%edi
400584:     e8 a7 fe ff ff          callq  400430 <write@plt>
400589:     ba 01 00 00 00          mov    $0x1,%edx
40058e:     be 41 10 60 00          mov    $0x601041,%esi
400593:     bf 00 00 00 00          mov    $0x0,%edi
400598:     e8 a3 fe ff ff          callq  400440 <read@plt>
```

Here it is in octal:

```
400575:     272 002 000 000 000             mov    $0x2,%edx
40057a:     276 144 006 100 000             mov    $0x400664,%esi
40057f:     277 001 000 000 000             mov    $0x1,%edi
400584:     350 247 376 377 377             callq  400430 <write@plt>
400589:     272 001 000 000 000             mov    $0x1,%edx
40058e:     276 101 020 140 000             mov    $0x601041,%esi
400593:     277 000 000 000 000             mov    $0x0,%edi
400598:     350 243 376 377 377             callq  400440 <read@plt>
```

Here you can see that, for example, all the “load immediate” instructions are "27x", with "x" representing the register: 2 for %edx, 6 for %esi, 7 for %edi, and so on. As it turns out, there are precisely 8 registers that can be addressed in this way, corresponding to the 8 octal digits. And these register numbers are consistent across instructions; here we can see (in some i386 code, from httpdito), 0 representing %eax, 1 representing %ecx, 2 representing %edx again, and 3 representing %ebx (yes, the numbers are not in the same order as the letters):

```
804811c:    120                             push   %eax
804811d:    122                             push   %edx
804811e:    350 354 377 377 377             call   0x804810f
8048123:    132                             pop    %edx
8048124:    130                             pop    %eax
...
8048130:    102                             inc    %edx
8048131:    271 353 226 004 010             mov    $0x80496eb,%ecx
8048136:    061 333                         xor    %ebx,%ebx
8048138:    103                             inc    %ebx
```

```
8048139:        103                     inc     %ebx
```

By contrast, in hexadecimal, the immediate-load instruction ba 02 00 00 00 and the "pop %edx" instruction 5a represent %edx as "a", while "push %edx" and "inc %edx" are 52 and 42 respectively, representing %edx as "2". Moreover, note that in hexadecimal, both "push" and "pop" of registers are "5x", while in octal they are "12x" and "13x" respectively.

So this is the sense in which I say 8086, i386, and amd64 machine code are dramatically more readable in octal. The octal digits correspond neatly to the bitfields in the instruction encoding, in most cases. But even the 8086 opcode map is rather large.

The 8080 opcode map

By contrast, every 8080 opcode is a single byte, though some are followed by one or two bytes of immediate data, so a full opcode table is only 256 cells. It, too, is more comprehensible in octal than in hexadecimal, organizing the instruction set into four 64-byte "pages", although it has some cases where an inconveniently-located two-bit field identifies one of the 8080's 16-bit register *pairs* rather than a single 8-bit *register*. A simple permutation of the rows and columns ameliorates this.

(Beware! None of this has been tested, and it would be surprising if I had found all the errrors in it.)

The 0xy page is largely register-pair operations, occupying three or four columns, with three single-register operations, occupying eight columns:

```
 oxy   x
 B  D  H  M  C  E  L  A
 BC  DE  HL  SP  BC  DE  HL  SP
 0  2  4  6  1  3  5  7
 y  0  NOP  -
 2  STAX  STA  LDAX  LHLD  LDA
 4  INR (increment register)
 6  MVI (mov immediate)
 1  LXI  DAD (double add)
 3  INX  DCX
 5  DCR
 7  RLC  RAC  DAA  STC  RRC  RAR  CMA  CMC
```

The 1xy page is entirely devoted to the MOV instruction, except for 166, which would logically be MOV M, M but is instead HLT.

```
 1xy   x (dest)
 B  D  H  M  C  E  L  A
 0  2  4  6  1  3  5  7
 y
 (src)  B  0  MOV
 D  2
 H  4
 M  6  HLT
 C  1
 E  3
 L  5
 A  7
```

The 2xy page consists of single-operand instructions that implicitly act on the accumulator A, but unlike the oxy page, the operand is in the final octal digit, not the middle one. If laid out consistently with the other pages, this makes the instructions columns:

```
2xy  x
0  2  4  6  1  3  5  7
y
(src)  B  0  ADD  SUB  ANA  ORA  ADC  SBB  XRA  CMP

D  2
H  4
M  6
C  1
E  3
L  5
A  7
```

Finally, the 3xy page contains all the control-flow and stack operations, plus some miscellaneous operations; some operate on register pairs, some on registers, some on neither. Three of these operations (Rcc, Jcc, Ccc) contain a 3-bit condition-code operand instead of a register operand, and the RST instruction contains an interrupt vector number.

```
3xy  x
B  D  H  M  C  E  L  A
BC  DE  HL  SP  BC  DE  HL  SP
NZ  NC  PO  P  Z  C  PE  M
0  2  4  6  1  3  5  7
y  0  Rcc
2  Jcc
4  Ccc
6  ADI  SUI  ANI  ORI  ACI  SBI  XRI  CPI
1  POP  POP
PSW  RET  -  PCHL  SPHL
3  JMP  OUT  XTHL  DI  -  IN  XCHG  EI
5  PUSH  PUSH
PSW  CALL  -
7  RST
```

Why?

The 8080 is interesting to me not just for nostalgic reasons (many of my first computers in the 1980s were Z80-based) but because it's nearly the smallest existing computer demonstrably capable of self-hosted software development with an assembler and running a usable user interface, at least if you have a character generator or printer connected to it. The PDP-8 and LGP-30 are simpler, but John Cowan tells me most PDP-8 development was actually done on a PDP-10 and cross-compiled, as with modern embedded microcontrollers, and the LGP-30 was normally programmed in machine code, with the programmer doing the "assembly" beforehand with pencil and paper. By contrast, although much significant software for the 8080 was written on a PDP-10 (notably Microsoft BASIC), much of it was written under CP/M on the 8080 itself.

Wirth's RISC for Oberon and James Bowman's J1A Forth CPU

are other reasonable candidates, and both are fairly inspired designs with much simpler instruction sets than the 8080, but I think both require more transistors than the 8080, and the available software for them is somewhat lacking.

The GreenArrays F18A CPU design requires, I think, fewer transistors than the 8080 (certainly the MuP21 did) and has a simpler and much more powerful instruction set, but almost no software exists for it, and in particular there is no published self-hosted development environment, as far as I know. (The 18-bit address space is nine times the size of the 8080's, but the chips made so far have only 64 words of RAM per CPU.)

By contrast, the 8080 has existing self-hosted assemblers as well as compilers for Turbo Pascal, Fortran, small-c, Tiny-C, and BASIC; computer algebra systems; display text editors; CP/M, which includes the assembler, a rudimentary filesystem, file management utilities, a REPL, and a debugger; and at least two free-software operating systems — Drew DeVault's KnightOS and David Given's CP/Mish. Yet you can fit the whole 8080 instruction set on a sheet of paper, and its full documentation is a 15-page chapter in the Intel manual.

This is an inspiring example of what is possible, even if the 8080 instruction set itself is kind of clumsy and lame, with the benefit of 40 years of hindsight. Its very imperfection is encouraging — it shows that even deeply flawed hacks can have enduring value and even achieve greatness.

BDS C

I *just learned* that there's a public-domain full C compiler for CP/M written in assembly; BD Software C (aka "BDS C") was dedicated to the public domain in 2002. According to p. 264 of Byte August 1983, this was one of the fastest C compilers available for CP/M, and supported a fairly complete version of the C language (well, for the platform.)

> I, Leor Zolman, hereby release all rights to BDS C (all binary and source code modules, including compiler, linker, library sources, utilities, and all documentation) into the Public Domain. Anyone is free to download, use, copy, modify, sell, fold, spindle or mutilate any part of this package forever more. If, however, anyone ever translates it to BASIC, FORTRAN or C#, please don't tell me.
>
> Leor Zolman
> 9/20/2002

From my point of view, at least, the availability of this software catapults the 8080 architecture from being a vaguely plausible but implausibly inconvenient architecture to program for, to being a simple architecture with a viable self-hosting development toolchain.

Topics

- Programming (p. 3658) (286 notes)
- Independence (p. 3520) (63 notes)
- Small is beautiful (p. 3714) (40 notes)
- Instruction sets (p. 3526) (40 notes)
- Assembly language (p. 3328) (25 notes)

Notes on SIP VoIP in 2019

Kragen Javier Sitaker, 2019-06-07 (updated 2019-06-28) (8 minutes)

Looking at getting a SIP provider for international calls. I am somewhat impeded by not knowing anything about SIP and RTP, although I have a SIP client on this phone.

An overview, as I understand it: SIP is the standard call setup protocol for VoIP; RTP is the protocol used for the actual data. The company that hooks up VoIP calls to the PSTN (public switched telephone network) is called a "SIP provider" or "ITSP" ("internet telephony service provider"). Much of the SIP business currently comes from companies who want to hook up their PBXes to the PSTN. A "DID" ("Direct Inward Dial [number]") is a non-toll-free incoming phone number. An "ATA" is a SIP-speaking device you can plug an analog phone into.

SIP phones can speak directly to each other over the internet as well.

Notes on particular providers

voip.ms

Voip.ms seems to be one of the default choices, and I think it would cost US$1.50 per month for E911 service (though maybe that's optional?), US$0.85 to US$1.25 per month for an incoming phone number (plus US$0.40 to set it up), and about US$0.01 per minute both inbound and outbound, including to "toll-free" numbers. You need a minimum of US$15 prepaid to do incoming or outgoing calls, but you can sign up for an account and use their voice call test thing without paying. They offer a broad spectrum of features.

SIPStation

SIPStation offers service at US$24.99 a month for a monthly plan, plus (?) US$1 per month plus US$0.024 per minute.

Vitelity

Vitelity apparently no longer allows new customers to sign up for VoIP since merging with Voyant in February 2018; they're 100% focused on the VoIP VAR market now.

The FreeSWITCH Wiki page has a lot of stuff about working around problems, which I take to mean that they were very popular, not that they had a lot of problems.

Google Voice

Google Voice apparently does support SIP but will cut you off if they detect you're calling 1-800 numbers from a non-US IP address. Added to that is the constant menace that they may link your phone number to your Google account, so a problem on one means losing the other. Not an option I'd ever consider.

Flowroute

Flowroute charges US$1 for setup and US$1.25 per month, plus the Federal Universal Service Fund charge (of some unknown amount?) plus US$0.012 per minute inbound and US$0.0098 per minute

outbound. They default to business accounts but also offer personal accounts. They have technical details.

The FreeSWITCH Wiki page just offers some XML from 2013.

Vonage

Vonage mostly doesn't offer SIP, though the FreeSWITCH Wiki page explains that secretly they do through resellers. Not an option.

Bandwidth.com

Bandwidth.com focuses on "the biggest brands" and has a free trial.

The FreeSWITCH Wiki page explains how to make it work as of 2011.

Notes on particular pages

The above is partly collated from the below.

Freeswitch.org has a list that is far too long to consider. But the page for voip.ms, aka Swiftvox (last updated 2014) shares chunks of XML to use to configure, I guess, FreeSWITCH, to work properly with voip.ms. Also implies voip.ms defaults to being configured for "ata device, ip phone or soft phone", which is what I have, of course.

A year ago, jhalstead was looking at "FlowRoute, Vonage, voip.ms, nextiva, Sipstation". Someone else in the thread suggested Telnyx; another person seconded the FlowRoute recommendation, and cyberchaplain said, "I've personally used Bandwidth, Level3, Flowroute, callwithus and voip.ms and can't complain about any of them really." Others mentioned Vitelity, Touchtone Communications, and Spectrum/TWC, and a Nextiva employee tried to persuade them to switch to hosted. This suggests that, at the time, FlowRoute and voip.ms were the popular options. But this thread is mostly oriented toward companies with PBXes.

An ad for Vitelity used SIPStation as their comparison competitor. The ad seems to be directed at small businesses like a "hardware store or restaurant" who want to get phone service for on the order of US$3.99 a month (vs. US$100 with SIPStation), but also those who want to "scale up to hundreds of trunks". It also mentions SIPStation's ability to spoof caller ID as a benefit, allowing you to use SIPStation for outgoing calls while receiving your incoming calls on a different number. The main point of the post seems to be that it's good to pay by the minute rather than by the phone line, and an addendum notes that eventually SIPStation switched to charging by the minute too. Another addendum notes that Vitelity is now Voyant Communications and "has halted new registrations for the time being".

An ad for SIPStation from 2017 touts the benefits of their pricing; its example company is paying US$499 a month for two locations with 10 lines, but by using SIPStation's "trunk groups" they can switch to just 15 lines an pay only US$374.25 a month. Pretty cheap if you're a company with multiple offices and dozens of employees, I guess.

voip.ms publishes their pricing information as follows:

Outgoing Calls, USA Rates

- Premium Route: $0.0100 (1¢) per minute
- Toll-Free Numbers Value Route: Free

• Toll-Free Numbers Premium Route: $0.0106 (1.06¢) per minute
• e411: $0.99 per call. Must be activated by customer in settings
Incoming calls, USA / Canada DID Per Minute Pricing

• Monthly Fee:
USA: $0.85 to $1.25
Canada: $0.85 to $1.70
• Per minute inbound:
$0.009 to $0.0125
• One time setup Fee: $0.40
• Billing Increment: 6 seconds
• Channels: 25
• Intended Use: Any
Extras Features

• E911/911 $1.50 per month

Seven months ago Blade_Fox moved from Vonage to voip.ms, though they didn't explain why, and wanted help getting SMS working. Mizzlezz, in the comments, is using Bandwidth.com.

Last year johndrwhosmith was looking for recommendations for a "hosted/cloud PBX", saying they were thinking of Nextiva, and slayter commented:

do not use voip.ms unless you are experienced in VOIP or intending for residential usage. They have failed to register as a telecom in Canada and could even be shut down tomorrow with nothing to say about it.

Maybe I'm being too blasé but I don't see that as a big problem.

Four years ago `pseud_o_nym` was going to "port" their phone number from Comcast to ring.to, and RocketTech99 recommended looking at voip.ms (though it was "not [their] favorite provider", they "have a very low entrance cost".)

Two years ago, jrdbm, a reseller of Bandwidth.com's VoIP service, asked for help with some porting problems, and got recommended FlowRoute.

This thread from last year explains when Google Voice cuts you off. People are also recommending voip.ms and Callcentric in there, but I think the Callcentric recommendation is from someone who works at Callcentric.

Other pages to read: a b c d e.

Topics

Counting the number of spaces to the left in parallel

Kragen Javier Sitaker, 2016-10-11 (5 minutes)

Suppose we want to transform a sequence like this:

```
.#....#.#..#....#.....##.#..#....#
```

into another sequence as follows:

```
.#....#.#..#....#.....##.#..#....#
10123401012012340123450010120123 40
```

which is to say, for each position, we want to compute the number of places to the left that you have to go to find a #. (This problem comes from a problem I'm thinking about with regard to coregistration of potentially translated sparse bitmaps with their pointwise products.)

This is super easy in an imperative language:

```
for item in seq:
    if item == '#':
        count = 0
    yield count
    count += 1
```

However, that approach is inherently serial. What does it look like to reformulate this in a way that we can compute with a prefix sum, so that we can automatically parallelize it?

Parallel prefix sum algorithms require the addition operation over which they're computing the sum to be associative, and no associativity is evident in the above at first glance.

The fully general procedure for this transformation on such loops over input is to formulate each iterations of the loop as a function from previous state to next state, the function in question being determined by the input on that iteration of the loop, and then to apply the prefix-sum algorithm to these functions with the "addition" operator being functional composition.

In this case, the loop state is merely `count`, and there are two possible functions:

- on #, `count` is set to 1
- on ., `count` is incremented

Thinking of these as functions from a previous to a next state, they are λc.1 and λc.c+1. These do not form a set that is closed under composition; under composition you have the set {n∈ℤ | λc.n, λc.c+n}, more or less. (λc.0 and λc.c+0 actually don't occur.) The composition rules are then the following:

- (λc.n) ∘ (λc.m) = λc.n
- (λc.n) ∘ (λc.c+m) = λc.n
- (λc.c+n) ∘ (λc.m) = λc.m+n

• (λc.c+n) ∘ (λc.c+m) = λc.c+(m+n)

(These are just the two functions + and K, of SKI-combinator fame.)

You can write this in OCaml with an appropriate data type as follows:

```
let compose = function K a -> (fun _ -> K a)
                     | Plus a -> function K b -> K (a + b)
                                        | Plus b -> Plus (a + b)
```

I feel like this algebra is some kind of semigroup that I should recognize. It isn't commutative, it has no inverses, and although λc.c+0 would be an identity element, that doesn't actually occur in my problem. But it is associative, which is all prefix-sum needs.

From the sequence of `count` values, you can reconstruct the desired original inputs: they're one less than the `count` values.

Given a representation of this set of functions, say =n for λc.n and +n for λc.c+n, you can compute the function in parallel in logarithmic time as follows:

```
 . # . . . . # . # . . # . . . . # . . . . . # # . # . . # . . . . #
+1=1+1+1+1+1=1+1=1+1+1=1+1+1+1+1=1+1+1+1+1+1=1=1+1=1+1+1=1+1+1+1+1=1
  =1  +2  +2  =2  =2  =1  +2  +2  =2  +2  +2  =1  =1  +2  =2  +2  =1
      =3      =2      =1      +4      =4      =1      =3      =4  =1
              =2              =5              =1              =4  =1
                              =5                              =4  =1
                                                              =4  =1
                                                                  =1
                                                              =4  =1
                              =5                              =4  =1
              =2              =5              =1              =4  =1
      =3      =2      =1      =5      =4      =1      =3      =4  =1
  =1  =3  =5  =2  =2  =1  =3  =5  =2  =4  =6  =1  =1  =3  =2  =4  =1
=2=1=2=3=4=5=1=2=1=2=3=1=2=3=4=5=1=2=3=4=5=6=1=1=2=1=2=3=1=2=3=4=5=1
 1 0 1 2 3 4 0 1 0 1 2 0 1 2 3 4 0 1 2 3 4 5 0 0 1 0 1 2 0 1 2 3 4 0
```

In theory, this only takes 14 computational steps on each of 34 processors, rather than the 34 needed to calculate the same thing serially. In practice, it is going to be difficult to find hardware that can realize a 2x speedup on that problem. But for 4096 positions instead of 34, it should only take 26 steps rather than 4096, so you do eventually get a speedup if you have enough hardware, even if your chunk size is larger.

This was only possible for this algorithm because the state kept from one loop iteration to the next was relatively compact. If the state grows proportional to the number of iterations (or worse), you will never get a speedup.

What kind of Sufficiently Smart Compiler would it take to analyze the serial program and parallelize it in this way? Because here's the code you get to write in OCaml for the parallel version:

```
type counter = K of int | Plus of int
let compose = function K a -> (fun _ -> K a)
                     | Plus a -> function K b -> K (a + b)
                                        | Plus b -> Plus (a + b)
```

```
and init = function '#' -> K 1 | _ -> Plus 1
and final = (-) 1
in prefixsum init compose final
```

which I feel is not just longer but dramatically less clear than the serial version

```
for item in seq:
    if item == '#':
        count = 0
    yield count
    count += 1
```

even though the latter is in some sense written at a lower level of abstraction.

Topics

Piezoelectric engraving

Kragen Javier Sitaker, 2017-07-19 (4 minutes)

You should be able to engrave permanent images on metal or glass using a ceramic or metal stylus on a flexing piezoelectric arm with no joints, somewhat similar to the needle of an STM or AFM. Aluminum oxide (sapphire) is probably the easiest ceramic to use, although it provides no electrically-conductive feedback about contact; metal (hardened tool steel, say), or conductive ceramics like tungsten carbide, would remedy that shortcoming, allowing sensitive calibration of engraving depth.

The total movement range of such a setup is likely very small, in the neighborhood of a millimeter, but it can potentially micro-forge the surface of the metal with sub-nanometer resolution, enabling the direct engraving of holograms. I'm not sure what it will do on glass, but I think it's possible to get it to make scratches rather than just break the glass.

To take a random example, the American Piezo catalog lists a "PSt 150/5x5/7" osi-type piezoelectric stack actuator of 5 mm × 5 mm, 9 mm long, with a maximum stroke of "13/9" µm (not sure what's up with the two numbers), 800 nF capacitance, resonant frequency of 100 kHz, 120 N/µm of stiffness, 1600 N of blocking force, with a maximum load of 2000 N, operating from -30 V to +150 V. If you hook up three of these things in parallel to a chunk of metal with a grain of aluminum oxide on its tip, they could jam that grain 9 µm into a bit of aluminum, titanium, glass, or even steel, pull it back out, move it into a different position, jam it back in, and repeat, at 25 kHz (two octaves below resonant) without any difficulty.

If the chunk of metal has a 10:1 aspect ratio, for example if the piezo actuators are attached to it 20 mm apart and it's 200 mm long, then you can wave that little grain of sapphire back and forth by 90 µm, almost the width of a hair. Some kind of flexure-lever arrangement to amplify this by another factor of 10 might be a good idea.

You could presumably engrave data on a little spot at about 50 kbps in this way. (They also offer flat chip actuators with a shorter stroke but much higher resonant frequency, like 500 kHz.) But then you would probably need some kind of repeatable positioning apparatus to engrave over a wider area.

The pressure generated is enormous. Supposing that the tip is up to 25 µm in diameter, 1600 N spread over 490 square microns is 3.3 terapascals. Different metals have yield stresses in the range of 90 MPa (copper) to 2.5 GPa (piano wire), with around 500 MPa being normal; this is 6600 times lower. (Actually annealed aluminum is down around 15–20 MPa.) Sapphire's ultimate tensile strength is only around 1.9 GPa, and diamond only 2.8.

(Oops, actually the force and pressure from the actuator is potentially 3× that if you have three parallel actuators.)

So you really could find a way to gear these actuators up by a factor of 1000 or so, it would give you a stroke of 9 mm with potentially nanometer precision, and still plenty of force to engrave the surface. (I suppose this is why normal STMs use bending actuators.) If you

could still manage 50 kbps, although this seems more dubious, you could engrave a 9 mm square area at 500-nm resolution at 0.0125 mm² per second, filling the whole area in about two hours. This is a good timescale.

Unfortunately, at least the Physikinstrumente devices that gear up piezoelectric actuators in such a fashion have much lower resonant frequencies, like 150 Hz.

As an alternative, maybe you could use electromagnetics, which can also reach up into the MHz range.

Topics

Review notes for Chris Anderson's "Makers"

Kragen Javier Sitaker, 2013-05-17 (5 minutes)

I didn't know he was a programmer.

His central thesis seems to be that digital designs, because they're files and therefore copyable, can be shared online --- and therefore, I suppose, will be; and that this will result in empowering individual inventors: "Would-be entrepreneurs and inventors are no longer at the mercy of large companies to manufacture their ideas." (p.29)

I'm probably not the best critical audience for this thesis, since it's a thesis I've been promoting myself since at least 1999. So I'm already prepared to believe it.

How could this decentralization and empowerment fail? One way would be in the way that Facebook and Apple have tournamentized software development, creating an incentive structure in which the aggregate value created by "app" vendors is dramatically less than the amount spent creating the apps, and only the very best apps are profitable. Anderson says, (p.61) "Today we are seeing a return to a new sort of cottage industry," while acknowledging that the original cottage industries "were always at the mercy of the industrialists." (p.60) He's optimistic that since the invention and not merely the manufacturing are distributed (indeed, in the model he promotes, it's the reverse: the manufacturing is centralized in service bureaus) that the new cottage industrialists will have the power to set their own prices and generate high profits.

Anderson's usual carelessness about the truth is in evidence; on p.44, for example, he talks of "spinning multiple threads of cotton from flax", a feat worthy of Rumpelstiltskin; on p.47, he repeats the gross distortion that the privatization of common grazing grounds was an "Improved farming method" that "avoided the 'tragedy of the commons' problem", which has been amply rebutted elsewhere; he claims, "The original Moore's Law, named after Intel researcher Gordon Moore, described the twenty-four-month doubling of processing power per dollar that has characterized the computer industry since the 1970s," (p.84) which contains at least four factual errors and possibly five; "a mathematical equation of how to make it... [i]s actually the way CAD programs work" (p.85); he consistently misapplies the term "scale-free network" (p.136) in a way that suggests he has no idea what it means: a network where there are a lot of nodes with many more connections than average, such that the fraction of nodes with more than k connections is proportional to k to some power around -2 or -3;

He refers back to his first book, *The Long Tail*, to explain his thesis: in the absence of shelf-space restrictions, many more books became profitable to sell, and too with music, software, and anything else that you can download. He cites a "shift in culture toward niche goods." But the strong version of the thesis in *The Long Tail* was that the majority of Amazon's sales came from books you couldn't find in your local bookstore; that is, that niche goods were not only more popular, but more popular than the "mainstream" goods that

shelf-space restrictions used to confine us to. That turned out to be wrong, because of bad estimates of Amazon's sales numbers. You can, of course, make it true by choosing your arbitrary dividing line between "mainstream" and "niche" to be above the median, but it turns out that if you do that, then there's an awful lot of "niche" books in bricks-and-mortar bookstores too. ("Even Wal-Mart now sells more than a hundred kinds of mustard," p.78.)

His enthusiasm for his new model of production ("the perfect combination", p.80; "anyone can get access to manufacturing and distribution", p.89; "this is revolutionary", p.97; "MakerBot is...a revolutionary act...a political statement", p.104;) turns me off and makes me doubt his objectivity.

One thing that I thought was interesting and new is the distinction he draws between Toffler's "mass customization" (Dell computers, Nike ID shoes, monogrammed iPads, etc.) and his new prediction of "a mass market for niche products" (p.88).

There are numerous useful tidbits of information; a square foot of laser-cut plywood or plastic might cost US$15 (p.107); the names of popular service bureaus like Ponoko and Pololu (p.107); the names of popular 3-D design web sites like Thingiverse (XXX); "entrepreneurs...price their product at at least 2.3 times its cost" (p.117); the reward hierarchy 3D Robotics uses to entice contributions from DIYDrones members (p.121); unfortunately, I worry that Anderson's lack of reliability on things I already know means that he's not a reliable source for information in things I don't.

Topics

- Pricing (p. 3646) (89 notes)
- Manufacturing (p. 3558) (50 notes)
- Digital fabrication (p. 3411) (42 notes)
- Politics (p. 3639) (39 notes)
- Economics (p. 3424) (33 notes)
- 3-D printing (p. 3301) (23 notes)
- The future (p. 3746) (20 notes)
- Decentralization (p. 3404) (13 notes)
- Book reviews (p. 3347) (5 notes)
- Charlatans

Plastic cutters

Kragen Javier Sitaker, 2019-04-20 (5 minutes)

My flesh has a Young's modulus much less than that of steel, but by holding a file or saw in my hand I can cut steel. The force on my flesh is the same as the force on the steel — except for a nearly irrelevant term of tool weight — but it is distributed over a larger contact area.

Similarly, I think we can produce plastic machinery that can cut harder materials, such as steel, by holding small "tools" or "teeth" of ceramic or hardened steel. The small tools might not themselves be of particularly ideal geometry — perhaps they are the results of crushing a zirconia knife or a silicon wafer with a hammer, for example — if the plastic machinery and control system is adapted to the particular geometry they have, if it uses sufficiently low cutting forces, and if the geometry has sufficiently sharp points (though points that are too sharp will slow down cutting).

The first part of the idea is simply that the rigidity of the connection between the tool and its holder should be at least comparable to, and ideally exceed, the rigidity of the contact between the tool and its work — each of which is a Young's modulus multiplied by a contact area. So if, for example, the tool holder is nylon or polystyrene, with their Young's modulus around 3 GPa, while the workpiece is a titanium alloy with a Young's modulus around 110 GPa (steel is around 200 GPa), the contact area on the workpiece needs to be more than 40 times smaller than the contact area on the tool holder — ideally more like 400.

That's actually the less important criterion, though; it's actually possible to cut with a tool held by a tool holder that deflects more than the material does, although it's going to need complicated control algorithms to get decent precision that way. The more important criterion is that the tool holder's yield strength needs to be greater than the ultimate strength of the cut surface — the first being the yield stress multiplied by the contact area, the second being the ultimate stress multiplied by the tearing area. Titanium's ultimate strength is 900 MPa with 6% aluminum and 4% vanadium; A36 steel's is only 400 MPa. Meanwhile 6–6 nylon's yield stress is around 45 MPa and polypropylene's is around 12–43 MPa.

So, even though the strength criterion is more important than the rigidity criterion, you can meet the strength criterion easily if you can meet the rigidity criterion, because common materials vary much more in elasticity than they do in strength.

For cutting softer materials like wood, bone, or fingernails, broken glass teeth would work fine.

Suppose that the tooth is silicon carbide, with its Young's modulus of around 450 GPa. (I don't know what its ultimate strength is, but I don't think it will often come into play here, since I assume everything else around it will break first under the desired low-shock conditions.) Suppose you're pushing it into A36 steel, with its 200 GPa modulus and 400 MPa ultimate strength, and that the splitting part of the metal is comparable in size to the contact area, which is, say, about a 1 mm circle, 0.79 mm². To get the steel to cut,

you need a force of 314.159 newtons, which will also have compressed the steel immediately around the cut by 0.2% and the tooth by 0.1%. If your holder is 6–6 nylon, then in order to not yield at 45 MPa, it needs a surface area of 7 mm² pressing on the root of the tooth, and that plastic will squish by 1.5%; to squish by less than 0.2% it would need to be 52 mm², an 8-millimeter-diameter circle — a rather large chunk of carborundum! The parts of the workpiece and the tooth holder ("gum"?) further away from the tooth will be resisting the same force over a larger cross-sectional area, unless the workpiece is very small, so the deformation will be less.

These rather demanding dimensions for the tooth and tooth holder can be improved by using an intermediate material between the plastic and the ceramic, such as brass, aluminum, or steel, into which the tooth will be set in order to be grasped by the plastic. In the above example, the 8-mm-diameter circle of plastic could be grasping a chunk of aluminum, brass, or steel, which in its turn grasps the tooth itself, and is perhaps brazed or soldered to it.

Topics

The fable of the specialized fox

Kragen Javier Sitaker, 2019-08-17 (1 minute)

A fox was hunting and had just caught a rabbit, and just after he began eating his dinner, a wolf appeared behind him.

"Hello," said the wolf, "I wanted to talk to you about economics. Have you heard how specialization improves productivity and brings prosperity to everyone?"

"No," said the fox.

"If we each specialize in the area where we have the most comparative advantage, we can get better at it and be more productive. Shall we try it?" said the wolf.

"Okay," said the fox.

"I see you're very good at hunting, so I think it would be great if you specialized in hunting, and I'll just specialize in allocating the resources that result from your work," said the wolf. And so he ate the rest of the rabbit the fox had caught.

"Well," said the fox, "I'm not sure I like this kind of prosperity. But I guess I can have a couple of specializations, right, as long as they don't conflict with yours?"

The wolf assented.

"I think I'll specialize in urination."

The next week, the wolf died of an exploded bladder.

Topics

How can we usefully cache screen images for incrementalization?

Kragen Javier Sitaker, 2013-05-17 (18 minutes)

Earlier: http://lists.canonical.org/pipermail/kragen-tol/2012-July/000963.html

Okay, so, if you write your program purely in terms of pure functions, you can memoize those functions, perhaps applying a global caching policy to their results to optimize throughput or limit expected latency. I've written about this a bit before.

You can do a bit better than pure functions: you can run arbitrary code with read-only access to the entire program state, as long as its access to that state is mediated by a sort of software-transactional-memory layer that records everything it reads. Then you can automatically invalidate the cached result whenever any of the state variables that fed into it are changed or invalidated.

I was thinking about trying to apply this paradigm to a smallish program this weekend, one which redraws the screen frequently. I'd like it to just redraw the relevant part of the screen. At the moment, I'm sending text and escape sequences to a tty, but more or less the same principles apply if you're updating a framebugger.

The program displays a spreadsheet-like table with editable text in the cells. Sometimes it updates the layout.

But most keystroke events result in a single letter being added to the screen, overwriting a blank space, or a single letter being replaced with a blank space; many others result in highlighting one cell of the table and unhighlighting another.

There are perhaps 100 characters per line and perhaps 20 lines on the screen, for about 2000 characters in total. The simplest way to make the program work is to redraw the whole screen after every keystroke, which does run acceptably fast on modern hardware. But this does something like 2000 times as much work as necessary: 22 Moore's Law years, taking us back to about 1980. It would be nice to find an approach to only redraw the changed part of the screen, not even the entire line.

Now, if you're only looking at the bytes coming out of the program, you can achieve this by redrawing an in-memory screen image, comparing it to the previous screen image, encoding the delta in bytes, and sending that to the tty. But that doesn't really touch the orders-of-magnitude issue.

Variable per character

You could have a couple of transactional-memory variables for each character position on the screen, and *monitor* a function for each of those characters. To update the screen, you'd see which characters had changed, and iterate over them sending them to the screen. This has presumably order-of-magnitude overheads, but it ought to be scalable.

What does the function for a character look like? Something like

```
char_to_show(x, y) = (highlight if is_current(cell_obj) else normal)(char)
where:
    cell_obj, label_index = table_cell_covering(x, y)
    label = label_of(cell_obj)
    char = label[label_index] if label_index < length(label) else ' '
```

Each of these functions could reasonably be cached, but `table_cell_covering` is kind of a bear, because it depends on the current layout, which depends ultimately on the contents of every cell that isn't to the right:

```
table_cell_covering(x, y) = cells[col][y], dx
where:
    col = find_col(x, 0)
    dx = x - col_start(col)

find_col(x, col) = col if found_col else find_col(x, col+1)
where:
    found_col = (col == n_cols - 1 or col_start(col+1) > x)

col_start(col) = 0 if col == 0 else col_start(col-1) + col_width(col-1)
col_width(col) = col_width(col, n_rows)
col_width(col, row) = 0 if row == 0 else max(col_width(col, row-1), w)
where:
    w = length(label_of(cells[col][row-1]))
```

Now, `col_start` can clearly be cached usefully: it has a small number of distinct argument values, and it depends ultimately only on the lengths of the labels of the columns of `cells` to the left of the requested position. You could alternatively cache `col_width`.

But that means that if you change the label of, say, the bottom left cell, the `col_start` values of every cell in the matrix needs to be revalidated --- just in case you changed the width of the first column. Usually you won't have, but if you did, the whole screen might need to be updated. You'd have to redo the `col_start` computation to see that it hadn't changed, and then, if you'd propagated an "invalidated" notification downstream to the `find_col`, `table_cell_covering`, and `char_to_show` computations, follow it up with a "revalidated" notification.

But that means you're delivering thousands of "revalidated" notifications, which is basically what you were trying to avoid in the first place.

How could you avoid this?

• Eagerness: instead of merely invalidating `col_width(col, row)` when you change `cells[col][row-1]`, necessarily propagating the invalidation to `col_width(col)`, `col_start(col+1)`, and so on, you could simply re-evaluate `col_width(col, row)` immediately, probably coming up with the same value again and avoiding pushing the change further downstream.

• An idea that doesn't work: Deep validation checking: instead of propagating the invalidation flag to thousands of character positions, so that the operation of cache validation for `char_to_show(x, y)` is O(1), you could make the is-valid check on the character position dig into

the dependencies and dependencies of dependencies of the character position, so that whenever you run the is-valid check on any of the changed character positions, it notices whether a relevant cached `col_start` value has changed. This doesn't work because it presupposes that you're iterating over all the thousands of character positions in order to figure out that you need to update only one character position on the screen, which is what I was trying to avoid in the first place.

• Aggregation: Instead of making each of the 2000 or so character positions a tracked variable, divide the screen into about sqrt(2000) regions: say, half a line each, which should give you about 40 variables. Then your initial invalidation notification ends up affecting, say, half the screen, but that's only about 20 invalidation notifications; and then you can iterate over those 20 chunks of visible text and figure out that 19 of them are still up-to-date, and the other one has changed, and redisplay its 50 characters. This means that you're doing about 100 things (including redisplaying 50 characters) instead of 2000, but that's still a long way from 1.

Side-effecting transactions: I don't think they'll work

In transactional memories used for concurrency control, your transaction code can choose any arbitrary variable in the STM to write to. Your writes are buffered, and if your transaction successfully commits, they become visible to later transactions, and if there are currently running concurrent transactions that have read the modified state variables, those other transactions are aborted.

By contrast, I've been proposing a purely-functional computational model here. The problem is that it's fairly absurd to think of going through a computation like this:

```
char_to_show(x, y) = (highlight if is_current(cell_obj) else normal)(char)
where:
    cell_obj, label_index = table_cell_covering(x, y)
    label = label_of(cell_obj)
    char = label[label_index] if label_index < length(label) else ' '

table_cell_covering(x, y) = cells[col][y], dx
where:
    col = find_col(x, 0)
    dx = x - col_start(col)

find_col(x, col) = col if found_col else find_col(x, col+1)
where:
    found_col = (col == n_cols - 1 or col_start(col+1) > x)
```

in order to figure out what to draw in a *single character position in a terminal*, even if `is_current`, `label_of`, and `col_start` are all fully cached. (If you extrapolate this example to pixels, the problem is even worse.)

Consider, by contrast, this imperative push-based code:

```
show_cell(row, col) =
    cell = cells[col][row]
```

```
label = label_of(cell)
x = col_start[col]
width = col_width[col]
start = row * screen_width + x
len = length(label)
memcpy(&screenbuf[start], charptr(label), len)
memset(&screenbuf[start + len], ' ', width - len)
color = (highlight if is_current(cell) else normal)
memset(&attrbuf[start], color, width)
```

This is doing about the same amount of work as the pull-based code above, but it's displaying the entire cell, not just a single character!

The basic problem here is that each cell of the table only affects a small number of character positions on the display. It's reminiscent of the problem with the functional stream-based "unfaithful Sieve of Eratosthenes" that Melissa O'Neill identified in her paper, *The Genuine Sieve of Eratosthenes*; where she says:

> In Eratosthenes's algorithm, we start crossing off multiples of 17 at 289 (i.e., 17× 17) and cross off the multiples 289, 306, 323, . . . , 510, 527, making fifteen crossings off in total. . . .
>
> After finding that 17 is prime, the unfaithful sieve will check all the numbers not divisible by 2, 3, 5, 7, 11 or 13 for divisibility by 17. It will perform this test on a total of ninety-nine numbers (19, 23, 29, 31, . . . , 523, 527).

Analogously, to figure out what to draw at the character position (60, 4), my side-effect-free table-drawing code must ask whether column 60 is in table column 0, then whether it's in table column 1, and so on, until it finds the correct column; and then it must do the same computation again for column 61, and column 62, and so on. By contrast, the imperative algorithm can do the analogous computation with a simple array lookup, and then it can handle the entire string contents of the label with a simple `memcpy`, which processes several bytes per cycle.

So it would be nice to make this work. But I don't think it will, because of the invalidation problem inherent in imperative overwrites.

STM dependency tracking works because you don't need to know what other variables a transaction could conceivably have read or written, had it found different values in the variables it did read; if need be, you can abort and restart the transaction with the new variable values, and record its new behavior.

In this case, though, we're interested not only in the variables that the transaction *did* write to, but the variables it *could have* written to. In this case, for example, we'd like to know what screen positions our cell could have written to if the layout had been different: either the cell itself was wider, overwriting positions to its right, or the cells to its left were narrower, so it overwrote positions to its left.

This is an insoluble problem as long as the code running inside the transaction is written in a Turing-complete language. You could maybe do it if it's expressed in something more restrictive and

"declarative", but this is enough to make me give up on this line of inquiry.

Composing the screen image from cacheable pieces

The problem with the above approaches is that you have lots of branching out in the dataflow graph: ultimately, the contents of every cell at least potentially affects the contents of every character position on the screen that isn't to its left.

What if we go the other way instead? Instead of branching out, we branch in: many table cells get turned into cacheable chunks of screen, which are composed into a screen image.

At first glance, this seems to be a non-solution: what do you do with the screen image? It's 2000 characters. Do you want to iterate over all of them?

Consider how you'd implement the `label_of()` function described above in a language like C, without thinking about caching, in particular for a cell whose label is really an integer, not simply a string; or maybe it's in a format like "t >> 6", where the 6 is an integer that can vary over time. You'd like to be able to cache the results.

You could dynamically allocate a string buffer, serialize the number into it (expanding it if necessary), and return the new string, attempting to communicate to the caller that they were responsible for deallocating it when they were done with it. (Maybe if your program was too efficient, you'd use reference counting.) But that's sort of wasteful; in the end, the string contents are going to get copied into some kind of screen image or something.

A disadvantage of the above method is that the cell can produce an arbitrarily long label, which means you need dynamic allocation, at least in principle. This is sort of silly if you're generating a label to fill a fixed-size space on the screen. You could use the interface provided by the `read` system call: when you call `read`, beyond telling it which file you want to read, you pass it a buffer pointer and a maximum length, and it writes your result into that buffer. Then you could pass the render-cell functions pointers to the relevant parts of your screen buffer.

For more flexibility, you could provide an "output" callback, which can be called, maybe more than once, to add characters to the label. This approach subsumes the previous two, since your output callback could simply add to a string buffer, but it could instead update a screen image. For caching purposes, we can consider the image produced by calls to the output callback to be the "return value" of your cell rendering function.

So suppose this is the interface we use for the things that generate parts of the screen contents: we pass in a "window" object to draw into, and we consider its dimensions part of the arguments for the sake of caching; it has a "subwindow" method that can be used to generate a smaller window to pass to subfunctions. Then we can do things this way:

```
image_of_table(win) = image_of_columns(win, n_cols)
image_of_columns(win, col) =
    if col > 0:
```

```
        start = col_start(col)
        image_of_columns(win.subwindow(width=start, col-1), col-1)
        image_of_column(win.subwindow(left=start, width=col_width(col)), col-1)
image_of_column(win, col) =
    if win.height > 0:
        image_of_column(win.subwindow(height=win.height-1), col)
        image_of_cell(win.subwindow(top=win.height-1), col, win.height-1)
image_of_cell(win, col, row) =
    color = highlight if is_current(cell) else normal
    win.show(label_of(cells[col][row]), color)
col_start(col) = 0 if col == 0 else col_start(col-1) + col_width(col-1)
col_width(col) = col_width(col, n_rows)
col_width(col, row) = 0 if row == 0 else max(col_width(col, row-1), w)
where:
    w = length(label_of(cells[col][row-1]))
```

(The imperative syntax here is fairly grating. It would probably be clearer to write these as expressions of some kind.)

Here, if you update the label of the upper-left-hand cell, it will initially invalidate `col_width` of the first column, the image of that cell, the image of its column, and the images of every set of columns; but it will not immediately invalidate the images of each other column, nor the other cells in the same column. If the newly-recomputed `col_width` is the same, the images of the other cells in the column will also not need to be recomputed; and so the only call to `win.show` will be for that single cell. The other cells and columns will be composited into the screen image in the same place they were before, which amounts to no change.

If you update a cell further down and to the right, the block of columns to its left remains cacheable, as do the block of cells above it — again, assuming they aren't invalidated by a changed `col_width`. In cases like this, you can reduce the number of invalidations by dividing screen regions up in a binary fashion rather than linearly:

```
image_of_column(win, first_row, col) =
    if win.height > 1:
        mid = win.height div 2
        image_of_column(win.subwindow(height=mid), first_row, col)
        image_of_column(win.subwindow(top=mid), first_row + mid, col)
    elif win.height == 1:
        image_of_cell(win, col, first_row)
```

(This strategy should be abstractable into higher-order functions for vertical and horizontal stacks.)

If the implementation of `show` actually stores its arguments into such a screen image, the updates to the screen image can be done without any heap allocation. However, if you're running a unified caching layer, you need to maintain the metadata that says which screen regions were produced by calling which functions with which arguments; in this case, there are about two or three such cache metadata nodes per cell that was displayed. I estimate that each such node is around 12 words, 48 bytes on a 32-bit machine, and the number of cells involved might be 100 — so 4800 bytes, less than three times the size of the screen in question. (And that's if you don't

discard any of the nodes.) I'm not sure if there's a reasonable way to manage that metadata without dynamic allocation.

Now, suppose that our screen image isn't just a flat array of bytes (or two of them), but instead, a replicated object that produces a data stream of the updates made to it to an interested observer. When a call to `win.show` is made, or if the cache copies data from one place in the screen image to another, the observer is informed.

Such an observer is just what you need to produce a stream of updates to send to a terminal emulator.

Generalizing and simplifying the screen-image example

So what do we have here? We seem to have constructed a sort of special-purpose cache manager for a certain kind of object, rectangular regions of a terminal screen. Was this necessary? How big is the benefit? And how many special-purpose cache managers will we want to make over time?

Topics

Ideas to pursue

Kragen Javier Sitaker, 2018-05-05 (updated 2018-08-16) (6 minutes)

As usual, I'm full of a lot of different ideas I'd like to investigate more deeply and develop into useful things. Among these are:

• Manufacturing robot: I want some kind of precise positioning system that I can use to deposit or remove material. The shortest path to this is likely inkjet printer carriages, but I need to figure out how to physically hook electronics up to their linear encoders, and then I probably need to rig up some kind of PID control to drive them with.

• The bed platform: I need to build a thing in my bedroom to turn it into more of a bedroom and be able to move in. This involves buying or salvaging more steel studs, hooking them together, and putting some shit on top, and then at least some kind of cloth around the edges.
• Extending the calculator: I have a somewhat awesome interactive calculator, but it still needs a lot of work. It would benefit a lot from localStorage, decimal points, and fixing editing so that I can add new numbers next to existing ones, and also from a multitouch UI.
• Multitouch UI: I have a promising prototype that isn't yet good enough to do anything useful.
• Ice vest: something to enable me to comfortably weather the summer in Buenos Aires.
• Dependency-driven recalculation: crossing `make` with Apache Spark.
• Graph notation: a general way to represent digraphs as relatively short sequences of symbols.
• Magic Kazoo: a synthesizer you can hold in your mouth and play like a kazoo
• Constraint-based quantitative modeling: a system for building quantitative models that can include things like three-dimensional shapes.
• Ghettobotics: bootstrapping a bitchin electronics lab from garbage.
• Quasicard: a generative form of hypertext that supports exploratory data analysis
• Stuff with optimization: a whole list of things that I should be able to attack with mathematical optimization, including:

• Dithering.
• Image approximation under other constraints, such as mosaic tiles or gradient swatches, using a psychovisual model.
• Or, for example, with an XY robot, an ink model, and a psychovisual model.
• Melody transcription.
• Magic Sinewave synthesis (with a given reactive filter and switching losses, say).
• Sparse approximation of convolution kernels (generalizations of the Hogenauer filter).
• Kinematic control.
• Topological optimization for panel cutting.
• FEM in general.

• Text layout, as TeX's badness-minimization approach. But e.g. for JSON.
• Toolpath optimization?
• Analog filter design.
• Relatedly, IIR filter design.
• Structure from motion, photogrammetry, and structure from shading.
• Tool choice from mesh model and tolerances — using more precise tools where tighter tolerances are needed.
• Circuit design from available components (measured e.g. with an M328)
• Execution planning in e.g. a query planner, where there are multiple possible ways to compute the same result.
• Index design for a database, which is the same thing taken up a level of abstraction — which indices would minimize query cost?
• Weld placement on weldments to optimize their strength
• Automated bricolage from gluing together found objects
• Mesh complexity reduction with bounded error
• Curve approximation with minimal numbers of control points
• List decoding of noisy signals
• Signal approximation (of e.g. audio signals) from given primitive signals and combining operators and a closeness metric (e.g. a psychoacoustic model). One aspect of this is audiomontage, where you combine prerecorded sounds to produce a desired sound, as in notjustmoreidlechatter.
• Speech recognition, given a speech synthesis program and a psychoacoustic model.
• Generating knitting patterns from triangle-mesh 3-D models
• Generating laser-cutting patterns from mesh models
• Cost optimization of laser cutting under various constraints, e.g. bounded error to a mesh model, or visual similarity using a psychovisual model.
• Cache allocation optimization
• Planning a travel path through a city to minimize cost, discomfort, worst-case or average-case travel time, or some combination
• Image coregistration
• Tensegrity design given constraints (e.g. maximum rigidity for comfort, minimum rigidity, minimum strength, impact energy, strength under given loads, support points)
• Graded-index optical systems with no surface scattering
• Materials property estimation from experimental results
• Circuit or electronic component estimation from experimental results
• Design of linkages or composite materials to produce a given force-displacement curve or force-displacement-time curve
• Inverse filtering to compensate precisely for known output transducer imperfections, including nonlinearities
• Also, of course, for input transducer imperfections
• Optimizing a predictor of the next sample used for compression (including lossless compression) for a given image or sound.
• Estimating a convolution kernel from two simultaneous signals, such as a stereo recording of someone talking in a church. Basically this is just inverse convolution.
• Beamforming

• Toolpath planning not just from a 3-D model but also, for example, to approximate a given grayscale image by, for example, carving plaster.
• Attributing motivations to human choices, e.g. social network analysis from Facebook "like" data
• IQlight design from mesh
• Robot path planning
• Robot attitude and position information from sensor data — actually SLAM as a whole
• Most-probable-fault diagnosis from a fault tree and observations
• Inferring a fault tree from observations
• Experiment design for fault diagnosis given a fault tree and partial information
• Learning a model of a controllable system (e.g. a robot arm) in order to control it and know where the controllable zones are
• Designing a procedure for a controllable system to optimize the important tolerances instead of the less important ones; this broadly includes things like planning to cast surfaces that don't need precision followed by grinding surfaces that do, but also planning to position workpieces such that the things that need precision are in the zones where your actuator and feedback are most precise
• Constraint satisfaction for 3-D modeling
• qyap
• a small, safe IRC client
• a daisy-chaining bus for simple programmable electronics

Topics

Fast sea salt evaporator

Kragen Javier Sitaker, 2017-06-01 (3 minutes)

The standard approach to making sea salt is to evaporate the seawater in salt ponds over a period of months to years. This time period is determined by the depth of the salt layer you want to end up with, the concentration of salt in the water (35 g/ℓ), the enthalpy of vaporization of water (2.26 MJ/kg), and the terrestrial solar constant — the irradiance from the sun at the surface of the Earth. (Or, really, its average over time, filtered through the atmosphere and clouds — "mean insolation". Peak terrestrial solar irradiance is typically about 1000W/m² but mean insolation at temperate latitudes is only about 180–280 W/m².)

(A small additional amount of heat is also added by air as it blows over the water, bringing in some solar heat from the surrounding area.)

However, the typical practice is actually to move the brine to smaller pools once it becomes more concentrated, thus evaporating most of the water over a large-area solar collector, then piling up the salt.

The vapor pressure of the water is dependent on the salinity; pure water will only condense at a relative humidity of 100% (by definition), but in the final stages, where the salt solution is saturated (359 g/ℓ) and salt begins to crystallize, the equilibrium favors condensation above 75.5% relative humidity (at 0°, slightly increasing with temperature). So heating the air somewhat may be necessary for the final stages of evaporation, to keep them from running backwards.

In theory, you should be able to adapt this progressive concentration to a continuous-flow process with a very large number of "ponds" and a very small amount of water, thus producing a continuous flow of salt very rapidly from seawater.

While the sun is shining at 1000W/m², water evaporates at 442μℓ/m²/second, producing salt (if the process gets that far) at 15.5 mg/m²/s. Given 4m² of solar collectors, this could give you 62 mg/s of salt, or a kilogram or two of salt per day, depending on how much the sun shines. If your collectors are merely water ponds 10mm deep, the collector contains 40 liters of water at any given time, which means that any given parcel of water on its way through the system will take about 6 hours to evaporate completely.

Possibly a more interesting approach is to use reflective solar concentrators to collect sunlight over a much larger area than the area actually covered in water. For example, if you could achieve three suns of concentration, water could pass through the system in only 2 hours, and as a bonus, you could heat it to a high enough temperature (up to 209°) to sterilize it as it entered, minimizing biofouling problems.

Topics

A Sunday in 2014

Kragen Javier Sitaker, 2014-02-24 (3 minutes)

I'd planned to do laundry at a friend's house today (since she has a washing machine and I don't), and to write some code for the $work project. First, I went out to find the hours for Teatro Ciego, petting a small spotted dog locked up in a neighborhood store on the way home, and then at home I began to write about a friend's idea.

Much to my surprise, the doorbell rang; a French sculptor I hadn't seen in maybe a year had dropped by unannounced, and so we went to sit nearby to meet a tall lesbian musician from Darmstadt. Neighborhood kids were playing dodgeball noisily against the metal shutters closing a storefront, so we sat elsewhere and caught up on our respective lives.

The musician arrived, irritated at having taken the wrong bus, and we walked around the neighborhood, buying empanadas on the way to the park. I introduced her to capresse empanadas, and she told me about her pessimism about Bitcoin and Wikipedia, based in part on the history of radio, in a mix of Portuguese and Spanish. Later, in my apartment, sweating from the summer heat, the three of us sipped green tea from atop a styrofoam cooler chest wrapped in cellophane tape.

Perhaps too late in the day, I texted my friend about the laundry. I never got a reply.

The three of us went to a bar where the musician would perform later. I ate a cold raw pickled eggplant burrito, which wasn't quite nauseating, as plump and tattooed young Argentines played rock to thunderous applause, despite their inability to sing in tune. Crammed between walls hung with psychedelic surrealist paintings, the crowd demanded an out-of-tune encore. A disheveled woman in a plaid shirt fanned herself and, inadvertently, me, with a folding fan, as I drank a Speed Unlimited energy drink — like a Red Bull with less vitamins.

The musician got up to play. She sounded like Janis Joplin, and is by far the most skilled musician I've ever heard perform in this bar. Instability in the power supplies for their green LEDs gave rise to a distracting yellow flicker in the spotlight.

A bespectacled young man with a mustache at the table in front of me, wearing the only button-down shirt in the bar, took his Android phone out of his satchel to check the time.

A tattooed young couple gazed into each other's eyes across a table nearby, stroking one another's hands before leaning slowly across the table for a quick, perfunctory kiss before parting.

Walking home, I pass a small boy in a baseball cap picking through the garbage, and then a middle-aged grandmother with her daughter and baby granddaughter doing the same. On a busier street, a young woman showed necklaces laid out on a blanket to a mother with covered hair and her two yarmulke-clad boys.

Topics

MiniOS

Kragen Javier Sitaker, 2016-12-28 (updated 2017-01-03) (6 minutes)

I want to write a minimal self-sustaining programming environment to work from.

What's the absolute minimum you need for a self-sustaining programming environment?

You need a user interface including some kind of text editor, stable storage, a compiler, and a bootloader. The stable storage thing needs to support at least some minimum of version control, so you can store previous versions of the code, if nothing else. And a textual user interface needs a font, which means you need some kind of drawing program.

It's probably also almost unavoidable at this point to support network access, hotplugging, power management, foreign filesystems, cryptography, and HTML; it would be very useful to also have type checking, JS, and a debugger.

If it's going to be self-sustaining down to the hardware design level, it also needs a CPU design, a RAM design, and some kind of circuit layout and simulation tools. If the CPU design is written at some higher level than netlists, it needs to be able to synthesize RTL from whatever the CPU design is expressed in and to synthesize netlists from RTL.

If it's additionally going to be self-sustaining down to the hardware fabrication level, it needs servo control algorithms, a cyclic fabrication system, motion planning, finite element simulation and optimization, and some 3-D geometry handling (even if only voxels).

User interface

For a visual UI, minimally you need at least one font, some kind of text layout system (to put text into lines, if nothing else), and enough rendering to get it onto the screen. It's also very desirable to have mouse support and windowing, especially for the drawing program.

A text editor needs to efficiently support movement, insertion, deletion, text search, and cut-and-paste, at least on the sizes of files you're likely to encounter. You may also need some interface to load and save files in it, if you have files.

There also needs to be some kind of way to invoke other programs that you've written; Emacs does this with ^J and M-x, as well as the shell.

You also need drivers for keyboard (or touchscreen) and mouse, if present.

Stable storage

You need some kind of filesystem. It doesn't need to necessarily be a traditional hierarchical filesystem, although that would ease compatibility with existing systems, but there needs to be some way to not have to retype everything from memory every time you power-cycle the machine. Smalltalk and other image-based environments do okay here, but they imply you need some kind of hot code upgrade facility, and then you have to build in some other way of doing version control.

Forth's filesystem is probably the most minimal here: the "files" are sequentially numbered disk blocks.

Compiler

For a self-sustaining system, interpreters are optional, but compilers are mandatory. The compiler can be very simple, down to simply concatenating prewritten snippets of code and fixing up pointers, but it needs to exist, or you can't ever run your code. Furthermore, you need two duplicates of it: an executable bootstrap compiler or interpreter that can run on some existing system, and a source-code compiler that can run under the bootstrap as well as compiling itself.

My experience with Ur-Scheme makes me think that dynamic typing, a relatively simple grammar, ruthlessly polysemic data types, and making everything explicit will minimize the difficulty of writing the self-compiling compiler. My experience with peg-bootstrap and Prolog makes me think that backtracking and similar logic-programming or constraint-solving tricks can simplify tokenization and parsing down to a very simple task. My minimal experience with Forth makes me think that you probably do want syntax and typechecking.

Bootloader

The bootloader is almost entirely dependent on the environment you're running in. UEFI lets you load whatever you like from a certain version of the FAT filesystem, already in 32-bit mode. BIOS is hairier, but you still only need a few dozen instructions.

Network access

An OS without TCP/IP is not useful for most purposes nowadays. Wi-Fi, maybe via a USB dongle, is perfectly adequate at the physical level, but you still need the whole stack on top of that. Contiki's lwip is the standard in tiny TCP/IPs, but it's sixty thousand lines of C. The VPRI tiny TCP/IP stack is much smaller, but I don't know where to find it or how to get it running.

Hotplugging

Lack of hotplugging is what doomed sysvinit in Linux, leading to its replacement with a poorly-designed monstrosity called systemd. I'm not totally 100% sure about why plugging devices into a "hot" USB network requires rewriting the entire software stack while plugging devices into a "hot" Ethernet doesn't, but there you have it.

You need some kind of event bus to coordinate responses to either kind of event, so you can't get by with the traditional Unix IPC mechanisms of pipes and a shared global mutable filesystem, unless you want to have to poll all the time. But this broadcast/multicast IPC mechanism could totally run as a user process. Then again, in Minix even memory management is a user process.

Power management

Foreign filesystems

Cryptography

HTML

Type checking

JS

A debugger

Topics

A 2007 overview of matrix barcodes

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

There are a bunch of 2-D barcode systems, with varying capacities.

PDF-417 is one of the older and better-known systems. It can handle up to 1108 bytes of binary data, and there are GPL PDF417 encoders and decoders --- http://www.totalshareware.com/ASP/detail_view.asp?application=40°1548 is "Grandzebu"'s Windows XP version, and then there's http://sourceforge.net/projects/pdf417encode/ "pdf417_encode" by "jtlien". There's a "pdf417decode" as well at http://dataconv.org/apps_barcode.html the "ooosawaddee3pdf417" homepage --- this is a very slightly modified version of Ian Goldberg's 1997 decoder from http://www.isaac.cs.berkeley.edu/tools/pdf417-1.0.tar.gz --- unfortunately that software does not have a free license, or any explicit license, actually. There's a SourceForge project at sf.net/projects/pdf417decode which claims to be GPL but seems to be the same unlicensed code, plus a bunch of new code to support error correction. There's a reasonably nice description of how PDF-417 works at http://grandzebu.net/index.php?page=/informatique/codbar-en/pdf°°417.htm.

DataMatrix/Semacode is a higher-capacity 2-D barcode, with up to 1556 bytes; I don't know if it's patent-safe. There's a GPL decoder in C# at http://datamatrixdec.berlios.de/index.php/Main_Page with a couple of releases from late 2006.

QR Code is the highest-capacity 2-D barcode in common use, with up to 2953 bytes (when it's 177x177 pixels); with gzip -9, that's enough space to encode the Project Gutenberg version of Genesis up to Genesis 3:5, 1624 words. Denso-Wave has promised not to enforce its patents against it. There seems to be a lot of activity around it lately, especially in Japan. There's a GPL decoder in Java at http://qrcode.sourceforge.jp/ or http://sourceforge.jp/projects/qrcode/ --- ThoughtWorks's .NET library doesn't seem to be free software. QR code is JIS-X-0510 and ISO/IEC18004. There's a comprehensive introduction at http://www.denso-wave.com/qrcode/qrfeature-e.html and an occasionally-updated blog in Japanese at http://www.qrcodeblog.com/.

Topics

Approaches to 3-D printing in sandstone

Kragen Javier Sitaker, 2017-08-03 (5 minutes)

There are five common adhesives or cements which work to turn sand into sandstone at low temperatures: plaster of paris, portland cement, slaked lime, clay, and sodium silicate.

Plaster of paris (dehydrated gypsum, i.e. calcium sulfate hemihydrate) is soft and weak. It is inert until exposed to water; then it sets in about 45', producing a fair bit of heat. It will set underwater, is nontoxic, and is a bright white. If used outdoors, rain will gradually erode it. It introduces under 100µm of surface roughness. On MercadoLibre 30 kg goes for AR$162 (at AR$18.20/US$, that's US$8.90 or US$0.30/kg).

Portland cement (calcium silicates with some aluminum and iron silicates) is strong and hard. It takes about 6 hours to set, which can happen underwater, but part of the setting process involves absorbing CO_2 from the atmosphere. It is highly alkaline and can cause chemical burns. It has a substantial thermal coefficient of expansion, which can cause cracks. The results are not suitable for high-temperature use (e.g. metal casting) because the trapped water will cause steam explosions. It is typically gray. On MercadoLibre 50 kg goes for AR$142 (US$7.80 or US$0.26/kg); the white grade costs about three times as much.

Slaked lime (calcium hydroxide) can be mixed with portland cement or applied alone. It sets entirely, over the course of hours to days, by absorbing carbon dioxide from air, and consequently will not set underwater. It's nearly as strong and hard as portland cement, but has a much smaller thermal coefficient of expansion. It's even more alkaline. The finished material is porous, although it can be waterproofed with soap, forming materials known as qadad and tadelakt. It forms a brilliant white. On MercadoLibre 25 kg goes for AR$67 (US$3.70 or US$0.15/kg).

Clay is a class of phyllosilicate hydrate minerals that plasticize significantly with the addition of more water, absorbing the water and expanding, then contracting again and hardening as the water evaporates. This process can happen very rapidly, but the strength of the resulting material is very low. However, the product can then be fired in a kiln, first dehydrating the clay and then sintering the clay particles into a ceramic. Bentonite clays are the traditional adhesive in the greensand used in metal casting; they are among the most plastic and expansive. Ball clays and kaolin fire to a bright white color. 25kg of bentonite costs AR$180 on MercadoLibre (US$9.90 or US$0.40/kg).

Sodium silicate is a class of mixtures of silica and sodium oxide; it sets over the course of days by absorbing carbon dioxide, and can be hardened instantly by the application of concentrated carbon dioxide gas or liquid acids. Some grades are very alkaline, but those are more difficult to find. It dissolves in water, forming a dense solution; generally this requires high temperatures and pressures, so it is usually sold predissolved. I haven't compared its strength to the other

cements mentioned above. It's sold on MercadoLibre as "bloqueador silicato" for AR$1670/20ℓ (US$91 or US$4.60/kg), although I think I've found somewhat lower prices.

There are other cements, including magnesium oxychloride, magnesium phosphate, magnesium oxysulfate, calcium aluminate, and bacterially-decomposed urea with calcium salts, and wollastonite with phosphoric acid. Unfortunately, I don't know where to get the ingredients here, or they have other disadvantages such as the smell.

Construction sand itself goes for AR$575/m³, and a m³ is about 2.4 tonnes. This is US$32 (US$0.013/kg). So to the extent that you can replace binder with sand filler, you can drop the materials cost of your printed object almost proportionally, while increasing its strength; with binders mostly in the US$0.15–0.40/kg range, except for sodium silicate's US$4.60, the sand doesn't comprise half the cost of the object until the binder is down to 3%–10% of the mix, and generally none of these binders are so awesome as to allow you to use so little binder.

For 3-D printing, whether powder-bed or nozzle-deposition, the optimal material would be free; remain liquid or powder forever while in storage, but harden instantly as soon as it was activated; would be very strong; and would be colorless, so that you could pigment it however you liked.

Topics

Barrel safety

Kragen Javier Sitaker, 2018-07-14 (3 minutes)

Water in 200ℓ drums is a reasonable thing to want to store in your house, both for drinking and for thermal storage, but water spills can cause serious property damage and even electrical hazards. How can we store it in an inherently leak-safe way?

This is a problem with waterbeds, too, which commonly contain 1000ℓ or so of water. The usual solution with waterbeds is to surround the vinyl bladder with a stiff plywood frame with an impermeable polyethylene liner; this way, when the vinyl eventually ruptures (as warm, plasticized vinyl under continuous stress eventually does) the water will be safely contained within the frame, as long as no holes in the liner have previously escaped detection.

You might think that this solution works so well for waterbeds precisely because the inner water container is flexible — if it starts to leak, the water outside of it will tend to collapse it, and the overall level of water will not rise above the edge of the frame. But a little thought will show that this is a fairly general property of such systems, even if the inner container is rigid. If you have a 200ℓ plastic drum of water standing on end inside, say, a 230ℓ bucket that is as tall as the drum, if water begins to escape, once the water level outside the drum equals the water level inside, it will stop rising, even if the drum is floating at that point (which will happen with polypropylene drums). Hopefully some kind of alarm will have gone off before this point, intake shutoff valves will have been closed, drains will have been opened, etc.

However, an additional layer of protection could be provided by a large tray holding several drums. Consider a rectangular array of N round 200ℓ drums next to each other. Each occupies π/4 of its square cell, 79% of it, leaving the other 21% empty. If you have four such drums in a 2×2 array, the empty space in the tray amounts to 86% of a square, which is 109% of the volume of one of the drums. So, if any one of the four drums breaks open and spills its contents, walls around the four-drum unit up to their height would keep any of it from escaping, even if the bottle were filled with air until all the water escaped. In a 4×4 array holding 3.2 tonnes of water, walls up to ¼ of the height of the drums would be adequate to provide such a containment service against the total failure of any single barrel.

This is not the only safety measure that is needed — as I said, automatic shutoff valves, drainage routes, and alarms are highly desirable, and of course there are the usual issues of cleanliness.

(Relatedly, I've previously written about inherently-safe fuel storage by floating small bottles of ethanol in water tanks.)

Topics

- Independence (p. 3520) (63 notes)
- Household management and home economics (p. 3504) (44 notes)
- Water (p. 3773) (13 notes)

Immersion plating of copper on iron with blue vitriol

Kragen Javier Sitaker, 2016-09-24 (8 minutes)

I'd previously thought about copper-plating everything in the house like US pennies for antibacterial reasons, but it seemed like it would be a huge hassle, what with cyanides and electric current and so on. But it turns out that iron in particular has an extremely easy electroless way to plate it, discovered by alchemists many generations ago: immersion in solution of blue vitriol oxidizes some iron on the surface into green vitriol, replacing it with copper.

This might be interesting for decorative purposes as well, including selectively marking part of an iron or steel surface.

Nowadays this is called "immersion plating" in the "metal finishing" trade, whose bible is the Metal Finishing Guidebook, or sometimes "displacement plating". There's a standard test for cleanliness using 2 oz./gal. of copper sulfate with 0.1 oz./gal. of H_2SO_4 RT, I suppose to immersion-plate copper wherever the steel is clean. A bronze-displacement-plating solution uses 7.5 g/ℓ stannous sulfate, 7.5 g/ℓ copper sulfate, and 10–30 g/ℓ sulfuric acid at 20° for 5 minutes.

Stannous sulfate is deliquescent but relatively nontoxic; you can make it by a displacement reaction with tin and copper sulfate, so it isn't necessary to buy it. I suspect you can make it with tin-lead solder and copper sulfate; if lead sulfate forms, it should be nearly insoluble.

At times this copper deposits in a powdery fashion, and apparently this also works on zinc, or more generally can displace any less noble metal with any more noble metal; reading on the "galvanic series" or "seawater series" or "electrolytic series" is suggested. Instructables has photos of the powdery copper deposit and reports better results with some electroplating. Caustic cleaning and an acid dip ahead of time are reported to give better results; electroplating success with salt and ethylene glycol was also reported to work, with the electrolyte recipe 100 cc white vinegar, 1 heaping teaspoon kosher salt, 3–6 cc of ethylene glycol.

Professional metal finishers don't recommend trying to do this, although they're concerned about questions like durability and pore-freeness as well as appearance, antisepsis, and adherence. They explain that the reason copper cyanide or copper pyrophosphate works better for electroplating is that the copper is "tightly complexed", so it won't plate out onto the steel without a current applied. "Finishing.com has been on the air 20 years now and no hobbyist has ever reported back that they achieved robust copper plating on steel from kitchen or hardware store chemistry. Maybe you'll be first; I hope so," immediately followed by someone reporting success in electroplating with copper sulfate from an electrolytic-copper anode, sulfuric acid, phosphoric acid, and dish detergent as a brightener.

The electrode potentials are confusing. More noble metals seem to be more positive, while more reactive metals are more negative. Zinc

makes its first appearance at -1.2, and iron at -1.2 also (but with cyanide), with tin at -1.1, alkaline iron at -0.9, silver (with sulfide) at -0.7, gold (with cyanide) at -0.6. Silver's last appearance is at +2.0. Copper can be displacement-plated with tin, although it's not trivial, and tin can also be displacement-coated with copper (with just copper sulfate).

In particular, though, it seems like immersion plating of silver and gold onto copper or onto nickel are popular; I'm not finding much information about immersion-gold-onto-copper (ECIG and related processes) though. One paper on immersion-gold-onto-nickel gives the recipe of "immersion gold bath…at 80 °C, where 2 g ℓ^{-1} $Na_3Au(SO_3)_2$, 40 g ℓ^{-1} Na_2SO_3, 15 g ℓ^{-1} NaS_2SO_3, and 10 g ℓ^{-1} $Na_2B_4O_7{\cdot}10H_2O$ were contained".

The important thing about the potentials, I think, is that iron is more noble than zinc, tin and copper are more noble than iron, silver is more noble than tin and copper, and gold is more noble than silver.

Popular Mechanics May 1906 gives the recipe of 1 oz. blue vitriol dissolved in 6 oz. water with ½ oz. sulfuric acid, but I think it was suggesting using that as the electrolyte for a copper-zinc battery for electroplating.

The ASTM test A-239 describes how to use the Preece Test to find the thinnest spot in a galvanized coating. The current version is A-239-14. The 1995 version A-239-95 prescribes 36 g of copper sulfate pentahydrate per 100 g of distilled water, plus an excess of $Cu(OH)_2$, about 1 g/ℓ, enough to not dissolve completely, or CuO of 0.8 g/ℓ; the solution should be 1.186 g/cc at 18°. The result is that this removes the zinc coating and, when it gets down to the steel underneath, deposits "bright, adherent copper deposits" that cannot be removed "with an ink eraser".

Instructables has a CC BY-NC-SA copper acetate electroplating recipe, using acetate from vinegar and 3% H_2O_2. It comes with an excellent introduction to electroplating. (Copper acetate is somewhat toxic. It bioaccumulates with a biological concentration factor of over 100, but it was used as a green food coloring in the 19th century and had a large death toll.)

Copper sulfate is sold as fertilizer and fungicide for AR$145 per kg (US$9.50/kg) by Cristina at e-Moyos on MercadoLibre. This is the pentahydrate, which is 25.5% copper by mass (because it's $CuSO_4$, 159.6 g/mol, water is 18.02 g/mol for a total molar mass of 249.7 g/mol, and copper is 63.55 g/mol), so this works out to US$37/kg for the copper. This is not a good price for copper; the USGS average 2015 published price for copper is 277¢/lb., which works out to US$6.11/kg. But it's high by less than an order of magnitude. The Amazon price is US$33.40/5 lb., which is more expensive at US$14.70/kg.

(It loses four of its water molecules, amounting to 28.9% of its mass, upon heating to 109°, and reabsorbs them at 63°, so it might be useful as a powerful and compact desiccant; however, upon heating to 650° it emits SO_3, so it may not be very safe, depending on how well-controlled the regeneration temperature is. But this is beside the point.)

Machinery's Shop Receipts and Formulas gives several recipes for such things; one of them suggests "brassing" with "a quart of water and ½ ounce each of sulphate of copper and pro-tochloride of tin",

which would have the great advantage that brass is both more yellow (less red) and a lot easier to keep polished. Nowadays tin protochloride, which is what's used for tin-plating cans, is called stannous chloride or tin(II) chloride, and it's a relatively safe chemical (in fact it's E512), but apparently a little bit tricky to deal with in solution. It's a lot harder to find than blue vitriol.

Other recipes from the collection suggest using e.g. "hydrochloric acid diluted with three times its volume of water, in which a few drops of a solution of sulphate of copper is poured". Others suggest using e.g. zinc chloride: "saturated solution of zinc chloride with a very little copper sulphate added, say a half-dozen drops of copper sulphate to a spoonful of zinc chloride solution". Zinc chloride is a deadly corrosive deliquescent salt of zinc that is considerably more easily available than stannous chloride because of its wide use as an acid welding flux.

Another recipe from the collection suggested caustic cleaning followed by charcoal cleaning, presumably to remove the caustic.

US patent 3,715,289 from the early 1970s gives a formula for a rather complicated brightener for copper electroplating; it says previous brighteners have included "casein, animal glue, sugar, urea and thiourea and their derivatives and polyvinyl alcohol."

Topics

Relational modeling

Kragen Javier Sitaker, 2017-05-17 (updated 2017-06-01) (6 minutes)

Suppose I want a cylinder with a given mass and aspect ratio made out of a material with a given density. For example, an aspect ratio of 10:1, made of quartz with a density of 2.65 g/cc, and weighing 1kg.

It's easy to write down and verify the equations that govern this system:

```
diameter = 2 · radius
area = π · radius²
volume = area · length
mass = volume · density
aspect_ratio = length / diameter

mass = 1 kg
aspect_ratio = 10/1
density = 2.65 g/cc
```

And it's not that hard to solve algebraically:

```
length / diameter = 10
length = 10 · diameter = 20 · radius
volume = π · radius² · 20 · radius = 20 · π · radius³
volume = mass / density = 1 kg / (2.65 g/cc) ≈ 377 cm³
377 cm³ ≈ 20 · π · radius³
6.00 cm³ ≈ radius³
1.82 cm ≈ radius [excluding the two complex solutions]
diameter ≈ 3.64 cm
area ≈ 10.4 cm²
length ≈ 36.4 cm
```

Except I got it wrong the first time I did it, coming up with a mass of 552 grams instead of 1000, because I calculated π 1.82 cm² instead of π (1.82 cm)², spending about 10 minutes on the problem. Then when I tried to apply it to some other cases in a spreadsheet, I accidentally used 20 instead of 2 · aspect_ratio, getting nonsensical answers.

The standard approach to reducing the hassle of problems like this is to solve the equations algebraically to get a procedure to compute radius, diameter, cross-sectional area, and length given mass, aspect ratio, and density, or perhaps just volume and aspect ratio:

```
radius = ∛(volume / 2 π aspect_ratio)
diameter = 2 · radius
length = 20 · radius
```

Then you can package this procedure up as a subroutine and use it many times, instead of doing the algebraic manipulation each time.

But it would be nicer to be able to simply specify the relations — or even refer to them from somewhere — and have the computer find a solution.

Spreadsheets offer a simple form of this as "Goal seek". In

Gnumeric and other similar spreadsheets, you can enter the problem this way, for example:

	A	B
1	Radius	5
2	Area	=pi()*B1^2
3	Diameter	=2*B1
4	Aspect ratio	10
5	Length	=B4*B3
6	Volume	=B5*B2
7	Density	2.65
8	Mass	=B7*B6

And then you can tell "Goal seek" to set B8 to 1000 by changing B1. However, this isn't composable (you can't use a "goal seek" as a formula in a cell), must be manually recomputed (or recomputed by a macro) when the inputs change, and can't be applied across a range (for example, if you have several different aspect ratios to solve for). So it's a step in the right direction, but it's awkward to use. Mac Excel has a somewhat more powerful constrained minimization solver called "Solver".

"Goal seek" and "Solver" and similar metaheuristic solvers can often find solutions to problems that have no closed-form algebraic solution.

There is a tradition of numerical constraint programming going back to the 1970s for creating graphics in systems like METAFONT, IDEAL, and Linogram, typically limited to cases that could be efficiently solved without recourse to possibly nonterminating algorithms. For interactive use, though, such restrictions seem like overkill — and, in many cases, modern solvers can make short work of problems that have no closed-form solution.

That is to say, maybe constraint logic programming over infinite domains would be a handy tool to have for calculations like this.

You could also imagine composing my example model above from existing submodels. For example:

```
circle:
    diameter = 2 · radius
    area = π · radius²

prism:
    volume = area · length

oblong:
    aspect_ratio = length / diameter

cylinder:
    circle
    prism
    oblong

uniform_solid:
    mass = volume · density

cylinder
```

```
uniform_solid
mass = 1 kg
aspect_ratio = 10/1
density = 2.65 g/cc
```

In this case everything is just all glommed into a single namespace, like with inheritance, but you could imagine composing it slightly differently with some hierarchy. For example, maybe this is a better model of a cylinder:

```
cylinder:
    circle c
    diameter = c.diameter
    prism p
    cross_sectional_area = p.area = c.area
    length = p.length
    oblong
```

Here we are namespacing the circle and prism attributes to subnamespaces, then explicitly exporting `p.length` and `c.diameter` to where `oblong` can find them implicitly.

We could imagine a froodier set of models with further properties like this:

```
circle:
    diameter = 2 · radius
    area = π · radius²
    perimeter = π · diameter

prism:
    volume = end.area · length
    surface_area = 2 · end.area + length · end.perimeter

cylinder:
    circle c
    prism p
    p.end = c
    oblong
```

Here, `prism` takes an `end` argument, which has to be something with an `area` and a `perimeter`, such as `circle`.

This shows how to include properties with values more complex than simply a number, thus enabling hierarchical decomposition of the problem (although, lacking conditional recursion, you can compile the hierarchical structure thus generated into a set of atomic variables with constraints between them). You could imagine including properties whose values are displayed as, for example, images, sparklines, or 3-D meshes.

I've written the above values with units, because that helps me a lot with interpreting things during debugging and in being assured that the result is in fact correct.

In terms of type systems, as I said, you could say that `prism` needs an `end` argument with both `area` and `perimeter`; but actually if you were to give `prism` a `length` and an `end` argument with just `area`, it could still compute the `volume`. And you could imagine that if you gave it a

`surface_area`, `volume`, `length`, and `end.area`, it could compute `end.perimeter`. I'm not sure exactly how to formalize this, but it seems like it could be useful to carry such deductions out as far as possible.

Topics

- Programming (p. 3658) (286 notes)
- Graphics (p. 3483) (91 notes)
- Programming languages (p. 3656) (47 notes)
- Syntax (p. 3738) (28 notes)
- Calculators (p. 3362) (11 notes)
- Constraint satisfaction (p. 3387) (9 notes)
- Solvers

Drone cutting

Kragen Javier Sitaker, 2019-06-11 (12 minutes)

Squirrel-cage induction motors are obsolete except in environments where weight and size is of no concern; BLDC motors are now superior in every way, often by two orders of magnitude.

The old motor

I have a ¼-horsepower 1430-rpm electric motor here, adequate for running a grinding wheel or something through the V-pulley it's got stuck on it. I think it's originally out of a washing machine; I bought it used. It's got a big capacitor wart on the side, because it's a single-phase motor, and that's a much less inefficient way to get a single-phase motors to start up than the hack in a shaded-pole motor. I haven't opened it up, but I'm pretty sure it's a squirrel-cage induction type, because it says "1430 rpm", and does seem to run at a pretty fixed speed. It's got cooling slots in the ends so you can blow dust into it.

It's rated for 2.4 amps at 220 volts, which works out to 530 W. ¼ hp is only 186 W, so presumably 2.4 A is the current it draws if the spindle stalls; these motors aren't that inefficient. I haven't measured either its power input or its power output.

I don't have a scale handy, but I feel like it weighs about 10 kg, and it's 150 mm in diameter and about 200 mm long.

The thing I want to point out here is that this is not a very good power density, on the order of 20 W/kg and 50 W/ℓ.

Milling machines

I was just watching a YouTube video by Stuart de Haro entitled "Milling machine anatomy", which is largely about the Bridgeport Model J head for Bridgeport milling machines, the most popular milling head for hobbyist metal milling in the US. We're talking about a floor-mounted mill with an X–Y table a couple of meters long, capable of precision on the order of 25 or 50 microns. (In this case it's equipped with a DRO that reads in microns, or 2.5-micron units in medieval-unit mode.) It's got this big honking motor on the back, hooked up to a variable-frequency drive, bigger than the motor I've got here, and my man Stuart explains that the Model R ½-horsepower milling head is "relatively light duty" (and for that reason he's never actually seen one, though he teaches machining at a college).

So we can deduce that, for Stuart and other Bridgeport machinists, one horsepower (746 watts in modern units) is adequate for the meter-scale workpieces they like to chew on, but half a horsepower (373 W) is "light duty" and would slow them down. And that, I suppose, is why they're willing to deal with these heavy honking motors on their milling machines.

Quadcopter motors

Modern electric quadcopter drones result from truly astounding progress in battery and motor technology. On MercadoLibre here in Argentina, I find a Turnigy BC2836-8 motor designed for

quadcopters for AR$2090 today (at AR$44.50/US$, that's US$47.) This motor weighs 70 g, measures 28×28×36 mm, wants to be driven with a 30-amp or 40-amp "ESC" (the kind of VFD you use for a BLDC motor), is rated at 1100 "K_V" (rpm per volt), and is rated at 336 W, intended to be driven from 2–4 LiPo batteries. (A Singaporean vendor lists the same motor at US$13.)

That is, this motor is almost *twice the power* of the big honking 10-kg beast I have here in my living room. But it weighs a bit under 1% of what the big motor does. I'm guessing it costs about 10%, too.

Four 3.7-volt LiPo batteries in series would be 14.8 volts, and 336 watts at that voltage would be 22.7 amps, so the volts and amps pretty much check out. At 1100 "K_V", its maximum speed should be a bit over 16000 rpm, which is pretty plausible.

So this works out to 4800 W/kg and 15000 W/kg, about 200–300 times better than this big motor.

In part this is made possible just by running the thing eleven times faster, which is made possible by having much better bearings and designing the thing to depend on an ESC. At a given torque, running the motor eleven times faster is going to give you eleven times the power. Permanent-magnet brushless "DC" motors like this one ("BLDC motors") also use rare-earth magnets, typically NdFeB, which gives them higher field strength and thus higher torque — although in theory it's possible to achieve similarly-high flux densities in induction motors. (The "electrical steel" used in squirrel-cage cores saturates at 1.6–2.2 T, while NdFeB's remanence is "only" 1–1.3 T.) Another significant factor may be cooling: the drone motor is of course designed to operate in the propeller downwash, which is a wonderful level of air-cooling. Finally, the drone motors are presumably designed for an MTBF of tens of hours, while the ¼-hp motor is designed for an MTBF of tens of thousands of hours.

BLDC motors can also maintain the optimal phase relationship between the rotating magnetic field applied to the stator and the magnetic field of the rotating rotor, enabling them to maintain the same torque at any speed; induction motors, by contrast, have a fixed torque–speed relationship which reaches zero torque at their natural or unloaded speed.

Permanent magnets are much smaller than field windings, which exist in the induction motor in the form of the copper "squirrel cage" within the rotor. This may account for a substantial fraction of the mass and volume of the motor, though far from 99%.

Neodymium magnets also have about an order of magnitude greater resistivity than iron does (1.1–1.7 μΩm, compared to iron's 0.1, though electrical steel's resistivity is higher than that), which might diminish eddy-current losses, and of course there are no hysteresis losses; although these are more properly efficiency concerns rather than power-density concerns, they do have some effect in that more power losses result in more necessary cooling.

Sometimes the low Curie temperature is cited as a disadvantage of neodymium magnet motors — it's only 310–400°, so the magnets will be destroyed if the motor ever overheats that much. However, I think the solder joints and winding insulation will fail at a lower temperature than that.

Milling with drone motors

So how could you build a milling machine or engine lathe with these little motors? Well, you probably need several of them, such as ten of them. (You don't want to be running the motors at their maximum power if you want them to last; as I said above, they aren't designed to last.) You'd need to gear them down. Our homeboy de Haro tells us that the spindle speeds in his classroom range from 50 rpm to 5000 rpm; trying to run that off this wimpy ¼-horsepower squirrel-cage motor I have here could require gearing it *up* by a factor of up to 3.5, or down by a factor of almost 30, although in actual fact you would probably use a VFD.

By contrast, you'd probably want to run the drone motors faster and gear them down. Maybe you could run them at 2000–6000 rpm, for example, although that difference in speed is less than I expected: you'd be gearing them down by factors of only 1.17 to 40, probably. You could imagine, though, that you'd want to take advantage of the drone motors' higher speeds to support higher milling speeds for small cutters.

Dremel-style cutting

Dremel-style small-cutter milling is potentially really interesting, especially coupled with the larger number of axes of control that smaller, cheaper, higher-power motors and control systems make possible. The conventional CNC way to mill weird shapes — shapes that are far from the shape of the stock they're milled from — is to convert most of the stock into chips, a slow and expensive process that becomes somewhat faster if you apply a large roughing milling cutter and a high-powered motor to the task. Smart machinists, and all but the most stubborn manual metalworkers, often use slitting saws and bandsaws to quickly, but imprecisely, remove much of the material ahead of time, thus avoiding much or all of the milling. With a six-axis machine, though, you could do this step under CNC control, using not only slitting saws but also narrow endmills or even drill bits or wire saws. Narrow endmills demand potentially much higher spindle speeds to reach the same surface speed, which is necessary for optimal tooth life.

Narrow endmills suffer from extremely low rigidity, resulting in chatter, imprecision, and potential breakage inside the part. Tapered endmills improve this situation dramatically; they already exist and are occasionally used in CNC milling even on three-axis machines. ConicalEndMills.com suggests using them for "draft angle & chamfer machining in all materials," for example. But maybe they could be much more widely applicable with high-speed motors and five- or six-axis control.

Fluid-cooling of stators

Wikipedia tellls me that the obstacle to even higher power densities in small BLDC motors is commonly heat dissipation: the motor must be small to spin fast, but this limits its available cooling surface area. Permanent-magnet BLDC motors put the windings, which are the primary heat-generating part of the motor, on the stator, which has a solid connection to the outside world — this facilitates getting the heat generated within them out, as well as getting the prodigious amount of current they use in. However, most of them still rely on air cooling

at this point, especially in the realm of quadcopters, where unobstructed high-speed airflow is not only guaranteed by construction but kind of the whole point.

Heat pipes are one possible way to improve the situation: you can run heat pipes through the stator to get heat out quicker. Heat pipes, unlike thermal conduction, can transfer heat at a rate that does not diminish with distance.

Another alternative, though, is forced-fluid cooling, in which a coolant fluid is pumped through channels in the stator to transfer heat out. Air is one common coolant (and, as I said above, in common use for these motors) with many advantages but also some drawbacks — its heat capacity is orders of magnitude lower than other viable fluids. (See Coolants (p. 3235) for a survey.)

One of air's key advantages as a coolant is its low viscosity, which enables it to cool even long, thin channels effectively. But, by adopting a fractal geometry for the cooling channels similar to that of vertebrate circulatory systems, we can enable the rapid, efficient circulation of even fairly viscous coolants. See Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) for more details and further applications.

Topics

Simplifying computing systems by having fewer kinds of graphics

Kragen Javier Sitaker, 2015-10-13 (10 minutes)

One way you could simplify computing systems is by having fewer redundant kinds of graphics. Typical computing systems have many different systems for windowing, drawing text, specifying fonts, making translucent windows, representing vector graphics, drawing and representing 3D graphics, zoomable graphics like maps, and doing text layout, among other things. For example, you might have VT-100 emulation, Xlib, PDF, Quartz, SVG, <canvas>, HTML with the CSS box model, PostScript, OpenGL, POVRay, H.264, MPEG-4, PNG, GIF, and JPEG, all on the same machine. Each typically has different tradeoffs related to performance, flexibility, and visual quality, but most of them just suck on all axes.

What if we had a graphics system that was both sufficiently expressive to cover nearly all of these applications, but also sufficiently performant to run in real time, while having a sufficiently simple implementation as to be understandable by a single person? Dan Amelang's Gezira and Nile are an influential effort in this direction (for the aspect of graphics that involves rendering to pixels, anyway), but they're resolutely two-dimensional.

Immediate-mode versus structured-mode

In my view, immediate-mode graphics APIs like <canvas> and PostScript have more predictable performance and substantially simpler code than structured-mode graphics systems like SVG.

File formats

File formats and graphics APIs are intimately related. At the simplest level, you can treat a file format as a "graphics API" in the sense that you can draw stuff by piping a stream of bytes to a decoder for that format; but the relationship goes the other direction too.

Of course, structured-mode graphics systems have this nailed down: all they have to do is serialize the in-memory object graph using a generic serialization system, and they're done. Direct-mode APIs are more complicated.

A "recording" of a sequence of direct-mode drawing operations can be "played back" by reinvoking the same operations in a new context, so a drawing API is in some sense capable of being serialized as a file format. This is the idea behind, for example, the WMF "Windows Metafile" vector format, which is just a serialized sequence of Windows GDI drawing operations. However, file formats and drawing APIs have some divergent needs.

First, graphics file formats typically benefit from having some kind of non-sequential access, for example for drawing particular areas, particular layers, or particular pages. Typical ways of selecting the graphics of interest doing this include bounding boxes, quadtrees, k-d trees, BSP-trees, and bitmaps of grids.

Second, programs using drawing APIs often want to make decisions about what to draw based on conditions that hold in a

particular case. For example, level-of-detail rendering is the name for a family of techniques that render images in more detail when you are zoomed in to see them, consuming more time, including things as simple as approximating Bézier curves with a larger number of straight lines; and of course programs often use bounding boxes to avoid spending time drawing objects that are offscreen or invisible because of some other clipping or occlusion.

You could imagine running such a program under some kind of backtracking replay system that inspects it to see what environmental conditions it's testing, snapshotting it at each test and later re-executing the conditionally-skipped parts, in order to derive the first of these things from the second. But that's not going to be universally applicable and anyway it's kind of rocket science.

A less-transparent, lower-tech approach would involve executing level-of-detail-conditional or bounding-box-conditional code in a fashion like how IMGUI libraries handle the logic for menus and windows that aren't currently being displayed:

```
if (bbox_visible(x0, y0, x1, y1)) {
    render_foo();
    render_bar();
    for (int i = 0; i < baz_count; i++) {
        render-baz(bazzes[i]);
    }
    end_bbox();
}
```

With this approach, interactive drawing can avoid rendering things inside the given bounding box if it is outside the display bounds, simply by returning false from `bbox_visible`, and metafile rendering can always return true from `bbox_visible`, but nest the objects thus constructed inside a bbox. Optionally, metafile rendering code could return false when you're inside a sufficiently small bbox, in order to support programs that want to do infinite level-of-detail rendering.

(My entire premise here is that we should forget about 2D graphics and just do 3D graphics, leaving 2D graphics as a special case where you're using an orthographic projection or something. So really these would be bounding volumes, not 2-D bounding boxes.)

Third, both direct-mode and structured-mode drawing systems are capable of returning general information to the program that's doing the drawing, not just pointwise questions like "is this layer visible" or "is this bounding volume visible". This is hazardous to file

Tiny POVRay code

POVRay is among the most flexible of these graphics systems. For example, https://mscharrer.net/povray/rays/ is an animation of manta rays swimming in a hazy ocean, made from these 464 fairly obfuscated bytes of POV-Ray code, written by Jeff Reifel in 2008:

```
#local
C=clock*pi;#macro
B(N,F)sphere{0F/7
1scale
1-pow(I.5)translate-I*F*x
```

```
rotate
y*N*90rotate-N*x*pow(5I)*10*sin(I*2-C*8+i)scale.2+x*.8translate-x}#end#local
i=C;#while(i<2*pi+C)#local
I=0;blob{#while(I<1)B(1,7)B(-1,7)B(0,3)#local
I=I+.01;#end

rotate<-90cos(i*3)*-45i*pi*36>translate<sin(i)+2*sin(2*i)5+cos(i)-2*cos(2*i)3*sin
(3*i)+7>*2rotate
x*37pigment{slope
y}}#local
i=i+pi/8;#end
light_source{<0,60,99>1spotlight}media{intervals
6scattering{2rgb<.1.2,1>/99}}
```

However, rendering the 300-frame looped animation on that page involved casting 685 million rays and took fifty thousand CPU seconds, two or three minutes per frame. That's only about 2 million rays per frame or 14000 rays per CPU-second.

What if your drawing primitives were sufficiently powerful to allow you to get graphical effects in such a tiny amount of code? It'd be a little bigger without the minification, but not that much. My POVRay is pretty rusty, but I think it is supposed to read as follows:

```
#local C=clock*pi;

#macro B(N,F)
    sphere {
        0F/7
        1
        scale 1 - pow(I, .5)
            translate -I*F*x
                rotate y*N*90
                    rotate -N * x * pow(5I) * 10 * sin(I*2 - C*8 + i)
                        scale .2+x*.8
                            translate -x
    }
#end

#local i=C;

#while (i < 2*pi+C)
    #local I=0;
    blob {
        #while (I < 1)
            B(1, 7)
            B(-1, 7)
            B(0, 3)
            #local I=I+.01;
        #end

        rotate <-90, cos(i*3)*-45, i*pi*36>
            translate <
                      sin(i) + 2*sin(2*i),
                      5 + cos(i) -2*cos(2*i),
                      3*sin(3*i) + 7
```

```
                    > * 2
                rotate x*37
                    pigment { slope y }
    }
    #local i=i+pi/8;
#end
light_source { <0, 60, 99> 1 spotlight }
media { intervals 6 scattering { 2 rgb<.1, .2, 1>/99 } }
```

That's still only about 35 lines of code. I mean, who knows how long he took tweaking it.

One promising approach to things like this is to use some kind of interval arithmetic or Monte Carlo rendering for level-of-detail rendering: render as much as you can before the frame deadline and display the result, and add more detail as long as the scene remains static. This is what Blender does, for example, with rotations of large meshes.

Computing performance

One of the biggest levers we have available now to simplify things is computing power to do things we couldn't do in the past, just because it was too expensive. What's the smallest amount of computing horsepower we can expect?

A Raspberry Pi 2 costs US$40 right now and can do about 250 megaflops on the CPU and 24 gigaflops on the GPUs, supposedly; a 64-Pi Version 1 Model B cluster hit 1.1 GFLOPS on LINPACK, or about 17 megaflops per Pi, and version 2 is supposedly 4 to 6 times as fast in aggregate, thus 70 to 100 megaflops. Hackaday got 93 double-precision megaflops per core, and 1186 VAX MIPS, on the Pi 2. Also, it was able to shade 900 texture-mapped triangles at 40 fps, although previous tests erroneously reported twice that at about a megapixel of resolution. That's only 36000 triangles per second.

Supposing that we can get 3 gigaflops out of a Pi 2 (geometric average of the 400 megaflops from the Hackaday results and the 24 supposed gigaflops from the GPUs) and we want 60 frames per second at 1 megapixel, we can spend up to about 50 floating-point operations per displayed pixel. Maybe I'm naïve, but that seems like it ought to be enough to do pretty decent antialiased rendering of some textures.

Derivative and interval approximations of imagery

Suppose you calculate a sparse approximation of the gradient of the screen image (the Jacobian, I guess) relative to the quantities in a scene model. Then, when you update the scene model slightly, you can multiply the update through your sparse gradient matrix to get a linear approximation of the change in the rendered image.

Calculating a gradient of the screen image relative to the scene model sounds like rocket science, but apparently automatic differentiation is now a well-established technique that can be applied to FORTRAN scientific codes of substantial size. In forward mode, it implies a constant-factor slowdown.

Alternatively, instead of doing the rendering with points and

derivatives, you could do it with interval arithmetic, which allows you to determine conservatively how big of a change in the input is needed to create any change at all in the rendered scene.

Topics

- Performance (p. 3621) (149 notes)
- Graphics (p. 3483) (91 notes)
- Pricing (p. 3646) (89 notes)
- Small is beautiful (p. 3714) (40 notes)
- Interval and affine arithmetic (p. 3528) (24 notes)
- BubbleOS (p. 3352) (17 notes)
- 3-D modeling (p. 3300) (9 notes)
- Immediate-mode GUIs (p. 3515) (8 notes)
- Gradients (p. 3481) (8 notes)
- Anytime algorithms (p. 3319) (7 notes)
- Image approximation (p. 3514) (5 notes)
- Ray tracing
- POVRay
- 3d

Electrodeposition 3d printing

Kragen Javier Sitaker, 2016-02-19 (4 minutes)

Suppose you have a matrix of 2×2 copper electrodes per millimeter. So if you have a 100×100 millimeter area, you might have 200×200 electrodes, a total of 40,000. To each you have attached a transistor. On top of this matrix, you have a bath of electrolyte containing, for example, copper salts. If you lay a sheet of metal atop this bath of electrolyte at a depth of less than half a millimeter or so, you can selectively copper-plate certain parts of the sheet, printing a copper pattern on the sheet, by passing current through some of the electrodes and not others.

You can renew the electrodes to some extent by taking the workpiece out, replacing it with a sacrificial copper sheet, and reversing the current.

With this approach, you could conceivably electrodeposit one layer after another of copper until you have generated whatever shape you desire, gradually lifting the workpiece out of the bath. However, copper plating is a slow process, with deposition speeds of tens of microns per hour. The very short distance traveled by the current in this approach might allow the process to run a little faster.

This suggests that this approach might be more practical at a smaller scale: if your deposition rate is 10 microns per hour, that's about 2.8 nm per second. So you could reasonably space your electrodes 500 microns apart and electrodeposit your copper in 500-nm layers, one every three minutes. If you are doing this over a 10mm×10mm square chip, each layer contains 400 million voxels, or about 2 million per second, which is an eminently feasible data rate. Filling a cube in this way would require 1000 hours (plus the time to refresh the electrodes), so you probably want to make thin things instead.

The ability to deposit multiple different materials, such as copper and zinc, or copper and nickel, or gold and nickel, or nickel and tin, or copper and chromium, or lead and chromium, seems like it would dramatically extend the reach of this technique, allowing the construction of metamaterials with vanishing thermal coefficients of expansion, high-current-density batteries, or bimetallic structures that deform in predetermined ways according to temperature, for example.

Material cost is likely to not be an issue, except for extremely exotic materials and maybe gold. A milliliter of gold weighs about 0.6 troy ounces, or about US$1000; above I estimated that the device would need 1000 hours to deposit it, using about US$1 of gold per hour, but probably costing significantly more than US$1 per hour in labor to operate. Other normal metals cost orders of magnitudes less. Extremely exotic materials might include things like pure lanthanides (as opposed to Mischmetall) and isotopically pure metals.

Using a molten-salt electrolyte like the Hall-Héroult cell, rather than an ionic-solution electrolyte, might allow electrodeposition that is more favorable in one or another way: for example, you could electrodeposit aluminum and other metals that react too easily with water.

Of course, the ability to selectively electrodeposit controllably doped semiconductors in this way would be immensely valuable, but I don't know of any semiconductors that would be viable candidates by themselves. It might be possible, though, to selectively electrodeposit a doped metal, then oxidize the metal into a semiconductor by exposing it to an oxidizer, maybe at high temperature. For example, you could oxidize zinc to zinc oxide or zinc sulfide, lead to lead sulfide (galena), copper to copper oxide, or titanium to titanium dioxide.

Topics

Compressing REST transactions with per-connection state

Kragen Javier Sitaker, 2018-04-27 (11 minutes)

Generally speaking, the REST constraints that requests and responses be fully self-describing and that servers be stateless makes REST protocols fairly bandwidth-intensive, not suitable for low-bandwidth links (in the range of 0.001–10000 bits per second). This is actually cited as one of the drawbacks of REST in Fielding's dissertation. But there is an approach fully within the spirit of REST that can fix this.

Inspirational examples from HTTP

A web server wants a web browser to display a page with some inline images. So it sends the browser some HTML containing the URLs of the images. The HTML is interpreted in the context of that connection, allowing the server to use short "relative URLs" to refer to the images. If the browser already has the images, it can avoid fetching them again (according to a caching policy which may have many factors, including commitments made by the server about image changes and demands made by the user about freshness) and thus use very little bandwidth. Or, if the connection is using HTTP/2 or SPDY, the server has the option to "push" the images to the browser before they are requested, if it thinks the bandwidth cost is worth the potential improvement in latency.

A web browser wants eBay's web server to add a product image to an image a user is listing on eBay. Rather than sending the image data, it sends an URL to an imgur image, thus using very little bandwidth.

The generalization of the inspirational examples: including by reference, context-sensitive abbreviation, caching

These are two examples of a more general mechanism for enabling the REST architecture to work well even for applications with very stringent bandwidth constraints. Specifically, any large chunk of data is made into a separate resource and *included by reference* rather than inlined; the references are *abbreviated in a context-sensitive way* according to the context of the connection; and the chunks of data are then *cached* so that they need not be fetched every time.

You could argue that allowing the client to use context-sensitive abbreviations in its request amounts to requiring the server to maintain session state, a violation of REST. But as long as this abbreviation layer is a layer below the requests and responses, it affects the protocol semantics no more than the RFC3749 data compression that used to be a feature of TLS before the CRIME and BREACH attacks were discovered.

The crucial properties are that any request message that would be valid on one connection at some time is also valid on a newly opened connection at that time, even if its on-the-wire representation might

need to use more bytes, that the amount of abbreviation state is small, and that the client knows the entire session state to be able to tell the server in a new conneciton if necessary. This largely preserves the advantages of the client-stateless-server style REST derives from: visibility is preserved because a monitoring system can easily maintain the abbreviation state, partial failures (of a server) are no more difficult to recover from, and the server is free to close connections to free up resources whenever desired.

A database query example, starting with HTTP

For example, suppose you want to run a query against a large database and fetch some of the results. You could send a request like this (newlines added for clarity):

```
GET http://elephant-server/walrusdb/v18032
    ?q=select+*+from+walruses+where[2000 bytes omitted]order+by+size
    &p=1&n=10
```

to fetch 10-item page 1 of the results of a 2000-character query, evaluated on immutable snapshot 18032 of some walrus database. This is a workable interface, but inefficient, because every page of results is going to involve a 2100-byte request. Suppose instead that you could say this:

```
GET http://elephant-server/walrusdb/v18032
    ?q=https://pastebin.com/raw/FhnZFrGN
    &p=1&n=10
```

This gets us down to 90 bytes, a 23-fold improvement. If the server doesn't have a valid cached version of the query, it can go fetch it from Pastebin and cache it. (Pastebin marks it as cacheable for 1801 seconds.) What's more, the server can cache the query plan and even query results as long as that cache item is valid — and the cache key is only 34 bytes.

If at some point the server decides to evict the query from its cache, this is transparent to the client.

Departing a bit from HTTP

But we can do better. First, note that in real, non-proxied HTTP/1.0, we didn't actually say

```
GET http://elephant-server/wa...
```

because we allowed the connection context to determine the scheme and host/port parts of the URL. Instead, we said

```
GET /wa...
```

substantially abbreviating the URL. Later on, because of the IPv4 address shortage, we decided that it was better to put it back in, and so we ended up with this as a backwards-compatible hack:

```
GET /wa...
```

```
Host: elephant-server
```

And, later still, TLS gained SNI, which would in theory allow us to take the "Host:" header back out. In fact, maybe HTTP/2 did. I don't know.

Anyway, suppose that we use some kind of escape sequence to represent base URLs, and that each request includes the base URL it's for, and that there's a separate kind of request line that defines such an escape sequence. If we use "$" for clarity, rather than the perhaps more realistic choice of some kind of unprintable character, we end up with this:

```
$s=http://elephant-server/walrusdb/v18032
$p=https://pastebin.com/raw/
get $s?q=$p/FhnZFrGN&p=1&n=10
```

So now, at the cost of 71 extra bytes at connection establishment time, each of our requests is down to 30 bytes.

Involving Pastebin is kind of shitty, though. Your queries are public, Pastebin can alter them as they wish, they are identified by 8-letter strings, and it may take you a long time to upload them there even if the server doesn't use them. It would be much better if the client of the query processor could simply be the server for when the query processor wants to fetch the query; then it could allocate resource identifiers as it sees fit, reducing them down to perhaps two letters or digits until it gets over about 3700 of them.

You could do that with connection-specific resource identifiers, but that's kind of suboptimal because then a connection loss and reconnection invalidates the server's cache. A better approach is for the client to generate a session signing key and sign the responses with it. Then it only needs an abbreviation for its public key or a hash thereof. For example:

```
$s=http://elephant-server/walrusdb/v18032
$p=dat://12c09257a5f9320fb00b769bf6b68b17fee5c33a1b6c89b723941a06e7c7b19e/
ihave $p
get $s?q=$p/2q&p=1&n=10
```

Now we have 126 bytes of setup, but we're down to 24 bytes per request, 20% less, with no external servers or non-end-to-end encryption involved, without losing connection-to-connection cacheability.

24 bytes per request is little enough that it would be usable over a 300-baud modem — although the response might be unpleasantly large, depending on how the response is structured.

Attempts at tighter encoding: CoAP (fail)

You could try to do better than 24 bytes. For example, CoAP tries to provide a tighter encoding of HTTP-like semantics, but without requiring a reliable connection-oriented protocol underneath. It isn't successful in this case:

```
>>> import aiocoap
>>> req = aiocoap.Message(code=aiocoap.GET, mtype=aiocoap.CON, mid=1)
```

```
>>> req.opt.uri_path = ('$s',)
>>> req.opt.uri_query = ('q=$p/2q', 'p=1', 'n=10')
>>> req.encode()
b'@\x01\x00\x01\xb2$sGq=$p/2q\x03p=1\x04n=10'
>>> len(_)
24
```

What it buys us is that the message-id and the request for confirmation are bundled into the first four bytes along with the GET. (GET itself is represented by the second byte being 0x01; 0x02 is POST, 0x03 PUT, and 0x04 DELETE; 0x45 'E' is the most common equivalent of HTTP 200 OK, while for example 0x84 is 4.04 Not Found.)

Attempts at tighter encoding: ncompress and gzip (these help)

ncompress implements LZW and has very little overhead and almost no codec latency. The following file contains 10 requests and, without compression, is 367 bytes, including 126 bytes of initial setup.

```
$s=http://elephant-server/walrusdb/v18032
$p=dat://12c09257a5f9320fb00b769bf6b68b17fee5c33a1b6c89b723941a06e7c7b19e/
ihave $p
get $s?q=$p/2q&p=1&n=10
get $s?q=$p/2q&p=2&n=10
get $s?q=$p/2q&p=3&n=10
get $s?q=$p/2q&p=4&n=10
get $s?q=$p/2q&p=5&n=10
get $s?q=$p/2q&p=6&n=10
get $s?q=$p/2q&p=7&n=10
get $s?q=$p/2q&p=8&n=10
get $s?q=$p/2q&p=9&n=10
get $s?q=$p/2q&p=10&n=10
```

This works out to 36.7 bytes per request. ncompress reduces it to 241 bytes, or 24 bytes per request; the initial setup was inflated to 135 bytes, but the marginal cost per request was then just 10.6 bytes.

By comparison, if we rely only on ncompress with no other abbreviations, we could try this:

```
ihave dat://12c09257a5f9320fb00b769bf6b68b17fee5c33a1b6c89b723941a06e7c7b19e/

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=1&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=2&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=3&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=4&n=10
```

```
get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=5&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=6&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=7&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=8&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=9&n=10

get http://elephant-server/walrusdb/v18032?q=dat://12c09257a5f9320fb00b769bf6b68b
17fee5c33a1b6c89b723941a06e7c7b19e/2q&p=10&n=10
```

Uncompressed, this is 1359 bytes. Compressed with ncompress, it is 626 bytes. We can conclude that ncompress is not an adequate replacement for the abbreviation system.

What about LZ77? With gzip -9, this same file compresses to 168 bytes, 16.8 bytes per request, the best showing so far. However, that's relying somewhat on a certain amount of cross-request compression. Flushing the compressor after every line brings the compressed size up to 266 bytes — 26.6 bytes per request. We can conclude that gzip is a nearly identically performing replacement for the abbreviation system.

Why not both? gzip -9 reduces the abbreviated 367-byte version we started with to 171 bytes, or 17.1 bytes per request; 127 bytes of this is the initial setup, leaving only 4.4 incremental bytes per request thereafter. However, if we flush every line, it's even worse than before — up to 283 bytes.

Clients composing things from pieces on the server or out in the world

Suppose the server you're talking to supports constructing some kind of filesystem environment to run programs in. You tell it what the filesystem looks like (with a list of Docker-like layers or a Git-style tree hash or whatever) and it downloads whatever files it's missing.

Suppose the files are mapped to resources on the network. Now you can build your filesystem environment from references to files that are located on the server itself, if it happens to also be a fileserver, but also on other servers near it, as well as files you yourself have. And you can do all of this without making the server responsible for maintaining that state in any kind of long term. You just need some kind of naming system for the resources.

Unifying abbreviations with resources

Topics

Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing

Kragen Javier Sitaker, 2019-01-25 (7 minutes)
(Originally posted at https://write.as/s7f1ywj0jh735.md)

The standard API designs for graphics APIs are "immediate mode" and "retained mode". In immediate mode, you emit a series of drawing operations, which typically change pixels, and that's how you make your graphic. This means you can emit more drawing operations than there are pixels in the canvas, and in some cases you can do animations by sending a series of operations. Memory usage and performance are predictable, but not necessarily very good. `<canvas>`, PostScript, GDI, and Xlib are immediate-mode APIs.

Retained-mode APIs, by contrast, maintain a set of graphical objects in memory, and the API lets you create and modify those objects. Often they're arranged into a tree structure by containment. Typically they are a lot more of a pain to use and use a lot more memory. The SVG DOM is a retained-mode API.

Immediate-mode APIs, although they're often easier to use, have a few drawbacks:

• It's a little bit tricky to make antialiasing work.
• You can't zoom. Or, rather, zooming involves redrawing. And that's slow.
• The drawn objects can't be clickable because they don't have persistent identities.

This bit about redrawing, though, that's interesting. Presumably the entire structure of your immediate-mode graphic somehow inheres in your code combined with the memory state it is interpreting. As long as you can re-execute the code deterministically, you can redraw or inspect whatever other behavior of the code you would like to inspect.

Memory transactions and caching

Optimistically-synchronized transactional-memory systems also rely on the ability of bits of code to re-execute deterministically, as does Umut Acar's Self-Adjusting Computation. Optimistic TMs will roll back a transaction when another transaction has written to a shared variable that it has read, retrying it from the beginning with the new value, potentially as late as when it attempts to commit. (Some TMs also allow writes to leak out during transaction execution; in these cases, if another transaction reads those writes, it must be rolled back and retried if the writes are rolled back. This can lead to livelock.)

The basic idea is that, if you can track all the bits of memory that the code reads from outside of its ephemeral internal state, you can be sure that the code would produce the same results if it were executed

again; and if you control its outputs, you can undo those results if necessary. This allows you to confidently cache the results of running it, among other things.

What if we did this with immediate-mode drawing? If we divide our drawing into a series of possibly nested transactions, the transaction system can provide the advantages of a retained-mode API potentially without all the expense —– the transaction system can intelligently trade off memory space against the possible inefficiency of having to re-execute. It can, for example, compute a bounding box for each transaction, and then, when zooming, only re-execute transactions that impinge on the visible screen area. (You could also offer an a-priori bounding-box function which tells you whether any of a given bounding box is visible, so as to avoid executing things inside of it, but this conflicts with the transaction system's necessity for the viewport not to affect the drawing; a reasonable compromise is to offer a "clip" function which limits the visibility of further drawing to a given bounding box.)

Also, of course, being able to confidently re-execute an immediate-mode drawing enables us to detect clicks, if that's a thing we want to do.

Data serialization

We can apply an analogous approach to serializing and deserializing data structures, with the highly desirable benefits of being able to orthogonally cache serializations and writing the serialization and deserialization code as one routine. An IMGUI library like Dear ImGui might establish a mapping between a character buffer and a widget on every drawn frame:

```
ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));
```

We might analogously establish a mapping between a character buffer and a chunk of the input/output stream:

```
int len = strlen(buf);
little_endian_32(&len);
bytes(buf, len);
```

When serializing, `little_endian_32` will serialize `len` to the stream (as a little-endian 32-bit binary number, I suppose); when deserializing, it will overwrite its current value with the value deserialized from the stream. Then `bytes` correspondingly reads or writes the given number of bytes.

We could imagine running the above in a transactional context that tracks its reads and the output bytes and is thus able to cache the output bytes and not re-execute the code when the inputs haven't changed.

Backtracking

A transaction system is perfectly entitled to checkpoint partially-executed transactions so that it can restart them from a checkpoint, rather than from the beginning, if it needs to retry them. This may be more expensive (it needs to save the entire ephemeral state of the transaction, not just the program counter and inputs) but

at some point it may be less expensive. This corresponds to chronological backtracking in AI search — if you checkpoint the transaction before returning the results of each nondeterministic choice, perhaps implemented as a read of a transactional variable, you can run a chronological-backtracking search over its execution space, searching for a set of nondeterministic choices that yields a successful execution.

We can do better, though; with nested transactions, we can do non-chronological backtracking as well. Normally the failure of a nested transaction will result in the failure of the outer transaction as well, but the nested transaction may depend on far less input data than the outer transaction. The inputs to the nested transaction (its arguments and the set of nondeterministic choices made within it) comprise, for the purposes of backtracking, a nogood set. This should allow some degree of non-chronological backtracking, although it doesn't allow the transaction system to direct the search procedure to nondeterministic choices with fewer remaining alternatives.

In the context of serialization and deserialization, chronological backtracking amounts to recursive-descent parsing, and the transaction system is capable of the kind of memoization that allows Packrat parsers to guarantee linear-time parsing. I'm not sure if it's possible to derive Earley parsing from recursive-descent parsing in this way.

Reducing working sets

Although all of the above is written from a big-computer perspective, the thing that originally attracted me to immediate-mode GUIs was Arduino menu systems. I hacked together an Arduino menu system that uses only a few bytes of RAM by using ImGui principles — from one call to the next, it only tracked the user's position in the menu tree, but the menu tree itself was generated on demand by callback code, which of course was not stored in RAM.

Topics

Subterranean glazing

Kragen Javier Sitaker, 2016-09-06 (25 minutes)

For some time I've dreamed of towns where all the building is underground, and the surface is devoted to soil — gardens and parks — but with some exposed windows to the sky. Subterranean construction faces a number of objections, one of which is a concern that it will be dim and dark, due to a lack of natural light.

We can achieve natural lighting conditions in almost entirely underground construction, with populations at low-urban densities, despite agriculture sustainability.

What it looks like

You're standing on a grassy, gently rolling hill, in the shade of a tree-sized sculpture, sort of shaped like a rotifer or sea lily, with a cap some five meters across and eight meters tall atop a three-meter-tall stalk that must be metal to support that much weight, although it looks like sandstone. The stalk is some 1.5m across, and the sandstone is warm to the touch if you put your arms around it. The cap is not vertical, and the stalk is not straight; it bends noticeably towards the southwest.

A flagstone path winds its way down the hill; a bit further down, it runs between some bamboo boxes containing raised-bed gardens, some filled with a mix of leafy deep green plants, leafy purple plants, and flowers, and tall stalks of corn, while others contain clearly recognizable onions mixed with staked-up tomatoes and eggplants. No other buildings are in sight, just gardens, hills, forests, and waterways, and a few other organically shaped sculptures.

You walk down the sunny flagstone path between the gardens to a narrow brook, only about a meter wide but flowing slowly along its meander. A stepping stone beckons you across, but you follow the flagstones along its bank. A mix of trees lines its other bank, none more than about three meters tall, because this is a new forest; it's dominated by fast-growing pines that are culled once slower-growing fruit trees have been nurtured in their shade. A clump of four-meter-tall bamboo, some culms three centimeters across, is visible a bit upstream.

A few meters downstream, the brook splashes down a little cataract into a pond, some seven meters across. Through the water of the pond, you can see the pond bottom clearly. Among its algae, you can see a number of rocks on the bottom; although many are opaque, as one normally expects, some of them are round and shining, like ovoids of dark glass. By the side of the pond is a weeping willow. Trails of bubbles are coming up on part of the pond's surface.

Another grassy hill is next to the pond; over its crest is visible a sort of wooden arch, standing alone, with what appears to be a large wooden snailshell atop it. The flagstone path curves around the hill. Rather than accepting the pond's invitation to swim, you follow it past a pink granite boulder protruding from the side of the hill. There is a three-meter-wide circular door in the boulder. You touch it and enter, ducking slightly and closing the ponderous door behind you with little effort, due to its high-precision bearings. It seals tightly.

You are in a three-meter-wide corridor with an arched ceiling, sloping down into the hill and beyond. Although it's not as blinding as the sunny meadow outside, it's brightly illuminated from all sides; the walls have the appearance of stained-glass windows, a brilliant glow from within illuminating the deep blues and reds that form the images on their surface.

The air in the corridor smells fresh, despite the apparent hermetic seal on the door and lack of noticeable draft inside.

You walk down the mild slope of the corridor, past a door labeled STAIRS near the entrance, past some 24 meters of the luminous murals, around a bit of a curve, past some benches, to a grouping of three large doors at the end of the corridor, two made from laminated bamboo and one apparently a dark-stained hardwood. The top of your head is now a bit lower than the threshold of the door through which you entered. You knock on the door of your host.

They open the door to welcome you. You enter, noticing that the door is quite a bit heavier than you would expect a laminated bamboo door to be.

The apartment is nearly circular, some 11 meters across, and as brightly illuminated and fresh-smelling as the corridor. It is nearly silent; the machine noises we often hear in conventional housing, from refrigerators, computers, air conditioners, etc., are completely absent. Also, there are no cars outside, and even if there were, the noise would not reach you underground. You hear only the sounds of your friend and yourself.

The walls here are white plaster, smoothly curving into the ceiling, rather than the luminous stained-glass look of the entry corridor. The upper parts of the walls and ceilings are illuminated by what seems to be sunlight, shining up at it from tiny alcoves in the walls. The ceilings are only three meters high, no higher than those of the corridor, because you are barely below ground at this point.

A 2-meter-wide round hole in the floor of the four-meter-across round room in the center of the apartment has a bamboo railing around it, providing a view to the four-meter-tall lower story. An archway on the outer edge of the apartment, near the door to the corridor, leads to a gently sloping two-meter-wide helical ramp curving around the perimeter of the apartment; after almost one and a half turns, this reaches the floor below, rendering the entire apartment wheelchair- accessible without elevators.

The total floor area of the apartment, not counting this outer helical ramp, is some 190m² (2045 sq ft); it is divided into ten rooms of different sizes, with different ceiling heights. The central downstairs room has a two-meter-wide hole in its four-meter-high ceiling leading to the three-meter-tall upstairs, for a total of over seven meters; while other parts of the downstairs have lofts for storage built over them, lowering the normally four-meter ceilings down to 2½ meters, and there are even a few alcoves where an adult cannot stand comfortably.

Different parts of the apartment are illuminated by different ways and to different degrees. Some parts have quartzite and agate protruding from the walls, brilliantly glowing from within; other parts have the ceilings flooded with sunlight; others have the stained-glass effect from the entry corridor.

There is another emergency-exit door labeled STAIRS in the

lower story of the apartment, and also another mysterious door, which turns out to link to an on-demand electric very-light-rail system allowing you to travel to other nearby houses even when it's snowing at speeds of 40 kph (11 m/s).

What "natural lighting" normally means in architecture

Brian Knight explains (confirmed by the Passive Solar Primer) that for passive winter solar heating of homes, you want 9–12% of the floor area of the house in south-facing windows, in the context of the 35° north latitude of Asheville, NC. "More than 12 percent puts the house at risk of overheating unless the design includes extra thermal mass," he explains. This tells us that we can get enough natural light to make people happy with our architecture with only about 10%.

But that's not 10% of the sunlight; that's 10% of the floor area in the form of vertical south-facing windows. At 35° latitude, you're only getting about sin(35°) ≈ 57% of that amount of sunlight, a bit more in summer, a lot less in winter. If you're getting the light from windows facing the heavens instead of standard horizontal windows, you only need about 5.7% of the roof space to gather the necessary light. And that illumination will be stabler throughout the year.

Photovoltaic-powered electrical lights are a bad idea

Now, one thing you could do would be to try to do it with photovoltaic solar panels, and run electrical lights from them. The problem with this is that normal solar panels are only 16% efficient, and the luminous efficiency of a regular lightbulb is only 2%, a quartz-halogen bulb can reach 3.5%, compact fluorescents and LED lamps are typically about 10%, while sunlight is 13.6% (because only 37% of its light is within the visible band, and much of that is far from our photopic sensitivity peak). That means that every lumen of sunlight can only convert to about 16% × 10% / 13.6% ≈ 0.12 lumens of artificially generated light indoors. So instead of 5.7% of the roof space, you'd need about 48% of the roof space for solar panels for illumination alone, not leaving much space for anything else.

Lightpipes are a good idea

[Light pipes] (or light tubes, or lightguides) are like fiber optics, but thicker. Big ones can be silvered on the inside, like a thermos, and filled with air; small ones can be solid or water-filled. The light suffers a 4% Fresnel loss upon entry and upon exit, plus some absorption on the way down. (This absorption can be advantageous if you want to reduce infrared and ultraviolet in order to increase the illumination-to-heat and illumination-to-sunburn ratio; but ultraviolet absorption by Biosphere 2's regular glass roof resulted in ecological problems within, so beware.)

Typical lightpipes can carry 70% to 90% of the sunlight fed into them for many meters. Optical concentration (imaging or nonimaging) can concentrate multiple square meters of sunlight into a lightpipe whose cross section is a fraction of a square meter.

So, supposing that you want to illuminate a 50m² subterranean

apartment to conventional "natural lighting" levels. You can gather sunlight from a 50m² × 5.7% ÷ 80% ≈ 3.6m² area on the surface, stuff it through a 20cm-diameter 80% efficient lightpipe to carry it down to your apartment (leaving the infrared and ultraviolet up on the surface), and let the other 46.4m² of sunlight nourish your garden and park. Perhaps you can put the solar collector a couple of meters above the ground as a sort of artificial shade tree — the rotifer and snailshell sculptures mentioned in the introduction.

Needing 7.2% of your surface area (per underground story) is much better than needing 48% of it.

Illuminating the 190m² subterranean apartment in the illustration to this level will require shading almost 14m² at the surface. The rotifer sculpture mentioned at the beginning would have a mouth area of up to 19.6m²; the seven-meter-wide pond totals about 38.5m², and if the glass stones on its bottom cover 36% of the bottom, they would collect a similar amount of sunlight.

Nonimaging optics are limited to a concentration factor of $C_{max} = n^2/\sin^2\theta$, where n is the index of refraction at the absorbing aperture (1 if you're doing it just via reflection) and θ is the maximum angle from the optical axis at which you're gathering the light (the half-angle of the radiation cone apex). The rotifer sculpture might have a concentrating parabolic reflector of 2.2m radius at the mouth, concentrating down to a lightpipe throat of 75cm radius inside its sandstone-finish stem; that's a concentration factor of 8.6, which limits it to collecting sunlight from no more than 20° off-axis, or a total of a 40° arc through the sky, unless you turn the rotifer to track the sun a bit, or feed multiple sculptures into the house, each covering part of the sky. Note that the tropics are almost 47° apart, so you'll probably have to reorient the rotifer seasonally at least.

Note that if you can track the sun perfectly, the theoretical maximum without refraction is $\approx 1/\sin^2(0.54°)$, or about 11000 — about 11MW/m² or 1100W/cm².

You can trade off azimuthal field of view for elevational field of view for a given concentration, though; in the extreme of an east-west trough concentrating reflector, you could get the same 8.6-sun concentration by orienting the trough within 6.68° (or 6.68° - 0.54°, really) of coplanar to the plane of the sun's nearly-planar apparent motion.

Agricultural sustainability

The apartment described in our introductory anecdote occupies some 133m² just below the surface, although it has considerably more floor area than that because of its two stories. It could host a fairly large number of people; it's exceedingly spacious for just one or two people. It might become a bit of a pain to clean, in fact.

But the smallest number I've been able to convince myself of, for agricultural sustainability, is 50m² of cultivated area per person, using soybeans, dwarf corn, and ample soil amendments. In colder climates, even approaching this level of productivity requires greenhouses and similarly extreme measures. And if only some of the surface is gardened, in order to leave some of it for forests, brooks, and light collection, you need more space. And lower-labor agricultural systems like apple orchards are also lower productivity per hectare. (Mature tall-spindle apple orchards can yield 1000 bushels per acre per

year; at 40 pounds per bushel and 52 kcal per 100g that's 2331 kcal/m²/year (310mW/m²!) which is about ⅐ of the raw productivity of high-yielding conventional agricultural systems like corn. Supposedly netting and Tatura training can close most of the gap, but I'm skeptical, and anyway that moves the apples back into the high-maintenance intensively-cultivated category.)

Anyway, let's say you need 100m² per person for gardening. Then 150m² per person gives you some space for meadows and forests and whatnot, too. That's 6700 people per square kilometer; this compares reasonably to many current cities, such as Dalian at 7100, Rio de Janeiro at 6850, Bangkok at 6450, London at 5100, Athens at 5400, the Buenos Aires metropolitan area at 4950 (Buenos Aires proper is 14000), Moscow at 4900, Berlin at 3750, Accra at 3300, Quito at 3150, and Los Angeles at 2750.

Emergency escape

The plan described in the introductory anecdote features a number of heavy doors that close tightly. The door to the outdoors is like this in order to facilitate indoor climate control, but the others are that way for fire-escape reasons: you need to be able to reach a fire escape that's sealed off with a fire door without having to travel too far. So the door that appears to be laminated bamboo is actually a steel fire-door core with laminated-bamboo surfaces, so that fires or chemical releases inside one apartment don't render the escape route for the others unviable.

Medieval fortress defensibility scale

I ran into a fascinating discussion of medieval sieges the other day; one of the discussants said:

> There is usually a balance that prevents long long term sieges...a large population is capable of fully defending the walls and keeping the invaders out, but consume more resources and shorten the time that they can hide behind walls. A smaller force will consume less resources and hold out for longer, but they risk not being able to defend the walls fully due to lack of man power.

But of course a sufficiently large fortress doesn't suffer from this problem, because its ratio of arable land to walls becomes arbitrarily large; and this happens sooner if productivity per acre is high. Leaving aside motte-and-bailey-type fortresses where the food is grown *outside* the fortress in the bailey, what's the scale at which the lifestyle outlined in this scenario would become able to defend a hypothetical wall?

Suppose we need one defender per two meters of wall. Then a circle 300 meters across would have 942 m of wall around it, and also 471 × 150m² of land inside of it, and therefore be able to support 471 inhabitants, sufficient to defend that wall. 309 meters across would support 500 inhabitants.

This is a size sufficiently larger than Dunbar's number that you would probably need some kind of official power structure within it.

Clustering

In the example scenario, you have three apartments that access the surface through the same entry tunnel. In practice, you might want to cluster houses together more than that; there are great advantages to being able to borrow a hammer or some yeast from your neighbor

without having to walk 100 meters, and in particular there are advantages to serendipitous meeting and chit-chat with neighbors. If you cluster groups of ten such apartments together, and figure two inhabitants per apartment, everybody's gardens are a little further off (2000m², for the 100m² of gardens for 20 people, excluding parks, has a radius of 25 meters), but they'll encounter each other much more often, and forests and ponds can be proportionally larger; and in the fortress case, you can go slightly motte-and-bailey and actually locate the parks for the outside the fortress walls, leaving only the arable land inside.

This allows you to reduce your 309-meter-across circle by about 30 meters: the 14 or so clusters around the outside shrink to 2000m² each, leaving the other 11 or so clusters in the middle with 3000m², for a total area of 61000m² and thus only 279 meters of diameter, or only 877 meters of outer wall.

The Subway

I said there was an on-demand light rail. Specifically, what I have in mind is kind of like a horizontal elevator; a small capsule comes when you call it, and then takes you where you want to go, and is computer-controlled so it won't hit other capsules and you don't have to pay attention to driving it. Suppose, as above, that you have clusters of ten apartments containing 20 people, and your overall community is 309 meters across and 500 people, spread over a total of 25 stations. Each station, except the first, needs to be connected to at least one other station, and the stations are about 60 meters apart, so you need at least 1440 meters of tube. Say 1800 to be on the safe side, allowing for sidings and redundant loops. At an average of 5m/s you need 12 seconds to get to the next station and 60 seconds to get all the way across town. And you might have to wait 15 seconds for a spare capsule to come pick you up.

(Note that this 1800 is 3.6 meters per inhabitant, i.e. tiny in cost.)

The tube can be quite narrow. Suppose the capsules are designed to accommodate up to six people, two abreast, or one abreast if it's a wheelchair; the cabin can be two meters wide, 1½ meters tall, and three meters long. The tube can be 2½ meters wide and 2 meters tall, so the 1800 meters of tube suggested above work out to 4500m², or 9m² or 18m³ per inhabitant.

Also, the capsules, rails, and power cables (or shafts, or whatever) can be fairly light. Six people weigh 600kg or less, normally; accelerating 600kg to 11m/s over 30m takes about 6 seconds, which means it's about 1m/s² (a comfortable 0.1 gees), 600N, with a peak power of 6600W or a bit under 9 horsepower. Amazon sells a 1½-horsepower motor for pool water pumps for US$250 (the Hayward SPX2710Z1M); it weighs 27.7 pounds (12.6kg), so we can figure that a durable 9 horsepower would weigh 75kg and cost about US$1500. (Starter motors can be similar in power and weigh much less, but pool water pump motors are sold to run for hours continuously every day, not for a second or two a few times a month.)

So the whole capsule might weigh 800kg and need 12 horsepower of motors (to hit its target speed. That's ten kilowatts of electricity, which would be 21 amps at 480 volts — very easy to supply with no significant loss with a third rail. Given the short distances and the

availability of rails, you could maybe even use 48 volts at 210 amps, eliminating the risk of electrical shock, although you could still get creamed by a capsule on its way through the station.

A single capsule would need 84 round trips (168 minutes) to transport the entire population from one extreme of the town to the other, so you might need some ten capsules in total, one for every 50 people; this would allow 12% of the population to be in transit at any given time, if necessary. That's a total of US$30 worth of motor per person, and maybe another US$30 or US$100 worth of capsule construction.

I don't know how heavy train rails need to be to support a 800-kilogram mini-train. Standard axle loads are 19.3 tonnes per axle running on rails that weigh about 44.6 kg/m; here we're talking about 0.4 tonnes per axle, which (with linear extrapolation) would work on 0.92 kg/m rail, which is like a fifteenth of normal "light rail" rail. That would be a total of about 1700kg of steel for the system as a whole, or 3.4kg per inhabitant.

(In fact, you could go even chintzier: mountain bike tires are up to 57mm wide and can reasonably have a 15cm contact patch at 50 psi, so they can support up to 300kg each. Four wheels with mountain bike tires would work.)

If you were to scale a system like this one up, its 11m/s top speed would allow you to cross Buenos Aires, which is about 18km across at its widest point, in 27 minutes, plus half a minute to walk 30m to the nearest station and 10 seconds for a capsule to arrive. If you could double the top speed to 22m/s (79 km/h, 49mph; this would not require bigger motors, only accelerating for four times as long) then you could cut it to 14 minutes. Currently crossing Buenos Aires from Puente Saavedra to Villa Riachuelo in the bus takes 85 minutes on the 28 or 21 bus, which travel along the freeway. If you take the 76 and 150 through the city instead, it's 109 minutes. And often you have to walk for hundreds of meters to reach the bus stop and wait there for tens of minutes.

Given the advantages of this kind of transportation system, which would cost US$95 of vehicle, 3.4kg of steel rail, and 18m³ of tunnel excavation per inhabitant even at the low urban density of 6700/km², As urban distances go down, the cost of rail and tunnel should go down proportionally. So having built such a system in Buenos Aires, with its 14000/km² population, would have been enormously useful.

Gordon Mohr suggests that you could recharge at subway stations instead of having a third rail. 60 seconds at 3kW is only 180kJ; at the 18kJ/kg of current supercapacitors, you'd only need 10kg or so of them. If you could recharge as you zipped through each station, you'd only need 6 seconds, or 18kJ. Amazon will sell you 2.7V 10F capacitors for US$2.84; they can store 36 J, so 18 J of capacity would cost you US$1420, comparable to the motor.

Topics

The continuous-web press and the continuous press of the World-Wide Web

Kragen Javier Sitaker, 2017-03-20 (6 minutes)

I saw a continuous-web press in operation for the first time last night; by chance, I walked by the door of a newspaper at 2:30 in the morning and saw the morning edition being printed, cut, folded, and baled. And I was reminded about the twentieth-century saying, "I never quarrel with a man who buys ink by the barrel," originated by Charles Brownson, a Congressman from Indiana. I pondered this as I watched the awesome spectacle of one of the most powerful weapons of the twentieth century in full operation: a mass duplicator of information, the weapon that centralized control over public opinion in dictatorships and democracies alike.

It occurred to me to estimate the bit rate of this duplicator. A crude estimate of its speed would be about 2 meters per second, printing on a web that's roughly a meter wide. A similar newspaper we have here in the house has twelve columns across a two-page spread, perhaps 50mm from one column to the next, 3.8 mm per line. Here are ten representative lines of text from one such column:

nomía nos trajeron hasta esta
decadencia; las correcciones son
dolorosas y sufridas, y encima
Cambiemos suele prescindir del
vijo axioma de Raúl Alfonsín:
"Hacer política es hacer docen-
cia." El 70% de la población sabía
que marchábamos hacia Vene-
zuela y que la luz no podía costar
lo mismo que un café americano,

This is 317 characters of text, including newlines, in a 50mm × 38mm area, about 170,000 characters per square meter. If we figure that each character of text is 3 bits, which is about right for zipped text, this is 510,000 bits per square meter. This means that two square meters per second is roughly 1.02 megabits per second.

(We should perhaps correct somewhat for the fact that some of the paper consists of colors, even full-color photos.)

So, in the mid-twentieth-century world, a one-megabit-per-second web printing press could give you near-dominion over a small town, or a substantial position in the affairs of a large city or a nation; it would make a US Congressman fear you. Even today such an asset grants substantial influence, both because of its signaling value and because it allows its owner to reach the rapidly-diminishing population of people who still don't have computers.

But the US just elected Donald Trump as President in spite of every newspaper in the country endorsing Hillary Clinton. It seems that the apparatus of manufacturing consent is no longer effective. And I think that we can explain this largely in terms of these bit rates.

A web site in Macedonia can easily sustain a megabit per second, and any cellphone on the planet can read it.

To put this in a concrete current economic context, DigitalOcean currently offers a virtual private server (VPS) with 512MB of memory, a one-core processor, a 20GB SSD disk, and a terabyte of data transferred for US$5 per month or US$0.007 per hour (US$5.11 per month). A terabyte per month is three megabits per second — average, not peak. So DigitalOcean's smallest server is probably substantially higher bandwidth than the continuous-web press I saw.

Moreover, publishing something on a web site only requires you to send it to the people who want to read it, while publishing it in the newspaper requires you to make one copy per copy of the newspaper. So those megabits go a lot farther.

Of course, newspapers also offered anyone the opportunity to publish advertisements costed per page, although the cost was higher. The ability to have your message printed on the high-speed press wasn't what gave you power; it was your ownership, which allowed you to choose which message to print.

Today, a Raspberry Pi 3 costs AR$930 here in Argentina, which is US$58. It has a 100-megabit-per-second Ethernet port, which is probably a bit faster than you can get information out of it in practice; Jeremy Morgan found nginx was the fastest option, managing about 3900 transmissions of a 95,881-byte HTML file in two minutes, which is about 25 megabits per second. (The Monkey httpd was a bit worse, and lighttpd and Apache were several times worse in his high-concurrency test.)

So the capital cost of a machine to emit 25 megabits per second is about US$60, a bit over US$2 per high-speed continuous-web press equivalent, or maybe US$0.10 if you try to figure in how much of the newspaper each person reads. If you were duplicating data onto SD cards or something instead, you would probably get higher bandwidth.

By making the press itself abundant, we have not eliminated inequalities in power; we have merely shifted the bottleneck and corresponding power to elsewhere in the social system. The natural expectation might be that it would move to the organizations that control the physical infrastructure needed to distribute the copied information, and that could indeed happen — witness the failure of USA internet companies to penetrate into China due to its lack of net neutrality favoring domestic firms — but it seems that what more commonly happens is that power shifts to the institutions that have accumulated the data and interpersonal relationships to make the most addictive uses of these network links.

Topics

- Performance (p. 3621) (149 notes)
- Pricing (p. 3646) (89 notes)
- History (p. 3500) (71 notes)
- Politics (p. 3639) (39 notes)
- Economics (p. 3424) (33 notes)

Digital noise generators

Kragen Javier Sitaker, 2018-10-28 (2 minutes)

The traditional approach to generating noise with a digital circuit is to use a maximal-length LFSR, which has the great advantage of using only a single bit operation per generated bit, although XOR is perhaps a somewhat more complicated bit operation than NAND or abjunction.

A crucial feature of the LFSR is that its state transitions are reversible; they do not lose information. Consequently its cycles form a partition of the state space; every state is the successor of some state, by a counting argument. In maximal-length LFSRs, there are two cycles: one of period 1, containing just the zero state, and one containing all the other states.

When thinking about generating digital noise on CPUs, it is interesting to think about other available reversible operations. For example:

- Rotating left or right by a constant number of bits.
- Adding or XORing a constant.
- Multiplying by an odd number.

- As a special case, 2's-complement negation.
- Permuting the bits or bytes of the state vector with a constant permutation.
- XORing any part of the state vector with the state vector, as long as no bits XOR with themselves, which is irreversible.
- Multiplying by a constant m modulo some other relatively prime constant n, iff the current state is less than n.
- Running an arbitrary substitution on the bit vector via a constant lookup table.

These are, as it turns out, the usual building blocks of symmetric cryptosystems and hash functions.

What if we investigate the periods and spectra of the functions comprised of different sequences of these operations? Perhaps we could find something that was fast to execute on a CPU, but also provided long-period white noise.

Such techniques are also useful for hashing to, for example, generate Perlin noise.

Topics

Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods

Kragen Javier Sitaker, 2019-08-17 (updated 2019-09-15) (15 minutes)

(Probably nothing in here is novel; it just documents my own exploration of the space of local search algorithms in vector spaces, which is well known, but not to me.)

When writing Rubber wheel pinch drive (p. 2912) I got confused with some basic trigonometry, trying to solve a triangle with the law of cosines† to find the intersection of two circles. So, impatient, I figured I'd solve the problem with brute force instead, so I wrote the following implementation of local search with random restarts for scalar functions of general vector spaces:

```
def search(f, p, restarts=30, steps=30):
    best = None
    for start in range(restarts):
        current = p()
        current_badness = f(current)

        for size in range(20):
            for i in range(steps):
                new = current + p() * 4**(4-size)
                new_badness = f(new)
                if new_badness < current_badness:
                    current, current_badness = new, new_badness

        if best is None or best[1] > current_badness:
            best = current, current_badness
```

Here `p` is a function that yields a vector from the domain of `f`, which returns a real.

At first this didn't work very well because I had erroneously provided a guess function `p` that was always non-negative, so each restart would progressively step from the neighborhood of 0 to the neighborhood of the correct solution and then get stuck. Nevertheless, it was able to find a reasonably good solution (and then I realized how to solve the problem in closed form without trigonometry).

† The law of cosines is $c^2 = a^2 + b^2 - 2ab \cos \gamma$, where γ is the angle opposite the side with length c.

Making local search in vector spaces more robust

Random restarts in general make metaheuristic search algorithms more robust; indeed, even the simplest possible metaheuristic search algorithm, "choose a random point", becomes workable with random

restarts.

In this implementation, the restarts aren't very random.

I tried to make the implementation somewhat robust against `p` functions of the wrong magnitude, as you can see; the algorithm tries a range of different exponentially-spaced sizes. Now that I'm using a non-broken `p`, this results in finding a solution correct to ten decimal places, which is pretty good for such a simple local search algorithm. But the particular orders of magnitude I chose are kind of arbitrary; basically I was just hoping they'd cover the right range.

It occurred to me that you can make it more robust against `p` guesses of the wrong order of magnitude by updating `size` incrementally, using a procedure like the following: when you find a step that improves the guess, try half that same step or twice that same step. If that improves the guess further, then multiply `size` by 2 or ½, respectively, before the next iteration. This way, if your step size is about right, `size` will experience a random walk around it, but if it's much too small (you're stepping around a locally flat linear region of the cost function) then `size` will grow exponentially, while if it's much too large (you're rocketing out of the basin with all the solutions in it) it will diminish exponentially.

By the same token, you could make it robust against a `p` function like the one I provided by trying a step of $-p$ when p makes the situation worse. That way, if you're in a relatively non-curved region, you can make progress by just shifting into reverse. Trying this actually gives you another crucial piece of information: if going both ways makes it worse, there's a good chance you're in a local optimum, using a step size that is too big to have a good chance of improving your position.

You could implement this as follows, though this code is a bit repetitive:

```
def climb(f, step):
    size, growth, here = 1, 2, step()
    cost = f(here)

    while True:
        yield here, cost  # This may be the same as last time

        where = size * step()
        new = here + where
        new_cost = f(new)
        if new_cost > cost:
            where = -1 * where
            new = here + where
            new_cost = f(new)

        if new_cost > cost:
            size /= 2
            continue

        where *= growth
        grown_new = here + where
        grown_new_cost = f(grown_new)
        if grown_new_cost < new_cost:
```

```
            new, new_cost = grown_new, grown_new_cost
            size *= growth
        else:
            growth = 1/growth

        here, cost = new, new_cost
```

This is able to find a good approximation of the intersection of two circles in "only" 128 iterations, using the following definitions, even when the answer is several orders of magnitude away from the starting step size:

```
def distance(p1, p2):
    return ((p1 - p2)**2).sum()**0.5

def circles(c1, r1, c2, r2):
    def badness(guess):
        return ((distance(guess, c1) - r1)**2 +
                (distance(guess, c2) - r2)**2)
    return badness

step = lambda: numpy.random.rand(2) - 0.5
```

Handling valleys (ridges), saddles, and higher-dimensional spaces

However, its adaptive step size will screw it in higher-dimensionality spaces, since when it's in a valley or saddle point where most directions are bad, it will tend to diminish the step size and find a point on the valley floor very precisely. Perhaps a better approach would be to diminish step sizes by a smaller amount that depends on the dimensionality, so that if there's at least one good direction, the step size will remain constant on average.

("Valleys" here are "ridges" in the usual hill-climbing metaphor, because here we're trying to *minimize* a cost function, while "hill climbing" is trying to *maximize* your altitude.)

To escape this trap, you could have different behavior when expanding the step size than when reducing it: keep increasing the step size without changing the step direction until the situation stops getting better. That is, walk along the line of the step by 2, 4, 8, 16, 32, etc., times the size of the initial step, rather than just 2 times, doing a crude line search once a promising direction is found. This will tend to ricochet you between the walls of a canyon, and eliminates the bias in the step-size random walk toward smallness, since any single lucky step in the right direction can increase the step size arbitrarily.

This is similar to the three-dimensional optimization strategy Dave Long tells me some bacteria use: flagella forward while things are improving, flagella backward (resulting in random tumbling) while things are getting worse.

In cases like those Adam and AdaGrad and the like are designed for, where the best step size varies enormously among different dimensions, you might want to use *hill climbing* — searching along one dimension at a time, so that you can use a separate step size for each dimension. This also potentially permits the use of Acar's

"self-adjusting computation" and similar generic incrementalization algorithms to speed up function evaluation. (See the section in More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) on incremental or self-adjusting computation.) Unfortunately, this means that you'll never make any further progress once you reach the floor of a diagonal valley. In *n* dimensions, there are $2n$ dimension-aligned directions a valley can descend in, but 2^n precisely diagonal orientations, so in some sense that's a much harder problem.

As an example of how relative scales on different dimensions matter, the above routine handles the following somewhat pathological problem reasonably well, although it takes it several hundred iterations; note the incorrect step function:

```
c = (3, 1e10, 4, 17, -5)
climb(lambda g: ((g-c)**2).sum(), lambda: numpy.random.random(5))
```

But it handles the following version much worse, requiring hundreds of thousands or possibly millions of iterations to solve it — it gradually exponentially converges on the right answer, but taking about 80'000 iterations per order of magnitude:

```
climb(lambda g: (((g-c)*(1,1,100,1,1))**2).sum(),
      lambda: numpy.random.random(5))
```

This induces a steep valley along the third dimension, and so the deltas in the other dimensions like the second and fourth dimensions suffer a drastically reduced step size, slowing their convergence enormously (from the outlandish values they took to get close to the optimum along the second dimension).

On one run, it finally found a solution accurate to two decimals on every dimension in iteration 886'803, after about 15 or 20 minutes of CPython interpretation:

```
886803 [  3.00416293e+00   1.00000000e+10   3.99999237e+00   1.70599240e+01
  -4.95000939e+00] 0.0177569891399
```

It goes about four times as fast with a non-buggy step function, converging to that same precision in only 207'804 generations:

```
207804 [  3.02129932e+00   1.00000000e+10   3.99999109e+00   1.70184720e+01
  -5.04976604e+00] 0.021740755647
```

The flagella-style variant described above improves on this by about a factor of 7:

```
28607 [  2.96086659e+00   1.00000000e+10   4.00023842e+00   1.69373547e+01
  -4.95732860e+00] 0.0155987598959
```

That's using the following implementation, which additionally is slightly simpler and only does one loss-function evaluation per iteration, instead of up to three:

```
def climb(f, step):
    size, here = 1, step()
```

```
    cost = f(here)

    while True:
        yield here, cost

        where = size * step()
        new = here + where
        new_cost = f(new)
        if new_cost > cost:
            size /= -2
            continue

        # We found a promising direction; swim!
        while True:
            yield new, new_cost
            where *= 2
            new_new = here + where
            new_new_cost = f(new_new)
            if new_new_cost < new_cost:
                new, new_cost = new_new, new_new_cost
                size *= 2
            else:
                break

        here, cost = new, new_cost
```

And this harness code:

```
from __future__ import division, print_function
import numpy

if __name__ == '__main__':
    c = (3, 1e10, 4, 17, -5)
    for i, (x, q) in enumerate(climb(lambda g: (((g-c)*(1,1,100,1,1))**2).sum(),
                                     lambda: numpy.random.random(5) - 0.5)):
        print(i, x, q)
```

As before, it takes four times as long if we leave out the `- 0.5`:

```
92241 [  3.04962626e+00   1.00000000e+10   3.99985342e+00   1.69648044e+01
  -4.97438500e+00] 0.00457665876023
```

Applied to the circle-intersection problem from before, it typically takes more iterations, like around 30 instead of around 10. (This is probably a somewhat smaller penalty than it sounds like, since the circle-intersection code from earlier is often doing two or three loss-function evaluations per iteration.)

It solves this geometric square model to a precision of 1e-6 typically in 1000 to 3000 iterations:

```
def square_constraints(points):
    p1, p2, p3, p4 = points[0:2], points[2:4], points[4:6], points[6:]
    side1 = p2 - p1
    side2 = p3 - p2
    side3 = p4 - p3
```

```
    side4 = p1 - p4
    return (side1.dot(side2)**2 + side2.dot(side3)**2 + side3.dot(side4)**2
            + side4.dot(side1)**2
            + ((side1**2).sum() - 5**2)**2
            + ((side2**2).sum() - (side3**2).sum())**2
    )

climb(square_constraints, lambda: 10 * (numpy.random.rand(8) - 0.5))
```

Without the fourth perpendicularity constraint, though, it converges very slowly, and the things it converges to don't look like squares. One sample result with the fourth perpendicularity constraint left out:

```
141071 [ 2.76119227 -2.36547768 -1.36787743  0.45423421 -1.35017868  0.48015388
 -1.33188753  0.50622078] 9.99823404211e-07
```

This has minuscule and nearly equal `side2` and `side3` (lengths about 0.032), whose dot product is consequently also very small even though they're very nearly parallel, being perpendicular to the nearly-parallel `side1` and `side4`. This allows `side1` and `side4` to minimize the function by becoming more and more nearly equal and opposite. It pays to be cautious when defining cost functions, since optimization algorithms will exploit whatever vulnerabilities they can find!

I haven't tried the per-dimension step-size approach above yet, though it would surely help for these examples.

Relationship to other kinds of optimization algorithms

See also Using the method of secants for general optimization (p. 1773) for a different generic derivative-free solver for scalar functions of general vector spaces, that one for zeroes instead of minima. I think both of these are only going to be reasonably fast for problems of low dimensionality, but I think hill-climbing suffers exponentially from high dimensionality, while the method of secants (like gradient-descent variants) will suffer only linearly from it.

Of course, you can find local minima of a computable continuous function by using automatic differentiation on it and using the method of secants to find a zero of its derivative.

Although the above implementations are not, hill-climbing and other kinds of local search are applicable to functions on discrete spaces without any kind of comparability between elements, such as graphs or strings of symbols. (That's the domain. The range still needs to be comparable.) The method of secants requires divisibility and zeroes. Hill-climbing also only requires comparability from its cost function — it takes no notice of its absolute magnitudes, just which values are higher and lower — and this *is* true of the implementations above.

Thoughts on quasi-Newton methods

For *regular* problems like the examples above, quasi-Newton methods would probably work better, but require differentiating the cost function. With the advent of automatic differentiation, this is no

longer a lot of work, but doing automatic differentiation on pure-Python functions like the above generally requires using forward-mode automatic differentiation, which is painfully slow. Reverse-mode automatic differentiation is dramatically faster for this sort of thing, but generally requires your program to be written differently.

As I said in Using the method of secants for general optimization (p. 1773), quasi-Newton methods require the maintenance of an approximation of the Hessian, and for functions of high-dimensional spaces, the Hessian is fucking humongous. A thing that has surely been tried, but that I haven't seen discussed, and which intuitively seems like it should work better for high-dimensional optimization problems, is using automatic differentiation to compute the gradient of the function to get a *direction* to move in, but then, rather than moving in that direction by an amount proportional to the *magnitude* of the gradient (as gradient-descent methods do), use automatic differentiation to compute the *second* derivative of the loss function *along that line*, which is a single number rather than a matrix of N^2 values (for a domain of dimensionality N) like the Hessian. Then you can use a Newton–Raphson step to figure out how far to move along that line: divide the first derivative (the dot product of the gradient and the direction) by the second derivative, thus extrapolating the distance to the point at which the gradient would fall to zero if the Hessian were constant over that region. (Alternatively, use affine arithmetic or reduced affine arithmetic to track down a precise zero along that line — see Fast mathematical optimization with affine arithmetic (p. 3163) for details.)

The appealing thing here is that gradient descent has linear convergence (i.e., the logarithm of the approximation error falls linearly with the number of iterations) while under appropriate circumstances Newton's method has quadratic convergence (i.e., the logarithm of the approximation error falls proportional to the square of the number of iterations). I'm not entirely sure whether Nesterov accelerated gradient descent, gradient descent with momentum, Adam, etc., achieve a better order of convergence, but I think they don't.

This would be big if true (see Things in Dercuano that would be big if true (p. 3136) and Top algorithms (p. 913)) which means that it, given how obvious it is, it probably doesn't work, or else I'm misunderstanding how quasi-Newton methods work, and they already do this. But it will be interesting to find out why not.

Topics

• Newton–Raphson iteration ("Newton's method") (p. 3595) (6 notes)
• Automatic differentiation (p. 3336) (6 notes)
• Method of secants (p. 3578) (4 notes)
• Gradient descent (p. 3480) (3 notes)

Very fast FIR filtering with time-domain zero stuffing and splines

Kragen Javier Sitaker, 2015-09-03 (updated 2015-09-07) (13 minutes)

It just occurred to me that maybe you can use splines to do convolution with large-support kernels without high-frequency components much more efficiently than using FFTs or time/space-domain convolution.

The underlying intuition here is that you can approximate the large kernel with a uniform spline of, say, second through fourth order, and then convolve that spline with the original data very efficiently.

This uses a series of inexpensive linear discrete-time time-invariant processing operations, specifically sparse-kernel convolution and boxcar filtering (simple moving average). But it does not fit into the standard FIR or IIR molds, so perhaps it's been overlooked so far. It is potentially an IIR filter, but the usual description of IIR filters in terms of linear combinations of past input and output terms does not yield an efficient implementation.

Splines

The spline curve through knots (0, 1) and (n, 0) for all other integral n approaches sinc as its degree approaches infinity.

XXX

Sparse-kernel convolution

A sparse convolution kernel is one that's zero at almost all points, except for a finite number of impulses. That is, the cardinality of its support is finite, and small compared to the bounds of its support. For example, in a discrete time domain, consider the kernel [-1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 -1]. It should be apparent that, although this kernel covers 17 samples, you don't need 17 multiplications per sample to convolve with it without transforming to the frequency domain; you only need three. (And, in this case, even those happen to be trivial multiplications.)

A *recursive* version of sparse convolution, where the convolution kernel is causal and its output is added to the input, is the computational primitive of Karplus-Strong string synthesis, and digital delay-line audio synthesis in general.

In general, convolving a discrete-time signal with an N-impulse sparse kernel in the time domain can be done in N multiply-accumulates per input sample, regardless of the width of the kernel's support bounds.

In particular, the sparse kernels we will consider have impulses at the knots of some uniform spline; since uniform splines have regularly spaced knots [XXX is that the right definition?], the impulses in our kernel are also regularly spaced.

Boxcar filtering

Boxcar filtering is convolution with a "rectangular function", which is some arbitrarily scaled and shifted version of the "boxcar

function", which is 1 from -½ to ½ and zero elsewhere. This filter is also known as the "moving average" (or, to avoid ambiguity, "simple moving average"), "running mean", "rolling mean", "box blur", or "mean filter". This kernel and sinc are Fourier-transform duals.

Boxcar filtering of a discrete-time signal is very efficient: a boxcar-filtered signal is just the difference between terms a fixed distance apart in the signal's prefix sum (aka indefinite sum or antidifference), although this imposes a constant multiplication on you that you may wish to undo with a division. This means that you can calculate an boxcar filter of arbitrary width with one addition and one subtraction per sample, plus a big enough intermediate FIFO to cover the boxcar's length.

There's an inherent tradeoff between preserving step response and filtering out noise. Among linear filters with a given step response, boxcar filtering by itself is optimal for reducing independent independently distributed additive noise; for example, removing noise from a photograph while minimally fuzzing out the edges. Nonlinear filters like median filters can do better, but other FIR and IIR filters can't.

Repeated convolution of the boxcar function with itself yields the sequence of cardinal B-splines, which are successive piecewise-polynomial approximations to a Gaussian. Any uniform B-spline can be expressed as a weighted sum of these cardinal B-splines.

That is actually a well-known technique for efficiently approximating a two-dimensional Gaussian blur of an image.

Efficiently approximating a spline-kernel convolution

Because you can express any uniform B-spline as a weighted sum of cardinal B-splines, you can choose a regularly-spaced sparse kernel that will produce a given uniform compactly-supported degree-M B-spline through the knots when convolved with a fixed-width boxcar filter M+1 times.

Since composition of convolutions is not just commutative but also associative, this means that convolving your original signal with the N-impulse sparse kernel and then with a boxcar filter M+1 times will precisely convolve the image with an arbitrary N-knot degree-M uniform B-spline. This requires N multiply-accumulates and 2M+2 additions and subtractions per sample, M+1 FIFOs whose size is each the impulse spacing. In floating-point, the initial sparse kernel can be premultiplied by a small constant to compensate for the undesired amplification resulting from calculating from computing the boxcar filters without normalization; in fixed-point, most likely a bit shift in between stages of box filtering will be needed.

(I think this spline technique generalizes to a nonseparable multiple-dimensional kernel; if that is so, its computational advantage in two or three dimensions should be enormously greater.)

This approach is a fully general optimization for linear FIR filters used as bandpass filters, band-stop filters, and low-pass filters. It probably isn't useful for high-pass filters, because you will frequencies all the way up to Nyquist in your kernel, which means that your spline knots are necessarily a single sample apart.

However, because you can ensure that your filter (the one you implemented, not just the one you were approximating) is

linear-phase, you can subtract the filtered signal from the (lagged to zero-phase or π-phase) original signal to achieve some limited high-pass filtering.

(Note that in this case, since your sparse kernel is required to be even, you can use the folded FIR filter trick to cut the multiplications required by half again.)

I am SWAGging that this means that you need a number of multiplications per sample comparable to the Q of your filter if it's a bandpass filter, regardless of frequency. That is, for a Q of 20, you might need 11, or 21, or 41 multiplications per sample, but I doubt that you'll need 128 or that you can get by with 5. This, in turn, means that these spline-based filters should be very competitive with FFT-based techniques for all one-dimensional filtering outside the design space where Goertzel takes over.

(I thought maybe you could improve the Q further by composing the filter with itself, but it turns out that doesn't work: the Q improves by √2, but the computation time doubles. You have to actually include more signal samples under the kernel to efficiently get better Q.)

Designing the sparse kernel

I think you can totally just take a continuous FIR kernel and sample it at its Nyquist frequency, i.e. multiply it by a comb filter, and then convolve the result with the sparse kernel of coefficients for the fundamental spline you're using.

Spline definitions I'm confused about

https://en.wikipedia.org/wiki/B-spline says, "Any spline function of given degree can be expressed as a linear combination of B-splines of that degree. Cardinal B-splines have knots that are equidistant from each other. … A fundamental theorem states that every spline function of a given degree, smoothness, and domain partition, can be uniquely represented as a linear combination of B-splines of that same degree and smoothness, and over that same partition. … A cardinal B-spline has a constant separation, h, between knots. The cardinal B-splines for a given degree n are just shifted copies of each other."

https://www.cs.unc.edu/~dm/UNC/COMP258/Papers/bsplbasic.pdf seems better. It says splines are the linear combinations of B-splines. "'B-spline' refers to a certain spline of minimal support and, contrary to usage unhappily current in CAGD [computer-aided geometric design], does not refer to a curve which happens to be written in terms of B-splines."

Aha, and it explains that cardinal splines are the splines whose knots are at ℤ. I thought those were "uniform splines".

Related work

Unser, Aldroubi, and Eden's delightful 1993 paper [0] pioneered the use of the running-sum algorithm for linear-time B-spline filtering. On p. 827, they explain B-spline filtering:

> We propose the concept of B-spline filtering which is the process of applying a filtering operator to the continuous B-spline representation of a signal. When the operator is discrete, this procedure is rather trivial and does not seem to have any specific advantages: due to the linearity of all operations, one may as well apply the

filter to the discrete signal and avoid the unnecessary transformation step. Of greater interest is the case where the impulse response of the filter itself is represented by a B-spline of order *p*.

(By "represented by a B-spline", they mean "represented by a weighted sum of B-splines", as the equation not quoted here clarifies; Unser et al. regularly use "B-spline" to mean "weighted sum of B-splines", a usage which de Boor deplores.)

From this they derive that you can do a discrete convolution of the B-spline coefficients representing the signal and the filter kernel to get the (higher-order) B-spline coefficients of the convolution result.

However, it appears that applying this result to a discrete signal requires first deriving the B-spline coefficients from the discrete signal by a discrete convolution with the B-spline coefficients of the fundamental spline, and then after computing the convolution result in B-spline form, converting back to discrete-signal form by discrete convolution with the (higher-order, and thus longer-support!) sampled B-spline. This is a substantial amount of computation. Also, the Unser et al. algorithm requires the signal and the filter to be sampled (i.e. have spline knots spaced) at the same sampling rate, so it is no more efficient for a filter containing only low frequencies.

This restricts the applicability of the 1993 spline filtering algorithm.

The present work improves over the Unser et al. algorithm by avoiding the necessity to convert the signal as well as the filter kernel to and from the spline representation; and, furthermore, for filter kernels bandlimited to low frequencies, it can approximate the kernel with a high-order spline with much more widely spaced knots, resulting in much less computation.

Ferrer-Arnau et al. 2013 1 developed a filtering algorithm which superficially sounds very similar to the present one: they improved Unser et al.'s spline filtering algorithm by an approximate FIR filter. Upon further investigation, they developed a particular FIR filter to approximate the process of fitting a cubic spline to noisy data, rather than developing a spline to approximate an arbitrary FIR filter.

Vrcelj and Vaidyanathan 2001 [2] also uses FIR filters to approximately transform a signal into B-spline coefficients, rather than using B-spline coefficients to approximate a FIR filter.

CIC filters (Hogenauer 1980) are very similar indeed to the present work, consisting as they do of a composition of a cascade of running-sum filters ("integrators"), factored into a cascade of integrators, decimation, and a cascade of differentiators; or, for upsampling, a cascade of differentiators, zero-stuffing, and a cascade of integrators. Hmm, XXX if you stick a FIR filter in between two CIC filters running at a lower rate, is it the same thing? No; the CIC filter is *much more efficient*. The recommendation to put your compensation filter's cutoff below a ¼ of the first null frequency probably requires more FIR coefficients than I was thinking were necessary, but probably also applies to the present work!

http://www.embedded.com/design/configurable-systems/4006446/oUnderstanding-cascaded-integrator-comb-filters

https://www.altera.com/content/dam/altera-www/global/en_US/opdfs/literature/an/an455.pdf

o "B-Spline Signal Processing: Part I—Theory", by Michael Unser, Akram Aldroubi, and Murray Eden, IEEE Transactions on

Signal Processing, Vol. 41, No. 2, February 1993, pp.821–832, IEEE Log Number 9205111,
http://bigwww.epfl.ch/publications/unser9301.pdf.

1 "Efficient cubic spline interpolation implemented with FIR filters", by Lluís Ferrer-Arnau, Ramón Reig-Bolaño, Pere Marti-Puig, Amàlia Manjabacas, and Vicenç Parisi-Baradad, International Journal of Computer Information Sy stems and Industrial Management Applications, ISSN 2150-7988 Volume 5 (2013) pp. 098-105.
http://digital.csic.es/bitstream/10261/71847/1/IJCISIMA5_98.pdf

[2] "Efficient Implementation of All-Digital Interpolation", Bojan Vrcelj and P.P. Vaidyanathan, July 19, 2001, EDICS number 2-INTR
http://gladstone.systems.caltech.edu/dsp/students/bojan/journ/IPtra°°n_01.pdf

Topics

- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Convolution (p. 3391) (15 notes)
- Splines (p. 3727) (6 notes)

Circle-portal GUI

Kragen Javier Sitaker, 2016-06-03 (11 minutes)

I was thinking about a zoomable, rotatable UI. In this world, there's some "world space" in which the scene graph exists, and your screen is an arbitrarily translated, rotated, and zoomed portal onto it, considered as a linear function in complex numbers: $y = mx + b$, where y is a point on the complex plane of world space, x is a point on the complex plane of screen space, and m and b are complex numbers that determine the translation, rotation, and zoom.

The world plane contains only one kind of object, which is mutable: portals.

A *portal* is a circle through which another part of the world plane is visible; like the screen, it consists of an (m, b) pair defining an arbitrary translation, rotation, and zoom, this time from the world plane onto itself. However, it also has a *clipping circle*, defined by a center and radius outside of which it is not visible, and a *background color*, which is an RGBA color that it renders underneath everything else seen through the portal.

Portals already existed (with that name) in Pad++ decades ago; in this system there are four differences:

- they are round, not rectangular;
- they can rotate as well as zooming;
- nothing else exists;
- they have a background color.

There is a total order among portals in the world plane, with portals earlier in the order being "painted on top of" portals later in the order. This may or may not result in the later portal being invisible because it's been obscured by an opaque background color.

Optimizations

Given the necessity to sample the world plane at pixel coordinates, you can do strength-reduction of the pixel-scanning operation; if Δx is the coordinate difference between pixels on the same row, you can compute $m\Delta x$ and scan the pixels on the row by repeatedly adding that.

To keep drawing quick, it's probably necessary to dynamically compute some kind of bounding volume hierarchy, which could be made out of circles or not.

As you move scan line by scan line down the image, you can use the midpoint algorithm (which incrementally updates an error accumulator $x^2 + y^2 - r^2$ as it changes x and y to walk along the circle) to incrementally adjust the start point of each circle.

You can cache parts of the world plane rasterized at some resolution, downsampling from there. However, it may be tricky to propagate updates.

Making things with it

First, of course, you can draw things by putting opaque portals on top of or next to each other. Those portals don't need to view anything in particular. At its simplest, you can use opaque black

circles all of the same size as if they were dot-matrix pixels, but you can also overlap a bunch of portals (either translucent or of different colors) to get a gradient, for example, or use two concentric circles of different colors and slightly different radii to get a curved line of some thickness.

Once you have a picture somewhere, you can add portals to it elsewhere in order to use it in more places. Changes you make to the original picture will be reflected in each of the copies. For example, you could draw a letter "C" in one place, add a portal to it, and add a crossbar on top of the portal, transforming it into a "G". Further modifications you make to the "C" — adding a serif, for example — will be reflected in the "G". If you draw glyphs for the letters "e", "t", "o", and "n", you can set up a series of portals to those letters to spell phrases like "no note on teen tenon tent to neon noon onto no tone". The rendering of the phrase will automatically reflect whatever changes you make to the original letterforms.

Furthermore, you can make a portal for a whole set of such pictures, such as an entire monospaced font, and use portals onto that portal to pull out individual letterforms. Then, if you redirect the portal to a different set of pictures, you have achieved a change of font. You could even have an area with several different font variants, such as plain, bold, italic, and bold-italic.

This works for monospaced fonts that don't differ too much in aspect ratio, and for proportional fonts that share the same font metrics, but it won't work very well for changing fonts between fonts with different metrics, including often changing between the plain and bold versions of the same font.

(You can make a crudely bolded version of such a font by making two overlapping portals to it that are shifted by less than a linewidth, thus thickening its lines.)

By putting portals within their own target, you can automatically generate fractals — which of course implies that the rendering code needs to handle infinite recursion gracefully.

User interface interaction techniques thus enabled

Since everything you see on the screen is a portal to "somewhere else", you can provide a command to jump to that other place, at which point you can zoom in or out as you wish. And you can add such a portal from anywhere to anywhere.

More commonly, you'd like to zoom in to look at a particular portal, without jumping through the portal. This should help a lot with the problem ZUIs have with constrained pixel displays, where things are almost always either too big or too small.

Conversely, you can display a set of "backlinks" to a place that are visual thumbnails in context of places that have a portal overlapping that place.

Because everything is a portal, in addition to jumping through it or moving it around, you can also drive it around the space it views, zooming, panning, and rotating.

Probably the usual way of making a new portal should be cloning an existing portal, after which you can start to make changes to it.

Minimal interactions necessary to use it

There are only a few basic actions:

- creating new portals;
- deleting portals;
- changing the background color of a portal;
- changing where a portal looks (its destination point, zoom, and rotation);
- changing where a portal appears (its location and radius).

For minimality of mechanism, the viewport portal should not require specially-implemented commands for it. It's just that deleting it or changing where it appears are inapplicable operations.

The usual map navigation actions are panning and zooming, and panning usually results from left-mouse dragging or finger-dragging. It would maybe be kind of unfortunate if left-mouse dragging did something to change the world, but of course that kind of conflicts with the desire for the viewport portal not to be special...

You might be able to get by with only jump-to rather than free panning and zooming: click on a portal to make it mostly fill your viewport. But that really only works for the viewport portal. Ideally (for minimality of mechanism) the viewport portal would be much like any other portal.

So that means that jump-to takes two different portals as arguments: the jump destination and the thing you want to cause to point at it (your "focus portal").

Even telling what you're clicking on may be a little tricky, since you can be looking at a portal that is viewing a zoomed-in view of another portal which is zoomed into a third portal, etc.

Practically speaking, you probably need to be able to copy portals so that you can make modified versions of them. But you can have two kinds of "copy relationships" between portals: a portal can simply be a view of another portal, or it can be a clone of it. It might make sense to start with a "create a view" command and then possibly later convert that into a "make a copy of the viewed scene".

At some point it may also be useful to push objects through a portal.

I'm thinking that probably when you click (or double-click?) on a portal a halo of buttons should appear around it offering you different options: one to change its size, one to move it around where it is displayed, one to delete it. Probably zooming and panning the selected portal should just be the usual mouse drag and scroll i

Prototyping

I've talked above a bit about techniques that might be useful for an efficient implementation of this system.

However, it should be possible to hack together a *simple* implementation of the system, enough to play with, much more easily... and so I spent a couple of hours on that in DHTML with `<canvas>`.

Real-time responsiveness

In the form described above, it's possible to require an arbitrary amount of computation per pixel; in fact, if the pixel happens to be a fixed point of a portal transformation, the amount is theoretically

infinite. This is intolerable for real-time user interface responsiveness.

To avoid this problem, I propose that we render portals each frame not from their *current* contents but from their *previous* contents, cached as a raster image from the previous frame. If no previous contents are available, use some placeholder texture that's easy to compute.

In the absence of alpha-compositing, this would guarantee that it's possible to calculate the screen image each frame with a single texture sample per screen pixel, plus some increment of work in updating offscreen texture buffers; if this increment is small, updates might take many frames to settle, while if it is large, they should settle quickly. This background increment itself has the same real-time pixel bound as the screen update. Normally you would expect the increment to be many times the size of the screen. For example, the VideoCore IV used in the Raspberry Pi claims a fill rate of a gigapixel per second, but only 2.1 megapixels of output screen resolution at most (and only one megapixel by default), so it should be able to compute almost seven screenfuls of offscreen texture updates per frame in the worst case.

To guarantee real-time responsiveness in the presence of alpha-compositing, partially-alpha-composited images should also be cached. Ideally you'd do this front-to-back, inverse-painter's-algorithm style.

Finally, it makes sense to schedule this offscreen texture updating to happen *before* the screen is painted rather than after, so that you'll only see lag in the case where there's more work to be done than can actually be done in a single frame time.

Topics

How inefficient is SNAT hole-punching via random port trials?

Kragen Javier Sitaker, 2018-04-27 (2 minutes)

Symmetric NAT is a tricky problem for peer-to-peer applications. Ierymenko says you can get 96% success in establishing connections if you do port prediction, but also that you can succeed eventually with random port trials because there are only 65535 ports.

The idea is that A and B send a pair of UDP packets to each other at the same time (on the mark of some third party) and have a 1/65535 chance of happening to guess the correct UDP port for the other. So if they do this once a second, they will succeed on average in 18 hours, at an average cost of 65535 packets per party. The time to success takes an exponential distribution.

I was thinking that perhaps you could increase your success by sending larger batches of packets, so that each packet has a larger "target size" to aim at, but that only makes sense for full-cone and address-restricted-cone NAT, which can be tackled by easier techniques anyway.

The purely random approach can still work faster than one packet per second. 16 packets twice a second should be feasible in most cases, which should take 2048 seconds on average. As long as this doesn't crash your NAT or disrupt your existing connections, which should be detectable, the expected bandwidth is (20 bytes IP header + 8 bytes UDP header) * 65536 = 1.8 megabytes, roughly the same as loading one extra web page.

Topics

- Programming (p. 3658) (286 notes)
- Decentralization (p. 3404) (13 notes)
- Networking (p. 3594) (7 notes)
- TCP/IP (2 notes)
- NAT

3-D printing glass with continuously varying refractive indices for optics without internal surfaces

Kragen Javier Sitaker, 2016-10-06 (3 minutes)

For the "cloak of invisibility" using optical metamaterials, you don't actually need a negative refractive index; you just need to be able to precisely grade the refractive index throughout space. The approach they're taking is to fabricate nanostructures made of resonant components, but this has high dispersion, and so the invisibility fades as you use multiple wavelengths.

As an alternative, you could vary the mix of materials going into a block of glass. More lead oxide in one part, more silicon and aluminum oxides in another. But how can you achieve that? Depositing glass powders and then sintering them will leave you with a part that's full of voids. Voids are potentially problematic in structural parts (e.g. metal mounting brackets) but totally fatal in optics.

If instead of sintering the powders you fully melt them so that bubbles can rise, the bubbles will mix together the glass that they float through, and to a lesser extent the glass around it, especially if you put it under vacuum to degas it like in resin casting. Also, if the viscosity is too low, you may get convection currents.

Depositing the glass by FDM in air seems like it would hardly be any better.

A possible alternative would be to immerse the workpiece being built in molten lead oxide, using either selective powder deposition or FDM extrusion to deposit a higher-melting glass "underwater", thus avoiding bubbles; the molten lead oxide would fill any voids. The lead oxide would immediately begin to "flux" or dissolve the higher-melting glass, so tight temporal control of the process is critical. That same process of dissolution or diffusion can continue even below the glass transition, but more slowly, and is crucial to achieving a smooth gradient, but it also limits the strength of the gradient that can be achieved. It may be necessary to bake the finished workpiece in a solid state for some period of time after completion.

(The lowest-melting glass may not be pure lead oxide, but rather some mixture; ideally you'd use the lowest-melting glass for the immersion medium. Above the glass you could use a soft vacuum)

Speaking of gradients, one way to reduce the variability in such a process would be to maintain a vertical temperature gradient in the workpiece being built layer by layer, such that only the surface is above the point of lead oxide, while the layers below are below the glass transition temperature. This won't prevent diffusion, just slow it down, but it will stop slumping.

Optical systems built in this way, using gradients rather than surfaces, can entirely avoid the problems of stray light from unwanted

reflections from the surfaces of lenses, although total internal reflection is still possible, as in a graded-index optical fiber.

Topics

Optimization-based painting software

Kragen Javier Sitaker, 2018-04-27 (1 minute)

Optimization-based painting software are going to be a big deal.

What I mean by that is software that generates an image according to some kind of model of the image-forming process, controlled by a sketch made by the user. Perhaps at a reduced pixel count, or reduced coloring, or with added noise due to mouse, or whatever.

Perhaps the model involves lines, or wavelets, or gradients, or boundaries between areas, or three-dimensional objects, or people, or animals, or convolutional neural networks, or whatever. It's relatively straightforward to generate an image from such a model, and then you can compare the image to the image the user has drawn, using a metric that takes into account the kinds of errors people don't intend, in order to infer a highly probable underlying model — if not the most probable one given the whole set of possible models, which is probably infeasible to compute, at least a reasonably probable one. Then, given this underlying model, you can generate a high-fidelity image of what the user intended.

But that's just the beginning, because then you can modify the image in order to correct the model, you can select among a variety of images representing different models, and you can select among many images that potentially represent the same model.

Topics

- Graphics (p. 3483) (91 notes)
- Mathematical optimization (p. 3611) (29 notes)

How should we design a UI for a new OS?

Kragen Javier Sitaker, 2012-10-10 (updated 2012-10-11) (4 minutes)

I'm suffering quite a bit from trying to use Linux, Windows, and MacOS on current hardware. There's no reason, for example, for this laptop to spend thirty seconds or more to become ready for use when I want to use it. Its video display holds perhaps 3 megabytes of data, and it's capable of reproducing 60fps fullscreen 1024x600 video, which means it can draw the full screen contents, including running the video codec, in 16ms. 30 seconds is 1800 times longer than it ought to take to get to a useful screen.

There are responsiveness and performance problems all over the place. It's totally unacceptable that Ubuntu's pretty main menu takes three seconds or more to come up when I press the Windows key. Alt-tab between two windows that are already open has a noticeable delay, which means it's over 200ms. And, although the machine has a gigabyte of RAM, it frequently runs into OOM kills and memory-shortage kernel panics. Even on Ubuntu machines that aren't suffering from that particular problem, the machine often has to swap for a few seconds in order for the screen saver to display its password prompt.

And then there are things that just don't work. The trackpad stopped working (maybe when I rebooted with Ubuntu 12.04). The wireless disconnects and then won't reconnect without rebooting, a problem I also had with Ubuntu 8.04 on a Dell Vostro 9. The machine doesn't sleep when I close the lid, and when I sleep with the sleep key, it doesn't wake up reliably --- sometimes it gets into a mode where the screen turns on after every keystroke for 100ms or so.

I suspect that a lot of these problems are a result of the monolithic kernel design. This kernel has 43 modules loaded, including modules for the sound, the wireless, the mouse, the nonexistent parallel port, and the nonexistent Bluetooth interface. None of these can be restarted cleanly, and a bug in any of these subsystems can cause problems in the system as a whole, and although this is of course also the case with any driver that controls hardware that supports DMA, many pieces of hardware nowadays don't support DMA, or support DMA only in well-defined ways, e.g. via USBI UHCI.

Furthermore, although it's often possible to work around bugs in these subsystems by rebooting the machine, the kernel that gets rebooted is also the kernel that's running all your applications, so your applications lose their state too --- even though there's really no possibility that vi, Emacs, the GIMP, Firefox, or GnuCash has runtime state that's so tightly coupled with your video card that it can't be reasonably preserved across a restart.

On top of all this, sound often skips when playing MP3s if there's other stuff going on, like a compile or some web browsing.

So I propose a new system architecture for end-user free-software systems:

• a small hard-real-time hypervisor that's responsible for providing a trusted path to a low-latency user interface, protecting access to any DMA-capable hardware, and checkpointing and resuming virtual-machine images running under it;
• a display server running directly under the real-time hypervisor, handling the keyboard, mouse, and monitor, which is unavoidably part of the TCB because it provides the trusted user interface path;
• a Linux kernel, or several, running under the real-time hypervisor to handle other devices that need DMA; this is unfortunately part of the TCB.
• a separate Linux kernel, or several, running applications that don't need direct access to hardware at all; these can be checkpointed and resumed without knowledge of the applications.
• other Linux kernels to run applications that need direct access to hardware that doesn't do DMA. These can't be checkpointed and restarted, but they also aren't part of the TCB.

Topics

Circuit notation

Kragen Javier Sitaker, 2016-09-08 (updated 2017-04-18) (7 minutes)

(See also Graph construction (p. 3226).)

I was thinking there must be a way to write down circuit diagrams in an expression-like textual string in a compact way. This "example circuits and netlists" page gives the following example netlist for SPICE:

```
Multiple dc sources
v1 1 0 dc 24
v2 3 0 dc 15
r1 1 2 10k
r2 2 3 8.1k
r3 2 0 4.7k
.end
```

That defines two voltage sources, three resistors, and four nets (nodes). But it's a purely series-parallel circuit. If we use `|` for parallel composition, mere concatenation for series, `'` for inversion, and `[]` for grouping, we could write it as `V 24 R 10k [R 4.7k | R 8.1k [V 15]']`, which I think would be a huge improvement. Better still, without the component values, that's just `VR[R|RV']`.

You'd still need some way to handle circuits that aren't purely series-parallel, that aren't circuits, or that contain non-two-terminal elements. For example, further down, we have this bridge-rectifier circuit:

```
fullwave bridge rectifier
v1 1 0 sin(0 15 60 0 0)
rload 1 0 10k
d1 1 2 mod1
d2 0 2 mod1
d3 3 1 mod1
d4 3 0 mod1
.model mod1 d
.tran .5m 25m
.plot tran v(1,0) v(2,3)
.end
```

That's not series-parallel. But you could imagine defining a particular two-terminal chunk of it that occurs in more than one path through the circuit as `rload = R 10k;  V ac 15 60 [D rload D | D' rload D']`.

Alternatively you could do what we do in programming languages with gotos, where the named entities are not subgraphs we wish to traverse and return from (like functions) but labels for points in the control-flow graph. Suppose we mark them with an `@` suffix to distinguish them. Then we might write that circuit as `ra@ R 10k rb@;  V ac 15 60 [D ra@ D' | D' rb@ D]`, which is slightly shorter but maybe less clear. It's not quite clear what to do when you want a branch to *end* in such a label rather than to merely include it. This does, however, provide a more reasonable way to handle three-terminal elements.

In both of these cases, the stuff before a semicolon is special in that

it's in some sense mere definitions; it's what's after the last semicolon that gives the circuit.

Right now I'm looking at a diagram of an energy-harvesting circuit in a 2003 paper which could be written as follows:

```
q1@ [I ac | C] q2@; [D' q1@ D' | D' q2@ D' | C] S [D' | L [ C | [V'|R] R]]
```

which, as you can plainly see, is a capacitive (piezoelectric) current source feeding into a bridge rectifier powering a buck converter to charge a battery, with the buck converter controller being represented by a plain switch.

Without spaces, that would be the rather awful `q1@[Iac|C]q2@;[D'q1@D'|D'q2@D'|C]S[D'|L[C|[V'|R]R]]`.

It would be very nice to be able to render such expressions into schematics (and SPICE netlists) automatically.

Stack machines

An alternative approach would be to add circuit elements using stack-machine operations. In this approach, `R` is a stack operation which takes a node off the operand stack, attaches a resistor to it (of a specified value, if one is present), and returns a newly created node at the other end of the resistor. `D` and `D'` are stack operations which attach diodes in opposite directions.

Three-terminal devices fit comfortably — although there are more potential ways to define them, as their terminals could conceivably be assigned to inputs and outputs in 18 different fashions. The most general form is to merely push all of their terminals onto the stack as fresh nodes; given that and some stack manipulation operations, you can define the other 17 fashions in those terms.

How can we hook up circuit elements in parallel? We need to save the beginning of the parallel section on the stack with `[`, save the end of the first branch with `|` and return to the beginning, and then connect the two ends with `]` when we reach them. (Note that this implies that parallelism is of strictly two branches.) This suggests that `[` is just DUP and `|` is just SWAP, while `]` is a graph-construction operation that shorts two nodes together and then drops one of them.

In this interpretation, then, non-series-parallel circuits are merely circuits that don't respect the stack discipline on node access; they could be defined like Forth constants or PostScript variables, and whenever invoked, they short the node on top of the stack to their node (and leave it on the stack). Perhaps rather than `80 CONSTANT WIDTH` we should write something like `80 ]: WIDTH`, since if we're defining a node, we probably started by duplicating a node with `[`, and it's nice if the parens line up.

So then what does this previous circuit look like?

```
q1@ [I ac | C] q2@; [D' q1@ D' | D' q2@ D' | C] S [D' | L [ C | [V'|R] R]]
```

It becomes the following:

```
[ D' [ [ ac I | C ] ]: q | [ D' q D' | C ] ] S [ D' | L [ C | [ V' | R] R ] ]
```

However, in this version, we have a way to handle the special case of the implicit circuit around everything: an extra `[` at the beginning

saves the starting point, and an extra] at the end unifies them. So this becomes:

```
[ [ D' [ [ ac I | C ] ]: q D' | [ D' q D' | C ] ] S [ D' | L [ C | [ V' | R] R ] ∘
∘] ]
```

Or, if we use a more sophisticated tokenizer than Forth's:

```
[[D' [[ac I|C]]: q D' | [D' q D' | C]] S [D' | L [C | [V' | R] R]]]
```

Or, if we use more fashionable punctuation, maybe

```
{{~d {{ac i, c}}=q ~d, {~d q ~d, c}} s {~d, l {c, {~v, r} r}}}
```

I'm not sure if this is an improvement in readability over the previous expression, but the semantics are a lot less muddy! It lacks the feature the other version had, though, where reversing the direction of a circuit element was a general-purpose feature.

This version is strongly reminiscent of Binate, though, with its concatenation-for-series-and-comma-for-parallel, and I think it's probably possible to take the analogy further to the possible benefit of one or the other language. In particular, Binate's approach to "named terminals" is more readable, and Binate does handle converse orthogonally, but this thing's approach to "named labels" is cleaner.

Topics

- Electronics (p. 3430) (138 notes)
- Syntax (p. 3738) (28 notes)
- Stacks (p. 3730) (21 notes)
- Graphs (p. 3486) (5 notes)

Some musings on applying Fitts's Law to user interface design and data compression

Kragen Javier Sitaker, 2019-05-06 (updated 2019-05-09) (27 minutes)

Fitts's Law says MT = a + b lg(1 + D/W): we divide the distance to the center of the target by its width along the line of motion, add 1, take the logarithm (in base 2, by convention), multiply by some parameter b, and add some other parameter a, to get the movement time.

As described in Dercuano drawings (p. 64), I want to use Fitts's Law to design a bandwidth-limited drawing system for Dercuano. I can type about 72 bits per second if we count ASCII, or 24 bits per second if we use the gzipped weight of the text, or 45 bits per second if we consider the keyboard keys as being 5 bits each. Since I often have to stop and think while I'm typing, probably the bit rate for drawing should be somewhere in the neighborhood — maybe 5 bits per second, or maybe 500, but probably not 1000 or 10000. That is, some part of my mouse movements is basically random, and could be rounded off with no loss of intentional information. Fitts's Law can perhaps tell me what part.

We know from Fitts's Law that the imprecision of movement in a given time is proportionally greater when larger distances are being covered, although this flattens out for movements comparable to the target size. Solving for D/W as a function of MT, a, and b:

```
(MT - a)/b = lg(1 + D/W)
2**((MT - a)/b) = 1 + D/W
2**((MT - a)/b) - 1 = D/W
```

So, when MT = a, the start/stop time, D/W must be zero; when MT = a + b/2, you can reach targets about 0.414 of their width from your starting point (this is a bit silly, since you start inside of them!); when MT = a + b, you can reach targets 1W away (requiring you to move between 0.5 and 1.5 times their width in their direction); when MT = a + 2b, you can reach targets 3W away; when MT = a + 3b, you can reach targets 7W away; when MT = a + 4b, you can reach targets 15W away; and so on.

What does Fitts's Law suggest about the channel bit rate?

Consider a pixel-resolution pointing device, and suppose your starting point is surrounded by approximate circles of circular target buttons, each of radius 15W, thus containing about 47 targets each. If you move your mouse to the right, you cross the first of 8 circles of 2-pixel targets after 30 pixels, and there are 8 such circles, the last ending before pixel 46. Pixels 46 to 61 are divided into three-pixel-wide targets, pixels 61 to 77 into 4-pixel-wide targets, pixels 77 to 92 into 5-pixel-wide targets, and so on. By the time we

reach pixel 400 we have crossed 44 targets and entered a 45th; Fitts's Law suggests that we should have been able to select any of these 44 targets in time a + 4b. Moreover, since each of the circles has 47 targets arranged around it, we had a total of 2068 targets, all selectable in that same time: 11 bits of target. If b ≫ a (let's assume that for now), then that's 2.8 bits per b.

Now consider filling the same area with smaller targets, each of radius 31W, thus fitting 97 round targets around each concentric circle, and also about twice as many circles: 89 circles, say. So we have 8633 possible targets, which makes 13 bits, selectable in time a + 5b, which is only 2.6 bits per b.

This trend suggests that the highest possible bit rate would come from packing circles with the lowest possible D/W, which would be about 1. So your 2-pixel target is pixels 1 and 2, a 3-pixel target is pixels 2:5, a 9-pixel target is pixels 5:14, a 28-pixel target is pixels 14:42, an 84-pixel target is pixels 42:126, and a 252-pixel target is pixels 126:378, so in the same area as before, you have six "circles", each of which can contain about six targets. (You could probably do a bit better by staggering the circles to move the centers of targets further apart, but the improvement should be small.) So you have about 36 targets, 5 bits, selectable in time a + b, which would be 5 bits per b if a were small. Our rationale for assuming a is small has gone away, since we're no longer looking at the limit of large D/W, but it still seems likely that this is the best case.

Still, that's a much less variable bit rate than I was intuiting. In this tautochronic arrangement, every doubling of D/W adds 2 bits to the information content of the selection and adds b to the time. So if a = 1.5b (or perhaps a bit more, if the denser arrangements I mentioned pan out), the bit rate would stay fixed at 2 bits per b.

Of course, if unboundedly far-off targets are allowed, there is no limit to the bit rate. Instead of six "circles", you could have 12 "circles" of targets (the largest having targets 183708 pixels wide) or 24 (the largest having targets 97629963228 pixels wide — about 16000 km), and that would give you almost 7 bits per selection, which — according to Fitts's Law — would still take the same amount of time, MT = a + b. The number of bits per constant-time selection grows as log log D, though, and as you can see, that's effectively constant. Also I am going to go out on a limb here: I don't think you can actually move your mouse cursor 16000 km in the same time you can move it 84 pixels, even if your screen *does* scroll and anywhere from 9200 km to 28000 km *would* be close enough.

A different reductio ad absurdum of Fitts's Law (in this basic form) is that it has no term for the width of the target in the circumferential direction of motion. Above, I've considered the case where the width of the target is the same in all dimensions, but consider crossing-based user interfaces; in these, to activate a target, instead of moving the mouse to within a target and clicking on it, you just need to move the mouse across the border of the target, possibly without even stopping. Effectively, the "target" has infinite width in the direction of motion, which means D/W = 0 and MT = a + lg (1 + 0) = a + lg 1 = a + 0 = a. This would imply a constant time to select one of many targets that can be crossed in a straight line from where your mouse starts, regardless of how many such targets there are and thus how precise your angle has to be to hit the target you want, and also regardless of

how far away the targets are from the mouse's starting point. (The same argument would suggest that the time to select a known item from a pie menu is independent of the number of items in the menu if its inner and outer radii remain unchanged.)

Estimating the parameters from my handwriting

My handwriting is about 13 words per minute, which is roughly one letter per second. Each letter could be reasonably approximated by a couple of cubic Bézier curves, each of whose control points has an error in the neighborhood of, eyeballing, 10% of the distance from the previous control point, so perhaps D/W is about 7. Given the guess of 8 control points per letter and thus 125 ms per control point, this suggests that a and b are in the neighborhood of 30 milliseconds each; this in turn suggests a bit rate on the order of 64 bits per second for my handwriting. I write about half as fast with the mouse, so perhaps 32 bits per second.

This suggests that it should be safe to round drawing movement coordinates to about 64 bits per second. For explicit placement of anchor points with separate clicks, this is straightforward to apply: if the time since the last click is 250 ms, round it to 8 bits each of x and y for a total of 16 bits, but if it was 283 ms, use 9 bits each, and at 313 ms, use 10 bits each, etc. (4 Hz is about as fast as I can click with a mouse.)

I've rigged up a primitive experiment with the mouse and some SVG and JS to present me some circles to click on, and the data from the experiment is surprising. Fitts's Law does seem to hold, broadly speaking, but there's a lot of variability, like, a lot of residuals are on the order of half of the height of the trend line — variability gets bigger as task time gets bigger, maybe because I have a hard time hitting the tiny circles with the mouse or sometimes even seeing them. The residuals are far from normally distributed. A linear regression on 417 data points finds $a = 220$ ms, $b = 340$ ms, $R^2 = 0.7$; this means clicking on circles that appeared centered under the mouse took about 220 ms; circles whose center was one diameter away from the mouse took about 560 ms; circles whose center was three diameters away took about 900 ms; circles whose center was seven diameters away took about 1200 ms; and so on. What I did in R was this:

```
fitts.data <- read.csv('fitts-data-cleaned.csv')
fitts.data$fitts <- log(1 + fitts.data$D / fitts.data$W) / log(2)
model <- lm(MT ~ fitts, fitts.data)
plot(fitts.data)
plot(model)
plot(MT ~ fitts, fitts.data)
abline(model)
summary(model)
```

This is dramatically slower than I had anticipated! It suggests that my bit rate at moving the mouse to an area on the screen is closer to 6 bits per second than to the 32 bits per second I was hoping for or the 64 bits per second I get with a ballpoint pen.

(Further examination of the data suggests that the biggest residuals do come from the smallest circles, but in both upward and downward direction, so removing the smallest circles actually reduces the R^2 of the regression. *The Grammar of Graphics* §9.1.4.1 suggests applying a projective transformation to the data (MT, fitts) → (1/MT, fitts/MT) in order to eliminate its egregious heteroskedasticity, but I haven't tried that.)

On the other hand, a better workflow may be to sketch a whole freehand curve with the mouse or other input device, and then optimize an overall representation for it, in terms of lines, splines, and corners. I just opened Inkscape and scrawled "On the other hand, a better workflow may be to" in it, which took 163 seconds (4 wpm, a third of my speed with a pencil or ballpoint) and involved 37 glyphs (say this would require 300 control points), with rather larger errors (say D/W ≈ 3). This is about two control points per second with about 3b entry time (implying about 6 bits per control point, 12 bits per second) each, which is about 4 times worse than the estimate above from my normal handwriting, but still twice as good as the estimate from my SVG experiment.

Going further in that direction, maybe the right approach is to sketch things on paper, photograph them, scan in the photographs, and construct low-Kolmogorov-complexity approximations of the images. If I'm really getting 6–12 baud with the mouse, 45 baud with the keyboard, and 64 baud with a ballpoint pen, it would seem to make sense to just use the pen! Otherwise I could easily end up spending an hour and a half on a sketch that could have taken ten minutes — or, more likely, not making the sketch at all.

Vector encoding

If the objective is not to impede drawing but to minimize the Dercuano download size, it isn't sufficient to avoid mixing in a bunch of unintentional quantization noise; we also need to think about how to represent the displacement vectors that make up the drawing as bits.

8 bits per coordinate? What, with fixed fields?

Perhaps this could be a floating-point format with a sign bit, 2 bits of exponent (2^0, 2^1, 2^2, or 2^3 pixels per count), and a 5-bit mantissa. Exponent of 0 would be "denormalized" pixel counts: 0 is 0, 1 is 1, 2 is 2, etc., up to 31 is 31, but exponent of 1 and up would have an implicit leading 1, so 0x20 would be 32, 0x21 34, 0x22 36, etc., up to 0x3f, which would be $32+2\times31 = 94$, and then the exponent of 2 would similarly start at 0x40 for 64 and go up by fours: 0x41 for 68, etc. 0x7f in that scheme ends up being $256+8\times31 = 504$ pixels.

Truncated Golomb coding?

Golomb coding is a lossless encoding that concatenates an unary bucket identifier with a binary within-bucket index; it's the optimal prefix code for the geometric distribution. The buckets are (the intervals between) multiples of the bucket size parameter M; to encode a nonnegative integer, you do an integer division by M, encode the integer quotient q in unary (as, say, a series of q-1 1s followed by a 0), and then append the binary encoding of the remainder, using the number of bits necessary to encode all

nonnegative integers less than M. That is, the remainder is a fixed-length field for a given parameter M.

Suppose that we use M = 64 and truncate the result to fit in 7 bits. (We can, again, encode the sign bit separately.) Numbers less than 64 are encoded in binary as 0xx xxxx; numbers 64:128 are encoded with only 5 significant bits, thus rounding to even pixels, as 10x xxxx; numbers 128:192 are encoded with 4 significant bits as 110 xxxx, thus rounding to every fourth pixel; 192:256 are 111 0xxx, rounding to every 8th pixel; and so on until 111 1110 represents 384 and 111 1111 represents 448. Thus we have progressively worse resolution for large moves, rather than the fixed resolution Fitts's Law would suggest.

(You could omit the initial 1 of the unary code in this fixed-width context, but that doesn't overcome the overall problem.)

How about truncated Elias delta or gamma coding?

Gamma coding is a prefix code that represents an arbitrary positive integer as an excess-1 unary bit count (traditionally written as a number of leading zeroes, with no leading zeroes meaning the one-bit number 1) and then the number itself; so $5 = 101_2$, is written as 00101. Delta coding is "flatter"; it uses gamma coding to encode the number of bits in the number and then appends the number (without the leading 1), so the 5-bit number $10101_2 = 21_{10}$ is written as 001010101.

$127_{10} = 111'1111_2$ is the longest number whose bit count fits in three bits, and so it is delta-coded as 00'111'11'1111, or 001 1111 1111 in the traditional 4-bit groups. All larger numbers, and no smaller numbers, have three zeroes at the beginning. So the probability distribution for which delta coding is optimal is one where numbers larger than 127 are ⅛ of the total universe of numbers, numbers larger than 65'535 are half of that, numbers larger than 4'294'967'296 half of that (one out of every 32 numbers), numbers larger than 18'446'744'073'709'551'616 half of that (one out of every 64 numbers), and so on.

1 is delta-coded as 1: zero 0s indicating a 1-bit length field, which is 1, followed by the number in binary, omitting its leading 1, which leaves the empty string. 2 is delta-coded as 0100, and 3 as 0101, and that's all the 4-bit numbers. Then 4 is delta-coded as 01100, five bits, and thus up to 01111, 7. That's everything that begins with 01, 10, or 11, thus implicitly ¾ of all numbers, and then we're into the 001 territory that takes us all the way to 127.

So even though Elias delta coding is able to handle very large numbers with moderate overhead over fixed-width binary (unlike Elias gamma coding, which uses double), it squanders ¾ of the probability mass on numbers 1 to 7 inclusive, which is not helpful for our goal of representing coordinate pairs at 64 bits per second, which almost guarantees that often we'll want to encode relative coordinates in 4 or 5 bits.

Jointly encoding pairs

The Elias coding discussion didn't even consider where we're going to stuff the multiplier implied by the rounding, the one we earlier described as the exponent field of a floating-point format.

Fitts's Law suggests that the multiplier is almost independent of the resolution of the final result, in the sense that you could reasonably want 4-bit precision with a multiplier of 1, 2, 4, 8, 16, 32, or 64. However, there are a couple of dependencies. If you have 4-bit

precision, you don't really need all those options; 1, 4, 16, or 64 would work just as well, at the cost of reducing your effective precision to 3 bits at times. Also, on a pixel screen you probably don't want a multiplier of 1024 and a mantissa of 15, or for that matter a multiplier of 0.125 (though zooming in to clean up drawings may be useful at times). Moreover, if you have a fairly precise point where the mouse lingered long enough to give 8-bit precision, you don't really need multipliers like 32 or 64. And for compression it would be convenient to be able to make the ranges of the different precisions nonoverlapping, in the way the implied leading 1 does in non-denormalized floating point. These interactions all seem too complicated to find a simple solution to right now, though, so I'm just noting them.

Suppose we code the (logarithm of the) multiplier, shared between ΔX and ΔY, in a three-bit field, meaning a power of 4 between 1 and 16384; and we have another three-bit field (biased by 2) for the length of the ΔX and ΔY fields, represented in 2's-complement. Then a minimally precise data point would be something like 010'000'00'01, which means +4 in the Y direction, +0 in X, and that's 10 bits. The size field 000 means one data bit per coordinate, and the 10-bit data with this size field form a family of 4×4 lattices of exponentially varying sizes, covering the points (-2, -2), (-2, -1), (-2, 0), (-2, +1), (-1, -2), … (+1, +1), multiplied by their respective multipliers. All have the same (0, 0) point redundantly encoded.

These 10-bit-coded pairs have their worst-case error at vectors like (+2.5, +2.5), which is in between the (+1, 0) and (+1, +1) of the smaller lattice and the (+4, 0) and (+4, +4) of the larger lattice. This vector would be thus encoded as (+1, +1) with an error of 2.12 units, 60% of its magnitude; this level of error would be justified for movements so fast that they couldn't hit a target whose D/W was more than about 1.2, which is to say movements of under a + 1.2b, which I estimated above as about 66 ms. This gives a worst-case bandwidth for this encoding of about 160 bits per second, six times better than scrawling in Inkscape but almost three times worse than desired.

If we can manage to encode points less frequently, as the 250 ms example I mentioned earlier, we can hit 64 bits per second with 16 bits per pair. Those 16 bits might look like 000'011'00110'11011, which would be +6 in X, -5 in Y. The lattices of 16-bit-representable vectors overlap greatly, eliminating the holes in the 10-bit-representable values, and the worst-case relative error is $\sqrt{2}/32$, about one part in 45, i.e., D/W ≈ 45, so a + 5b movement time. With my pen handwriting guess figures, that would be 180 ms, but the four times worse time with Inkscape suggests more like 720 ms. So at this level our bandwidth is in the ballpark.

(Is three bits of $\log_4$ for the multiplier excessive? If we only had two bits of this exponent, the multipliers would be 1, 4, 16, and 64, which last would need only 5 bits of mantissa to reach the edge of the screen, and we'd improve worst-case bandwidth by 10%. Variable-length mantissas give us an escape hatch, though here they only allow us up to ±256.)

Successive approximations by alternately zooming in and out

What if we represented these vectors not with a single data point but a series of movements? If we're looking at a square picture divided into nine subpicture, we could indicate which subpicture to zoom in on with a number from 1 to 9, after which we can make another move, or we could indicate that we want to stop zooming with the number 0. This gives a base-10 prefix code that uniquely identifies any arbitrary recursively-divided square node.

The same digits in reverse indicate how to get back to the original viewpoint from the zoomed-in viewpoint, so such a code can also represent an arbitrary zoom out rather than in.

So if you want to indicate a sequence of points coupled with zoom levels, relative to some reference point, you could use a sequence of pairs of such codes: one to zoom out, then another to zoom in to the destination. The null movement case is 00; zooming in to the upper-right corner is, perhaps, 090; zooming out by one unit and then in to the lower-right corner is, perhaps, 5030.

Each digit takes 3.32 bits, so the minimal (null) movement is 6.64 bits, while the four-digit example "5030" is 13.28 bits. A movement that ends in a single zoom has a relative error of 0.5 either direction, i.e., D/W = 1, so the time is a + b, and each additional zoom (which must eventually be undone) gives you a factor of 3 (1.58 bits) extra precision in both X and Y, i.e., each extra 1.58b of MT adds 6.64 bits, or 4.2 bits per b, which is reasonably close to the 2 to 5 bits per b minimally needed. At my pen-based estimate of b = 30 ms, that's 140 bits per second, but again the Inkscape results being about four times slower would give more like 35 bits per second.

Maybe a better approach here might be just to specify the number of levels to zoom out, rather than the specific direction to do it in, which doesn't matter very much.

Quite aside from the problem considered here of encoding a mouse selection in a data file, this approach could be used to encode a mouse selection on the keyboard, too, and it would enjoy Fitts's-Law-like efficiency properties, which the conventional mouse-simulation approach of moving the pointer with arrow keys definitely does not. Maybe it could even be faster than using a shitty mouse. When I worked in a call center a quarter century ago, I was pretty quick on the keypad. I could always enter people's credit card numbers faster than I could convince them to say them. I haven't used keypads much since then, though, and in a quick typing test with typespeed(5) just now I was only able to reach about 1.49 digits per second; on a second trial I reached 1.66. which is about 5.5 bits per second, about the same as the 6-bit number I measured above for the mouse. (By comparison, typespeed measures me at 5.84 characters per second, 70 wpm, on English words; in actual text I'm closer to 90.) So maybe at least this wouldn't be much *slower* than an actual mouse.

Can we just punt to gzip somehow?

Dercuano is compressed for download as a gzipped tar file. What if, instead of coming up with an up-front hypothesis about the distribution of vectors in the drawings and then congealing it in a pile of bit-twiddling code, we just handle the rounding part, represent the vectors as bytes in some kind of straightforward way, and then let gzip handle the entropy-coding part? Gzip can also do things like recognize common patterns (if they repeat exactly) which we haven't

contemplated above. When it plows into the front end of the drawings, its entropy model is probably going to be attuned to HTML, but if all the drawings are together in the tar file, then it should have a pretty decent entropy model for drawings after a few kilobytes.

I don't know if this could actually work. I think the tricky part is really not the bit twiddling, but the decisions about which points should be in the "trellis" at each level of rounding.

Topics

Thredsnek: a tiny Python-flavored programming language

Kragen Javier Sitaker, 2017-03-20 (7 minutes)

I was thinking about making a tiny Python-like language, called Threadsnake or Thredsnek for the time being. The idea is to implement as much as possible in, say, 1.5KiB or 3KiB or 6KiB of code.

I think the most important linguistic reasons current Python is so effective are the following:

• The memory model is the Lisp-style object-graph, a garbage-collected heap.
• A small number of general-purpose, flexible-sized data structures (containers): specifically hash tables ("dicts") and dynamic arrays ("lists"). Pervasive mutability is key to Python's current flavor, but it clashes with the object-graph memory model in a lot of ways, so I think we could probably drop it. (Python also has tuples, sets, frozensets, namedtuples, deques, Counters, OrderedDicts, and defaultdicts, but these are less necessary.)
• "Duck typing": interactions between objects are indirected through interfaces, and so it generally doesn't matter what concrete type an object is, just what interfaces or protocols it supports. So, for example, keys in the built-in hash tables need not be strings; they can be any object that supports the `__hash__` and `__eq__` methods. (The original term for this was "object orientation", "message passing", or "late binding", but people got confused about what those terms meant; "duck typing" is the current term.) In particular, any code can define its own "types" which are really just wrappers that provide a duck-typed façade to dicts.
• As a special case, iteration indirects through an "iterator protocol" which allows for composable iterators, and there is a coroutine facility for this ("generators") which can be put to many different uses. (I am sympathetic to Alexandrescu's argument that maybe we should use a "range protocol" instead of an iterator protocol for this.)
• Errors are handled with exceptions. Ambiguity and implicitness is an error; there is very little DWIM. (Implicit variable declaration is an exception to this, and was probably a mistake.) Consequently, if your program runs and produces an answer instead of crashing, you can have a fair bit of confidence that the answer is correct.
• Reflection allows the in-language implementation of debugging facilities and transparent and semitransparent persistence facilities like ORMs.
• Tasteful and largish standard library ("batteries included"). It's very easy to, for example, run unit tests, split strings on whitespace, join together many strings with a separator (such as a space), concatenate strings, do formatted string interpolation, do integer and floating-point arithmetic, read a line from a file, read an entire file into memory, parse RFC-822 syntax, parse or emit JSON or XML or

CSV, do a substring search, extract a substring by indices, lowercase a string, remove leading and/or trailing whitespace from strings, display an unambiguous debugging representation of Python objects in HTML or in text, run binary search, round a number to a given number of decimals, search strings for regular expressions, collate the number of occurrences of each item in an iterable sequence and return the top items, sort and reverse lists, encode and decode UTF-8 and other common character encodings, and so on.

This is not as poetic as Tim Peters's *The Zen of Python*, but I think it might be more helpful.

Duck typing in particular means that all your data structures (heaps, sorted lists, whatever) and algorithms (sort algorithms, graph algorithms, whatever) are automatically polymorphic over whatever types are available, and that you can substitute persistent containers (in the disk sense) for nonpersistent ones transparently.

Python has some major shortcomings. The semantics of its procedure and class definitions make live code upgrade impossible; its efficiency in general is terrible; its semantics are so flexible that many errors cannot be caught until runtime even in principle; its absence of first-class lambdas or Smalltalk-style blocks has given rise to a steadily growing pile of kludges such as method decorators and context managers. But it remains immensely popular and immensely practical.

And this is in spite of its lack of many things commonly considered critically important in a programming language: encapsulation, structs, nested scoping (at least in old versions), a decent GUI library...

It's clearly impossible to implement a sizable standard library in a few K of code. But how much code would Thredsnek need to implement garbage collection, dicts and lists, duck typing (including for iteration), exceptions, and reflection? On modern machines, it would still be useful even with fairly inefficient approaches to these problems.

What kind of abstract machine would be needed to implement this? You need to be able to send messages (invoke methods), store and retrieve local variables (including parameters), retrieve constants, raise and catch exceptions, instantiate objects (including dicts and lists; possibly by sending messages to a class), and do reflection (possibly by sending messages to some kind of reflection object, which seems preferable to invoking magic non-implementable reflection messages on arbitrary objects). You probably also need control-flow and method-return operations.

For variable-length argument lists, like those used for instantiating lists or dictionaries, it might be useful to begin by pushing a stack mark and pass everything down to that mark as arguments.

As a point of reference, the Smalltalk-78 virtual machine for the NoteTaker was supposedly 6KiB of 8086 machine code. This used the Smalltalk-76 bytecode, which was simpler than the Smalltalk-80; according to Ingalls's 1978 paper on it, its high nibble selected the overall category of operation:

- 0x1x: load an instance field (implicitly limited to 16);
- 0x2x: load a local variable;
- 0x3x: load from the table of constants for this method;

• 0x4x: load *indirectly* from the table of constants (I think this was used for accessing classes and other global variables);
• 0x5x: load from a Context field (I don't understand this);
• 0x6x: load one of the universally common constants, -1, 0, 1, 2, 10, true, false, and nil;
• 0x7x: sends a message to the Context, but I don't know what for;
• 0x8x: sends one of SpecialMessages, which I assume are things like + and =;
• 0x90–0x97: jump forward up to 8 bytecodes unconditionally;
• 0x98–0x9f: jump forward up to 8 bytecodes if false;
• 0xa0 0xxx – 0xa7 0xxx: jump unconditionally up to 768 bytes back or 2047 forward;
• 0xa8 0xxx – 0xaf 0xxx: jump conditionally (by the same amount);
• 0xb0: discard stack top;
• 0xb1 0xxx: copy stack top into given location;
• 0xb2 0xxx: move stack top into given location (and pop it);
• 0xb3: return stack top as method return value.

The bytes denoting locations to store into were the same as the load opcodes.

Topics

The economics of solar energy

Kragen Javier Sitaker, 2008 (27 minutes)

Item #1 on the National Academy of Engineering's list of the most important engineering challenges of this century:

• Make solar energy economical. Solar energy provides less than 1% of the world's total energy, but it has the potential to provide much, much more.

Calculations of Limits

The detail page claims that the blurb is wrong; only 0.001% of the world's total energy usage is sold in the market, of which a fraction is fossil fuel. Additionally, although it's not mentioned in the page, many of the world's poorest people live entirely off of solar energy. It's just that the other 99.999% of the energy isn't easily used to drive mechanical engines at the moment.

Some of it is captured by plants in sugars and cellulose through photosynthesis, and that is called the planet's "net primary production". Peter Vitousek and others estimated in 1986 ("Human Appropriation of the Products of Photosynthesis") that humans consume as food or firewood 3% of its net primary production (including food for livestock) totaling 7.2 Pg/year of dry biomass, and "co-opt" another 19%, out of a total of 132.1 Pg/year.

7.2 Pg of carbohydrates is about 1.5 * 10^20 joules. 132.1 Pg of carbohydrates is about 27.7 * 10^20 joules. The US DoE EIA International Energy Outlook 2007 (PDF) reports that "world marketed energy consumption" was 447 quadrillion Btu in 2004, which is about 4.7 * 10^20 joules. This figure doesn't appear to include food sales.

A crude calculation (earthradius_equatorial^2 * pi * (1000 W/m^2) * 1 year in `units(1)` --- gosh, Unix is great!) suggests that the total solar energy falling on the earth is about 40000 * 10^20 joules per year. If Vitousek's figures are right, that means that the earth's ecosystem is about 0.07% efficient at converting sunlight into biomass, and therefore probably not more than 1.5% efficient at converting sunlight into anything but heat.

This leads to the conclusion that "world marketed energy consumption" is about 17% of the planet's net primary production, assuming that figure hasn't changed much since 1979, but that the planet's net primary production is only 0.007% of the energy received from the sun by the Earth. 17% of 0.007% is indeed about 0.001%, the figure from the original blurb.

At some point we'll have self-reproducing hardware, and the cost of manufacturing the solar cells will cease to be a problem; it will simply be a question of land use. To supply the 4.7 * 10^20 joules per year being currently sold in the market with the 40%-efficient solar cells in some labs today, we'd need to capture 11.75 * 10^20 joules per year of sunlight, which is 0.03% of the surface of the earth, including oceans. Matthias Loster made a lovely visualization of this and put it on Wikipedia.

If you were using 4%-efficient solar cells instead, you'd need 0.3% of the earth's surface area. Off-the-shelf inexpensive thin-film solar

panels are about 8% XXX efficient.

Calculations of Growth Rates

In 2005, the National Renewable Energy Laboratory published a FAQ on energy payback times of photovoltaic cells, which explained that at the time, multicrystalline photovoltaic cells produced more energy than had been used to make them in 2–4 years. Here are their references; I haven't read any of them:

E. Alsema, "Energy Requirements and CO_2 Mitigation Potential of PV Systems," Photovoltaics and the Environment, Keystone, CO. Workshop Proceedings, July 1998.

R. Dones; R. Frischknecht, "Life Cycle Assessment of Photovoltaic Systems: Results of Swiss Studies on Energy Chains." Appendix B-9. Environmental Aspects of PV Power Systems. Utrecht, The Netherlands: Utrecht University, Report Number 97072, 1997.

K. Kato; A. Murata; K. Sakuta, "Energy Payback Time and Life-Cycle CO_2 Emission of Residential PV Power System with Silicon PV Module." Appendix B-8. Environmental Aspects of PV Power Systems. Utrecht, The Netherlands: Utrecht University, Report Number 97072, 1997.

K. Knapp; T.L. Jester, "An Empirical Perspective on the Energy Payback Time for PV Modules." Solar 2000 Conference, Madison, WI, June 16–21, 2000.

J. Mason, "Life Cycle Analysis of a Field, Grid-Connected, Multi-Crystalline PV Plant: A Case Study of Tucson Electric Power's Springerville PV Plant." Final report prepared for Tucson Electric Power, November 2004.

W. Palz.; H. Zibetta, "Energy Payback Time of Photovoltaic Modules." International Journal of Solar Energy Volume 10, Number 3-4, pp. 211–216, 1991.

The payback time is a representation of a sort of compound interest rate; a payback time of 2 years is a 50% annual percentage growth, 4 years is 25%. If you spend the produced energy on making new solar cells, those are the actual growth rates of your stock of solar cells (plus a bonus, in the form of compounding the interest more frequently, if you get the new cells into production in less than a year).

So consider Evergreen Solar's current situation. They're a small company with a market capitalization of US$917M as of their last annual report. Their current manufacturing capacity is 15 megawatts per year, and they've contracted to manufacture 125 megawatts in 2009, 300 megawatts in 2010, 600 megawatts in 2011, and 850 megawatts in 2012. Suppose a company of similar size were to invest its 300-megawatt 2010 production merely in making more solar cells, and that it had no non-energy costs. An annual growth rate of 50% --- that is, a 2-year payback --- would look like this:

```
In [3]: ["%d: %.1fMW" % (2010 + x, 300*1.5**x) for x in range(30)]
Out[3]:
['2010: 300.0MW',
 '2011: 450.0MW',
 '2012: 675.0MW',
 '2013: 1012.5MW',
 '2014: 1518.8MW',
 '2015: 2278.1MW',
 '2016: 3417.2MW',
 '2017: 5125.8MW',
 '2018: 7688.7MW',
 '2019: 11533.0MW',
 '2020: 17299.5MW',
 '2021: 25949.3MW',
 '2022: 38923.9MW',
```

```
'2023: 58385.9MW',
'2024: 87578.8MW',
'2025: 131368.2MW',
'2026: 197052.3MW',
'2027: 295578.4MW',
'2028: 443367.6MW',
'2029: 665051.3MW',
'2030: 997577.0MW',
'2031: 1496365.5MW',
'2032: 2244548.3MW',
'2033: 3366822.4MW',
'2034: 5050233.7MW',
'2035: 7575350.5MW',
'2036: 11363025.7MW',
'2037: 17044538.6MW',
'2038: 25566807.9MW',
'2039: 38350211.8MW']
```

The current 4.7 * 10^20 joules/year being sold in the market is 14 893 719 megawatts, which this curve crosses around 2037, and with the usual 5:1 ratio between peak watts and achieved watts (due to nighttime, solar angle changes, clouds, etc.) you don't reach it until 2041 or 2042. If you start with the 1744 megawatts that Evergreen says Solarbuzz said constituted the global solar power market in 2006, you gain about 8 years.

However, the *financial* payback time on solar panels is still dramatically longer, which is why they still haven't reached "grid parity" --- costing less per watt-hour than power from the grid. Solar panels still cost US$4-$5 per peak watt at retail. That's about US$20 per average watt, which is 8760 watt-hours per year; that's about US$0.87 of electricity at grid rates. That's a 23-year *financial* payback, and that doesn't include things like batteries, inverters, wiring, and installation.

Evergreen Solar

Evergreen's annual report suggests some reasons for this: the market is expanding at 42% per year, their own production capacity has to expand by a factor of more than 50 from 2007 to 2012 (this from a company that's already 13 years old). They had 276 full-time employees in manufacturing to reach their 15-megawatt-per-year capacity: 18 employee-years per megawatt. Their new 80-megawatt-per-year facility is expected to require another 410 employees (5 employee-years per megawatt). They are struggling to increase manufacturing capacity fast enough to keep up with demand, and apparently so are their "polysilicon" suppliers, because there's an industrywide shortage of polysilicon; building new polysilicon manufacturing facilities takes several years.

(I'm a little bit dubious about their terminology; I think the company's management may just not be very technical, or maybe not very smart. "Polysilicon" is short for "polycrystalline silicon", and silicon becomes polysilicon at the point that you crystallize it in a polycrystalline form in your furnaces. So the suppliers are supplying Evergreen with silicon; how many crystals are in each piece of silicon they supply is somewhat immaterial, since Evergreen melts the silicon

down and crystallizes it in polycrystalline silicon ribbons in their furnaces.)

They report that they had US$58M of product revenues in 2007, with US$53M "cost of revenue", which presumably includes things like manufacturing employee salaries, energy, and raw materials. They spent US$21M on research and development and another US$21M on "Selling, general, and administrative", and US$1.4M on "facility start-up", building a new plant to increase their manufacturing capacity from 15MW to 95MW this year.

So suppose they manufactured 15MW in 2007, as their annual report suggests. That would mean they got paid US$3.87 per watt on average, which is more or less in keeping with the US$4-$5 the panels cost at retail. They spent US$3.53 of that on their actual manufacturing costs. They explain:

> The main purpose of our Marlboro facility [where all of their manufacturing currently takes place] is to develop and prototype new manufacturing process technologies which, when developed, will be employed in new factories. As such, our manufacturing costs incurred in Marlboro are substantially burdened by additional engineering costs and also reflect inefficiencies typically inherent in pilot and development operations.

Elsewhere they explain that they use about 5 grams of silicon per watt; metallurgical-grade silicon costs about US$0.77 per pound , or US$0.0017/g.

They don't break out the costs of the silicon they buy from their suppliers, which might cost considerably more than the metallurgical-grade silicon it's made from. It appears that they have made prepayments and cash loans of, as I read it, about US$50M on a set of multi-year silicon supply contracts, although they only list US$23M in their "prepaid cost of inventory" line item; and elsewhere they say, "we have silicon under contract to reach annual production levels of approximately 125MW in 2009, 300MW in 2010, 600MW in 2011, and 850MW in 2012", for a total of 1875MW; and they say, "We believe future enhancements to our technology will enable us to gradually reduce our silicon consumption [from 5g/W] to approximately two-and-a-half grams per watt by 2012."

So suppose those 1875MW are to be made at an average of 3.5 g/W; that's 6600 million grams of silicon. And suppose the US$50M represents about half of the total price of that silicon; that would give us US$0.015 per gram of silicon. That's more or less in line with the raw silicon cost I estimated above for metallurgical-grade silicon --- it's higher, and by a factor of only about 2 --- which gives me some confidence that my guesstimate that the cost of the silicon is not yet a significant factor in the cost of the solar cells.

However, note that securing one of these long-term silicon supply agreements required selling about 15% of the company to the silicon supplier. The restrictions on that stock "will lapse upon the delivery of 500 metric tons of polysilicon to the Company", so we can guess that the total contract with that supplier is in the neighborhood of 1000-2000 million grams.

Also, they list $629M in "raw materials purchase commitments" among their "contractual cash obligations". This, plus the prepayments, is perhaps a ceiling on the amount of payment they may have committed to for the silicon; US$679M for 6600 million grams of silicon would be US$0.10 per gram, which would raise the cost of

silicon above from US$0.008 per watt to US$0.35 per watt. (It's possible that they have other raw materials purchase commitments, say for silane or hydrofluoric acid.)

In 2006, Evergreen and EverQ bought US$8M worth of silicon from REC, who I think was their sole silicon supplier at the time (unless DC Chemical was also a supplier?). During that year, they had $102K of sales. In 2007, Evergreen bought US$3M worth from REC, which is US$0.20 per watt if they produced 15MW.

Let's assume that their per-employee cost of labor on the factory floor is about US$120 000 per year. At 18 employee-years per megawatt, that's about US$2.2M per megawatt, or about US$2.20 per watt.

If we assume that the NREL numbers are applicable to their manufacturing, then each peak watt of panels required about 4kWh of energy; let's assume that costs US$0.10/kWh. So, per watt, we have:

revenue	US$3.87
gross profit	US$0.34
electricity	US$0.40
raw silicon	US$0.008
labor	US$2.20
OTHER	US$0.92

At present, they're also spending about half of their revenue on research and development. (That's part of why they're still losing money.) We can expect that the cost of labor per watt will decrease substantially in their 80MW non-pilot facility: 5 employee-years per megawatt would be US$0.60 per watt.

They also have been spending on the order of US$50M per year on capital expenditures, mostly equipment and facilities improvements. They report that their "fixed assets, net" are worth US$115M, including US$53M of "laboratory and manufacturing equipment", US$14M of "leasehold improvements", and US$67M of "assets under construction". They seem to expect that constructing the first 80MW/y production line in their new facility will cost around US$100M, although they don't really break it out that way in the report. That's about US$1.25/watt/year.

A capital cost of US$1.25 per watt/year of manufacturing capacity does not unavoidably contribute much to the cost per watt; after all, you can in principle amortize it over an arbitrary number of years. However, in an industry with a 42% annual growth rate, almost all cells will necessarily have come out of factories built within the last year or two, so it probably adds US$0.60/watt or more to the cost of the cells.

EverQ, a separate company that Evergreen owns a third of, had operating revenue of US$194M, cost of goods sold of US$160M, "other expenses" of US$27M, and assets of US$556M. I wish I had handy EverQ's manufacturing capacity numbers.

Nanosolar

Nanosolar claims an energy payback time of one month and a per-watt cost of 30 cents with their copper indium gallium diselenide

thin-film cells, in a November 2007 article on Celsias, although they had expected a cost in the sixties of cents per watt in a July 2007 interview. In the Celsias article, they also say they plan to reach 430 megawatts of production per year in 2008.

In the interview, CEO Martin Roscheisen also says:

> ...it is clear we are going to be manufacturing capacity limited for about as far out as we can see. There's presently really only two truly scalable solar markets in the world — Germany and Spain — and we do a lot there. Being a scalable market is today as much about feed-in-tariffs as about the administrative framework; tomorrow, with grid-parity PV systems, it is primarily about the latter.

Material Shortages

As I said before, Evergreen is experiencing an industrywide polysilicon shortage; however, the raw material silicon is extremely abundant, being the principal component of one of the most common minerals in the terrestrial crust.

However, the materials used in copper indium gallium diselenide (CIGS) thin-film cells like Nanosolar's are somewhat less abundant. Copper has been a precious metal since the Bronze Age, but indium, gallium, and selenium are all fairly rare.

As a point of comparison, after years of rapid increase, silver prices averaged US$13.40 per troy ounce in 2007, according to the USGS's silver report. That's US$430 per kilogram. About 20 700 tons of silver were mined in 2007.

Indium, by contrast, cost US$795 per kg in 2007, and averaged an even higher US$918 per kg in 2006, and only 510 tons were refined in 2007, making it 40 times rarer than silver and 85% more expensive. The USGS claims, "Thin-film ... CIGS solar cells require approximately 50 metric tons of indium to produce 1 gigawatt of solar power," which still makes it a tiny fraction of the total cost. (I am assuming the USGS is referring to peak watts at one sun, i.e. in direct sunlight without lenses or mirrors, and not average output or solar-concentrator output.) That's US$0.04 of indium per watt, so the price of indium would have to increase by a factor of 75 to increase the cost of thin-film cells by US$3 per watt. That would be about US$60 000 per kg. I think grid parity is somewhere around US$1 per watt, which would be around US$20 000 per kg.

At higher prices, you would expect new low-concentration sources of indium to become economic to refine, which would be nice, because current world indium production is only enough for about 10 gigawatts of CIGS per year. It's difficult to predict what kinds of improvements could occur and how much they could increase indium production. However, we can get a little bit of a clue by looking at the last several years. In 2002, indium cost only US$130 per kilogram, so we've already experienced a dramatic price increase, driven by dramatically increased production of LCD displays, which use indium tin oxide for thin-film transparent electrodes. So how much did indium production increase when the price increased by a factor of seven over four years? It increased from 335 tons to 510 tons. [XXX check that. probably slightly wrong.]

So, although it's error-prone to predict, the evidence suggests that indium production capacity will prove quite difficult to scale up over the next several years, which could limit CIGS thin-film solar cells to a small fraction of the overall energy market.

Gallium is only slightly more expensive than silver, at US$460 per kg. Supplies of gallium are even more limited than those of indium; the USGS report *estimates* world primary gallium production capacity at 184 metric tons per year, and actual production at 80 metric tons per year, making it 250 times rarer than silver. In the absence of the LCD demand that has caused indium's price to skyrocket over the last several years, its price has remained relatively constant 2002–2007 even as imports have more than doubled. This would seem to suggest that gallium's production could be increased considerably more easily than indium, but I suspect that this is not the case, as I explain below.

The gallium prices are stated for extremely pure gallium, with less than 0.1ppm impurities, because this is what is needed for its largest-volume use, high-performance integrated circuits made of gallium arsenide, largely for RF components in cell phones. The USGS also reports some information on "low-grade" 99.99% pure gallium:

> Prices for low-grade (99.99%-pure) gallium increased in the first half of 2007 from $300 to $350 per kilogram at the beginning of the year to about $500 per kilogram by midyear. Producers in China claimed that there was a shortage of supply, which was the principal reason for the increase in prices. Some were offering gallium at prices as high as $800 per kilogram, but little business was completed at this price level.

The reason I think gallium production will hit limits similar to indium production is that indium and gallium are chemically very similar, and they are both primarily refined from trace amounts (50 ppm or more, at present) found in zinc ores and bauxite, and consequently they are found as impurities in zinc. So I think it is unlikely that there are large amounts of easily recoverable gallium hiding somewhere without corresponding amounts of indium accompanying them.

Because of their chemical similarity, they are substitutable for one another in some semiconductor applications.

I believe CIGS contains equal numbers of atoms of indium and gallium, but I think the gallium is somewhat heavier. XXX I need to look at a fucking periodic table.

Selenium is also only found in trace amounts in the Earth's crust. I don't know how much it costs or how much is being mined.

Silicon solar cells are made from silicon, arsenic, boron XXX, and aluminum --- some of the most common elements on Earth. However, their processing XXX

Solar Concentrators

Everything above --- costs per watt, factory production capacities in watts, materials per watt, etc. --- is about solar cells in "one sun", i.e. the intensity of sunlight that naturally reaches the surface of the Earth, which is about 1000 W/m². Silicon photovoltaic cells can theoretically turn up to 31% of that into electricity, but the less expensive polycrystalline cells in common use are only about 12% efficient, with even lower efficiencies of 9–12% or so for thin-film cells and 6% for amorphous silicon cells. There are more expensive "multijunction" non-silicon cells available for sale now that are 34%

efficient, and 41%-efficient cells in laboratories that will presumably reach production soon; and there are quantum-dot and photonic-crystal approaches that could reach 60% in theory. (Some of these numbers are from the NREL report cited earlier, while others are from the National Academy of Engineering page cited earlier.)

However, these more-watts-per-unit-area approaches are very expensive per watt, so they are currently mostly only used in space missions --- to power satellites and the like.

Most types of photovoltaic cells continue to work in higher-intensity light, even working at higher efficiencies [XXX]. If you have mirrors that cost less per square meter than your solar cells, you can use mirrors to gather the same amount of sunlight onto a smaller area of expensive solar cell, for a lower overall system cost. This sort of thing is called a "solar concentrator", and there are some very-large-scale systems that don't even use photovoltaic cells at the focal point, instead using heat engines like an old-time locomotive, which can be more efficient at sufficiently high temperatures.

One experimental project uses a balloon, half of aluminized mylar, half of transparent mylar, to make a concave reflector for a small photovoltaic panel. In photos, it looks like it generates about "100 suns", or 100 times the normal intensity of sunlight. This means that "one watt" of solar panels, rated according to normal sunlight, can produce 100 watts or a little more [XXX confirm this], with the aid of a square meter or so of aluminized mylar, which costs on the order of US$2, and can be recovered abundantly from garbage in many areas. However, I suspect it needs some special cooling [XXX check this].

This kind of setup could theoretically be quite inexpensive and sturdy, but there are difficulties. Your hundred-sun system will suddenly become a zero-sun system if it's not pointed fairly accurately at the sun, so it requires control motors to follow the sun across the sky; this adds to the cost, and also reduces reliability. Your balloons will eventually deflate, and you have to reinflate them. And on cloudy days, your hundred-sun system is, at best, a one-sun system. So most of the production solar concentrators I've heard of have been large-scale thermal generators.

If your 1m² concentrating mirror cost US$5, your 100cm² 12%-efficient photovoltaic cell cost US$5, your motors and control system cost another US$20, and your cooling cost another US$20, you'd have a US$50 system producing about 120 watts, or about US$0.42 per watt. If you could upgrade to 24% efficient cells that cost another US$10 (I have no idea if this price is realistic), you'd have a US$60 system producing about 240 watts, or about US$0.25 per watt --- even though the solar-cell component of the system cost 50% more per watt, the system as a whole cost less per watt. In this way, photovoltaic concentrator systems can economically take advantage of more expensive photovoltaic materials, as long as the solar cells themselves are a small part of the cost of the system.

You would think that this kind of technology would have been adopted wholesale long ago, since it would appear to cost dramatically less per watt than fossil-fuel plants, not even counting the cost of the fuels. So there must be some difficulties that have prevented it from achieving the kind of efficiencies I've suggested above, at least scalably.

There are various experimental systems working on this principle: Solient's (see also the Technology Review article),

In summary: photovoltaic solar concentrators could, in theory, provide electrical generating capacity for US$0.05--US$0.50 per watt with current technology, and I don't know of any practical reason this potential won't be realized. But I also don't know why it hasn't already been realized, say ten or fifteen years ago, and there must be a reason; and maybe that reason still applies.

Forecasts

Topics

Incremental recomputation

Kragen Javier Sitaker, 2018-04-27 (12 minutes)

Let's solve cache invalidation, legendarily one of the two hardest problems in computer science.

Why is it hard? We can construct a hard example in three lines of code.

Suppose we construct a unidirectional data flow graph as follows:

```
x, y, z = Input(value=3), Input(value=4), Input(value=2)
p = Max(x, y)
for i in range(10000): p = Max(p, Subtract(p, z))
```

This constructs a directed acyclic graph of dataflow with 20004 nodes: three Input nodes, 10001 Max nodes, and 10000 Subtract nodes. If we increase y's value to 5, this changes the value of all of the Max nodes to 5 and all of the Subtract nodes to 3. By contrast, if we increase or decrease the value of x by 1, no other nodes have new values. How can we implement this efficiently?

There are a variety of different strategies you can use to evaluate such a dataflow graph, but most of them have serious pitfalls which can be demonstrated on the above graph.

To keep the scope of this note manageable, I'm only writing about algorithms for *static* dataflow graphs here, where the topology of the computation is independent of the data that flows through it, like an electronic circuit. To some extent, you can force conditionals into a static dataflow framework as conditional or multiplexing nodes, and iteration as array-valued values (as in TensorFlow), so this is somewhat less restrictive than it sounds.

Pure immediate mode

The simplest strategy is to calculate values on demand. To be concrete, you might have an `eval` method of no arguments which invokes the `eval` methods of the node's inputs:

```
def eval(self):  # for Max nodes
    return max(self.input1.eval(), self.input2.eval())
```

This is pretty much the approach I use in Pytebeat, for example, with values being arrays of 1024 numbers. On some dataflow graphs, this works fine, but on this one, it requires $4 + 2^{10001}$ calls to `eval`, which on my laptop would take about 10^{2995} years, at which point all the stars will have gone out and essentially all matter will be either iron or leptons, depending on whether protons are stable; this is not a practical way to evaluate this dataflow graph. The issue is that nodes whose output is used more than once will be evaluated more than once, which is exponential-time in such pathological cases.

(In Python it actually fails sooner than that by default because the default stack limit is not high enough.)

Topologically-sorted immediate mode

You can calculate values on demand in guaranteed linear time,

though. You begin by doing a topological sort of the nodes to construct a sequence such that each node comes before the nodes that use its value; then you can simply iterate over the sequence, storing the output of each node in an array or something.

This allows you to do the calculation several times for different input values without repeating the topological sort. In effect, you have compiled the dataflow graph into a branchless virtual machine program with a certain flavor of common subexpression elimination.

But, if you think about the standard topological sort algorithm, you will see that you could also do the equivalent of interpretation — you can evaluate the value for each node as you generate the topologically-sorted sequence.

(As it happens, in the example code, the nodes are necessarily created in a topologically-sorted sequence, because each node is constructed with its inputs as parameters. So in this case you can simply define `Max, Subtract, Input = max, operator.sub, lambda value: value` and never construct the graph in memory at all.)

Either way, this approach only touches each node once; on this graph it takes perhaps 100000 function calls, or about a millisecond for this graph, if you do it in Python with an existing in-memory graph.

This doesn't sound so bad, and in the case where you are evaluating some dataflow graph on all new inputs you've never seen before it's absolutely optimal and everything else we'll discuss in this note is slower, but in the case where you just changed x by 1, it takes 100000 function calls to tell you that nothing else has changed as a result. That's about three to five orders of magnitude slower than would seem reasonable. So let's look at incremental approaches to dataflow evaluation.

Invalidation propagation

I wrote about how you could treat the topologically-sorted sequence of operations as a sort of program. But this suggests that you could backtrack in it: if you have 50 operations, and input A isn't used before the 25th operation, the first 25 are guaranteed not to change their value just because input A changed. So you could just start the execution from operation 25, using the previously calculated values for operations 0–24, and then you don't have to do a bunch of redundant recalculation of values that are guaranteed not to have changed.

But this actually works a lot better with the "intepretive" topological sort, because it doesn't have to arbitrarily pick an ordering among nodes that have no actual data dependencies, so it can evaluate only nodes that changed data feeds into.

The way this looks is that changing an input works in two steps: first, you "retract" the current value, causing all the nodes that depend on it to also retract their current values; then, you assert a new value, causing all the nodes that depend on it to be recomputed. If a node gets an input asserted while some other input whose value it needs is still retracted, its output remains retracted. Only once all the required inputs are asserted can it assert a value.

In many cases, you can improve performance by retracting several things at once and then asserting their new values, instead of changing their values one at a time.

This approach is pretty broadly applicable, but it doesn't help for

our example graph; when we retract x=3, the invalidation propagates through 20002 nodes, and then when we assert x=2, the same value propagates through those same nodes again. It takes about twice as long as just calculating all the values from scratch without trying to be incremental.

Pull-based invalidation, or laziness

As I've described it above, invalidation propagation is eager — when you initially assert values for x and y, that pushes the Max node to compute their max, which then pushes the first Subtract node to compute a zero, which then pushes another Max node to compute a value, and so on. Computation is initiated by inputs and results in outputs.

It may happen that the end result of all this computation is, for example, to display a number on the screen, which can only happen at most about 60 times a second, depending on your hardware. But suppose you are changing x or y much more often than this — most of the values computed will never be used.

Now, if the value computed has to be available on a tight deadline, you need some kind of eager computation to store it ahead of time instead of computing it on demand. But what if it's okay to compute it on demand?

In that case, you can have nodes assert their outputs only upon request. In the case of our example graph, first you construct the graph, but don't calculate values for any of the nodes except x and y. Then, you request the output from the final p node. Since its inputs are retracted, it requests them to compute; whichever input tries to compute first finds that its inputs are also retracted, so it asks one of them to compute; and so on, until we are requesting the output of Max(x, y). Its inputs are asserted, so it computes its result (4) and asserts it; this allows a Subtract node to compute its result (2) and assert it; and so on.

This is roughly two thirds as efficient as the push-based approach in this case, and when we retract x later, it does the same invalidation propagation. The difference is that, when we assert x again, no further computation happens — nothing is currently requesting x's value. We can change x or y many times in between screen frames, cheaply.

However, it still has to recompute all 20002 computational nodes in order to discover that changing x doesn't change the final output value.

Change propagation

What if we don't have a separate retraction step, but just assert the new value of x? This helps a lot in this case — the Max node that depends on x can detect that its output values remains 4, so it doesn't need to propagate any change.

But what happens when we change y? If we change y to 8, the Max node updates to 8. This value needs to be sent to the next Max node and the Subtract node. Suppose we pick the Subtract node. Now its value (y-z) is 6, so it updates its value to 6. This gets pushed to the next Max node, which is now comparing 6 to 5, and so it asserts 6 on its output, pushing to two nodes. Later on, when the new value from the previous Max node reaches it, it must recompute a

second time. This kind of thing can result in an exponential number of recomputations, just as with pure immediate mode.

XXX hmm, maybe I need a better example to show the potential exponential behavior, because it peters out here

XXX does breadth-first propagation solve the exponential-time problem?

There's another problem, though, besides efficiency: "glitches" or "timing hazards". That second Max node's output was 4 at the beginning and 8 at the end. But for a moment in the middle, after its first recomputation but before its second one, its output was 6. 6 was never a correct value for max(max(x, y), max(x, y) - z)! If that value change results in your software system taking some action, you could be in trouble.

This happens at a very concrete level in digital logic circuits, in which "glitch" is a technical term for this kind of transient wrong answer. There, the standard solution is a clock whose edges trigger flip-flops.

Memoization

As an alternative to storing the current value of each node, we could instead store some set of previously computed values, according to the inputs that affect them. In the most direct form, of course, applied to a single node, this doesn't help much — computing the max of 3 and 4 is faster than looking up (3, 4) in a cache to return 4. But we could imagine, for example, memoizing a larger chunk of the graph.

The potential benefit of memoization, as opposed to the strategies described above, is that if part of the graph returns to a previous state, the result of that previous state can be retrieved from the cache — it isn't just a matter of "state unchanged" or "state changed".

Chunking

In fact, chunking the graph into smaller subgraphs — giant nodes that perhaps produce multiple values — is an approach that can be applied to all of the previous

Reduced affine arithmetic

Delta propagation

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- Caching (p. 3361) (25 notes)
- Incremental computation (p. 3517) (24 notes)
- Graphs (p. 3486) (5 notes)
- Dataflow (p. 3401) (5 notes)

Honk development

Kragen Javier Sitaker, 2019-03-21 (2 minutes)

As I listened to a car honking its horn for a long period of time far away with no pause, I noticed that the quality of the sound changed subtly several times over the first second or two. I think this is a result of echoes: at first I heard only the incident horn, but after a short time it was joined by one, two, three, several echoes of the original horn. Depending on the particular details of the time delays, some of the harmonics in each echo interfered constructively with those in the original sound, while others interfered destructively.

Aside from what this implies about what we can learn about our built environment from analyzing the sound, it occurred to me as being a very easy effect to simulate; this took me about 15 minutes:

```
/* ./horn | aplay */
#include <stdio.h>
#include <stdint.h>

typedef uint8_t u8;

u8 wave(long long t)
{
  return (t & 128 ? 256 - (t & 255) : 128 + (t & 127)) >> 1;
}

u8 horn(long long t)
{
  enum { attack = 600 };
  int v = t < 0 ? 0 : t > attack ? 256 : t * t * 256 / attack / attack;
  return wave(t) * v >> 8;
}

int main()
{
  for (int a = 0; a < 32000; a++) {
    putchar(horn(a - 1000)
            + (horn(a - 6242) * 64 >> 8)
            + (horn(a - 8932) * 32 >> 8)
            + (horn(a - 12333) * 64 >> 8)
            + (horn(a - 3013) * 128 >> 8)
    );
  }
}
```

The waveform of the sound doesn't sound very much like a horn, but the changes in the tonal quality over time are similar to what I was hearing.

Topics

a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees)

Kragen Javier Sitaker, 2012-12-06 (updated 2013-05-17) (10 minutes)

Summary

There's an alternative to summed-area tables with a small, linear space cost and linear construction time and space, providing worst-case logarithmic-time reduction over arbitrary intervals under arbitrary semigroup operations, and which supports updates efficiently, unlike summed-area tables. The algorithm is like twenty fricking lines of code if you leave out the "update" and "small" parts.

I am surely not the original discoverer of any of this.

Introduction to abstract algebra

Abstract algebra is the study of what you can deduce from minimal sets of axioms about some set of things and operations on them. A lot of it seems to be taxonomic, assigning names to particular sets of axioms. This is cool because once you establish that, say, 32-bit bitstrings form a semilattice under the bitwise-OR operation, you can apply every theorem that anybody's ever proven about semilattices to 32-bit bitstrings and OR.

In particular, as I think Stepanov first realized, the correctness of an algorithm depends on these algebraic properties of the data it's manipulating. (Typically it also depends on operations being computable, and its efficiency depends on the complexity of that computation, which are perhaps unfortunately not within the purview of abstract algebra.)

A magma is a set associated with a binary operation that's closed over that set.

A semigroup is an associative magma. An example is the set of nonempty finite strings over some alphabet, with concatenation as the binary operation.

A semilattice is a semigroup whose operation is commutative and idempotent; in general a semilattice is a partially ordered set of some kind, with a unique smallest element, where the binary operation is the operation of finding the largest upper bound of the elements. Aside from the obvious examples of totally ordered sets like integers, things like 32-bit bitstrings under bitwise OR or AND form semilattices.

A monoid is a semigroup with identity. String concatenation is the usual example; the empty string is the identity element.

A group is a monoid where every element has inverse for every

element. It's sufficient to have a left inverse for every element; from that you can get identity (I think!) and right inverse.

Summed-area tables

Franklin Crow's 1984 paper, "Summed-area tables for texture mapping" calls them "summed-area tables", and Graphics Gems called them "sum tables". More recently, they're known as "integral images". In the one-dimensional case, they allow you to calculate the sum of values in an arbitrary interval in constant time by subtracting the values from the summed-area table at the ends of the interval: sum(f[m:n]) = -sat(f)[m] + sat(f)[n], where sat(f)[i] = sum(f[0:i]), assuming f's indexes start at 0.

N-dimensional case

You can compute an N-dimensional sum table; sat(f)[i0, i1, ... in] is sum(f[0:i0, 0:i1, ... 0:in]). In some interesting sense, more dimensions makes it more powerful: the set of queries that can be answered in constant time grows exponentially with the number of dimensions, while the constant-time factor only grows linearly with the number of dimensions.

Decimation

As an extension, you can use a decimated summed-area table, with values only present every (e.g.) 16th or 32nd index, without losing the constant-time property. You may have to consult the original array, but only up to 2*(16-1) or 2*(32-1) values of it, which is constant. If you needed to keep the original array around anyway, this dramatically reduces the space cost of the technique without slowing it down too much, which (I speculate) might actually make it faster.

N-dimensional decimation is nontrivial because you can't just store the values at the lattice points; to keep the constant-time guarantee, you have to store values for every point *at least one* of whose coordinates is a round number. This means decimation basically only saves you a linear factor of 16 or 32 or whatever, and you have to use a sparse-array representation to get any good out of it.

Generalization over operations

This problem, sum(f[m:n]), is a specific case of the general idea of " range queries".

Sum tables generalize beyond integer addition. Clearly they work fine for mod-N integer addition, vector addition, and the combination of the two (e.g. XOR). In fact, you can use summed-area tables over arbitrary groups, as long as the group operation and inverse are computable. (The range query is constant time only as long as those computations are constant time.)

(For the N-dimensional case, I think you may also need commutativity, but I'm not sure.)

A logarithmic-time alternative to sum tables for semigroups

But what do you do if you're interested in an operation that doesn't have a left inverse? For example, the "minimum" operation (or in general the meet operation of a meet-semilattice) can't have inverses of elements, because it's idempotent, so you can't compute it with a

sum table.

But you *can* compute it in logarithmic time with a tree. Let

```
mint(f, m, n) = nil if m == n
              = (m, n, min(f[m:n]), mint(f, m, floor((m+n)/2)),
                                    mint(f, floor((m+n)/2), n)) otherwise
```

You can compute this in linear time, assuming a constant-time binary min operation, as follows:

```
mint(f, m, n) = nil if m == n else
                (m, n, f[m], nil, nil) if m == n - 1 else
                (m, n, a, b, c) where
                    k = floor((m+n)/2) and
                    (_, _, a0, _, _) = b = mint(f, m, k) and
                    (_, _, a1, _, _) = c = mint(f, k, n) and
                    a = min(a0, a1)
```

Now if you precompute mint(f, 0, f.length), which is a balanced binary tree with 2*f.length - 1 nodes, not counting the nils, and which can be computed in linear time, you can compute min(f[m:n]) for arbitrary m, n in logarithmic time given that tree. That algorithm is straightforward:

```
tmin((a, b, c, d, e), m, n) =
    c   if a >= m and b <= n
    nil if a >= n or b <= m
    nmin(tmin(d, m, n), tmin(e, m, n)) otherwise

where nmin(a, b) = b if a == nil
                   a if b == nil
                   min(a, b) otherwise
```

This algorithm applies to any semigroup over the elements; it can be used to calculate sums as easily as it can be used to calculate minima, although less efficiently than a sum table.

Space reduction: decimation

Analogously to sum tables, if your leaf nodes represent spans of some 16 or 32 elements instead of 1, you get a dramatic space reduction without losing the logarithmic-time asymptotic performance.

Space reduction: array storage

The contents of the tree produced by the mint() function depends only on m and n, except for the min(f[m:n]); and if f.length is a power of 2, it is a full binary tree. A full binary tree can be stored, as in the classic binary heap, in an array a such that the children of the element at a[i] are at a[2i+1] and a[2i+2] (zero-based). So you can store the minima for the tree in an array (without decimation, of 2*f.length - 1 elements) rather than allocating numerous nodes on the heap.

This requires a slight enhancement to the lookup algorithm to recompute the same (m, n) as the construction algorithm, rather than looking them up in the tree.

Constant-space bottom-up construction

If you construct the tree recursively, in addition to the O(N) space for the results, you need O(log N) stack space. But that is not necessary. If you're using the array storage suggested in the previous section, you can fill the array starting from the end, so that the only auxiliary storage you need for the construction process is a simple counter.

Enhancement: updates in logarithmic time

If you update an element of the original array, you can update the tree nodes going back up to the root to reflect your update in worst-case O(log N) time. Appending or removing elements at the end of the array can be handled similarly, although sometimes appending an element will involve creating a new root node, which (in the array representation of the tree) is worst-case O(N), but amortized constant time.

This is a reason you might actually want to use these trees to handle range-sum queries rather than using sum tables: updating this tree takes O(log N) time, while updating a sum table takes O(N) time.

Enhancement: indices

In the case where the semigroup operation is exactly minimum or maximum over a totally ordered set, the value stored in each treenode will be the value of one of the items in the original array. In this case it is strictly more powerful to store the index of that item rather than its value. This may be useful if you have some other data that are indexed the same way.

This allows the algorithm to solve the "range minimum query" or RMQ problem, for which it is known as the "segment tree" algorithm. Danielp wrote a really awesome tutorial on RMQ on Topcoder.

The constant-time "sparse table" algorithm given in that article unfortunately only works for semilattices rather than general semigroups including arbitrary monoids.

N-dimensional case

This generalizes easily to quadtrees, octrees, etc., although the efficiency guarantees are not as good.

A constant-time alternative to sum tables for semigroups

A.C. Yao published one in 1982, "Space-Time Tradeoff for Answering Range Queries", but I don't know it. I think it's explained in the aforementioned really awesome tutorial on RMQ, involving a reduction to the least-common-ancestor problem, but I don't understand it yet.

Thanks

To John Cowan, Gian Perrone, and Seth David Schoen for discussion.

Topics

A tournament to decide which notes to devote attention to polishing

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

I should start publishing some of these notes. But how to select? There are 236 of them! Maybe read through four of them today, pick the two that seem to have the most potential, improve them, then repeat the process the next day, then do the same with the four winners.

To find some that are in the top, say, 10%, we need three or four such tournament levels. Four tournament levels look like this:

```
          Day 1: A B C D
          Day 2: E F G H
      Day 3:     A B E F
          Day 4: I J K L
          Day 5: M N O P
      Day 6:     I J M N
   Day 7:        A B I J
          Day 8: Q R S T
          Day 9: U V W X
      Day a:     Q R U V
          Day b: Y Z α β
          Day c: γ δ ε ζ
      Day d:     Y Z γ δ
   Day e:        Q R Y Z
Day f:           A B Q R
```

That should give two items that would, in the perfect-fairness case, be in the top 6.25% of the essays, with four revisions, at the cost of reading an essay 60 times (reading 32 unique essays) and doing 30 revisions. The two-of-four comparison system obviously deviates somewhat from perfect fairness, although, if the comparisons are reliable, it will find the best two; but the runners-up may not be the #3 and #4 best. (In the above example, C and D could have been the #3 and 4 best, but still be eliminated on day 1 because of having the bad luck of sharing the same day with A and B, if those were #1 and #2.)

Topics

- Algorithms (p. 3310) (123 notes)
- Psychology (p. 3669) (18 notes)
- Strategy (p. 3734) (10 notes)

Likely-feasible non-flux-deposition powder-bed 3-D printing processes

Kragen Javier Sitaker, 2015-09-11 (updated 2019-12-20) (49 minutes)

I just wrote this long thing in Flux deposition for 3-D printing in glass and metals (p. 1366) about a powder-bed 3-D printing technique that deposits a binder that's completely inert at room temperature, but upon firing the print in a kiln, becomes active. (See also 3-D printing by flux deposition (p. 466).)

I think there are a variety of other possibilities in powder-bed 3-D printing that have not yet been fully explored.

Powder-bed 3-D printing, *in general*, consists of depositing one layer after another of powder, alternating with selectively applying some kind of treatment to the top layer of powder which results in causing it to solidify. The classic inkjet-binder-deposition 3-D printing is one example, but selective laser sintering and selective laser melting are other processes in this category.

Magnesium oxychloride (Sorel cement) or zinc oxychloride

Sorel cement is a combination of highly water-soluble magnesium chloride (nigari) with highly water-insoluble magnesium oxide (milk of magnesia); it's a cement similar to Portland cement, but more refractory, less water-resistant (and won't harden underwater), and nearly twice as strong.

So, although I'd have to investigate more, I think you could use an aqueous solution of magnesium chloride to moisten a powder bed of sand and dry magnesium oxide to form a very strong mortar.

Zinc oxychloride might work in the same way: zinc oxide is insoluble, like magnesium oxide, while zinc chloride is so soluble it's deliquescent; and zinc oxychloride or zinc hydroxychloride formed in precisely this way was formerly used as a dental cement, like the zinc phosphate mentioned below. Zinc chloride, however, is acidic, corrosive, and a skin irritant, while magnesium chloride is free of these problems. In fact, Sorel investigated zinc oxychloride before settling on magnesium oxychloride!

Selective hammering

Instead of squirting binders onto a powder bed like an inkjet printer, you could bang the shit out of it with hammers like a dot-matrix printer, ideally under vacuum so that you don't generate explosive gas expulsions. The impact will stick together the particles in the vicinity, affecting a total mass of powder material similar to the total mass of the hammer. (This suggests that low-mass hammers are in some sense optimal.)

Selective electrical sintering

For beds of metal particles, instead of squirting binders, you could

touch the surface of the powder with an electrode and drive a large current into it, sintering the nearby particles together through joule heating of their contact points, like an old-fashioned coherer.

The electrode would probably have to be a carbon rod, since any other plausible material is likely to stop working due to surface oxidation.

This probably won't produce a strongly bonded part, but might be enough to produce a solid part that can then be solidified further by other means.

Cement precipitation by cross-linking with calcium or other polyvalent cations

A number of anions, such as phosphate, carbonate, and alginate, form water-soluble compounds with monovalent cations like those of the alkali metals (sodium, potassium) and ammonium, while forming water-insoluble compounds with divalent cations like those of the alkaline earth metals (calcium, magnesium). Calcium and magnesium also have highly water-soluble salts, such as their nontoxic chlorides. Phosphate is also water-soluble in the form of phosphoric acid.

This means that by mixing two liquids you can precipitate a solid through a double ion replacement reaction. This is used in molecular gastronomy spherification of foods, forming a flexible calcium alginate membrane around a liquid center with sodium alginate dissolved in it.

(I'm pretty sure this is because these anions are polyvalent and are strongly enough bonded to their cations that they are solvated together with them, rather than separately, so that once the cations are also polyvalent, the individual anions floating around with their individual cation harems are replaced by endless chains in which each cation links together different anions. But I'm no chemist.)

Candidate cements and fillers

Other polyvalent cations, like Cu_{2+}, Zn_{2+}, Fe_{3+}, Fe_{2+}, and Al_{3+}, should also work for this. Most of these also have relatively innocuous water-soluble salts; $ZnCl_2$, $Fe(NO_3)_3$, $Cu(NO_3)_2$, $FeCl_3$, and $AlCl_3$, as well as blue, white, and green vitriol, of course, which last are innocuous enough to use as nutritional supplements, but are subject to onerous reporting paperwork in places nowadays; acetates of calcium, magnesium, copper, zinc, and ferrous iron (II) are also all soluble, though acetate of zinc only a bit, and acetate of ferric iron (III) not at all. Ferrous citrate is also soluble.

So the plan is that you precipitate a solid cement in the interstices of an aggregate or filler, such as quartz, grog, carbon black, fumed silica, mullite needles, aluminum oxide crystals, rutile needles, zircon crystals, mica, chopped carbon fiber, chopped basalt fiber, chopped glass fiber, powdered graphite, powdered copper, powdered silver, hollow glass spheres, hollow steel spheres, chopped cellulose fiber (such as sawdust), silicon carbide, clay (especially finely dispersed bentonite), diatomaceous earth, etc.; or a mixture. If the cement is relatively inert, unlike the aggressively alkaline slaked lime and portland cement, a wide variety of fillers are possible that couldn't withstand the harsh chemistry of everyday building materials.

Different possible resulting cements include the following; I'm

including Mohs hardnesses as an imprecise but readily available and roughly accurate guide to strength:

• Aluminum phosphate — the rare mineral berlinite, with Mohs hardness 6.5, or, when hydrated, the unusual mineral variscite, with Mohs hardness 4.5, used as a dental cement; or, sometimes, aluminum triphosphate, aluminum hexametaphosphate, or aluminum tetrametaphosphate.
• Calcium phosphate — probably hydroxyapatite, like tooth enamel, Mohs hardness 5; can incorporate iron(II) and manganese substituting freely for calcium to form the equally hard graftonite;
• Calcium carbonate — calcite, Mohs hardness 3;
• Calcium alginate — a silicone-like insoluble, nontoxic organic water-gel-forming elastomer;
• Ammonium magnesium phosphate — the light, very soft (Mohs ≤ 2) mineral struvite, which might be formed if ammonium phosphate is the phosphate salt used;
• Magnesium phosphate is a GRAS food additive for buffering acidity, but I don't know anything about its mechanical properties;
• Magnesium carbonate — the soft mineral magnesite, Mohs hardness 3.5–4.5, which can be calcined at only 500–800° to magnesium oxide, or magnesia alba (periclase, as mentioned above), which doesn't melt until 2852° and is used as a stronger alternative to gypsum in drywall;

• Calcium magnesium carbonate — the mineral dolomite, Mohs 3.5–4, which probably will *not* form even if its constituents are available (because it's picky about crystallizing);
• Magnesium alginate ought to exist and be similar to calcium alginate;
• Copper phosphate — a blue-to-green insoluble copper salt;
• Copper carbonate — a bright blue to green pigment, depending on degree of hydration, occurring as malachite and azurite in nature (which differ in their degree of carbonation); "very sensitive to acids". Mohs 3.5–4.
• Zinc phosphate — one of the oldest and most widely used dental cements, so nontoxic and biocompatible, made in something like the way I'm suggesting here (mixing zinc oxide and magnesium oxide powder with buffered aqueous phosphoric acid); occurs naturally as the rare mineral hopeite (Mohs 3–3.5);
• Zinc carbonate — the mineral smithsonite (Mohs 4.5), one of the two minerals known as calamine (the other being zinc silicate);
• Ferric phosphate — non-toxic, except to mollusks, and sometimes used as an iron nutritional supplement, but almost insoluble in water; "heterosite" or "wolfeite"?
• Ferrous phosphate — the soft deep blue to bluish green mineral vivianite, used to kill garden slugs, Mohs 1.5–2;
• Iron carbonate — the dense yellow mineral siderite, Mohs 3.75–4.25;

• Manganese carbonate — the rose-red mineral rhodochrosite, Mohs 3.5–4.

So you should be able to get relatively high strength, almost as high as portland cement (whose strength comes mainly from belite, which is known as larnite in nature, Mohs 6), by precipitating calcium phosphate crystals from a water-soluble calcium salt such as calcium

chloride and a water-soluble phosphate salt such as monoammonium phosphate; you *may* be able to get a highly refractory bond by calcining the phosphate or carbonate of magnesium into magnesia; you can get an instant nontoxic aqueous elastomeric gel with calcium alginate; you can get biocompatibility (and guaranteed-working recipes) from zinc and magnesium oxides with buffered aqueous phosphoric acid; and there are thirteen other combinations that will probably work as well.

Further alternative polycations might include nickel, mercury, and vanadium ions, but these have some disadvantages (carcinogenicity, higher toxicity) and not much in the way of available information. Further alternative polyanions might include sulfate (which does have some insoluble salts, notably calcium sulfate (gypsum) and barium sulfate), oxalate, silicate (see below), sulfide (soluble with lithium, sodium, and ammonium, but should precipitate transition metals) and perhaps some carrageenans.

Iron sulfide in particular — fool's gold — is 6–6.5 on the Mohs scale, harder than apatite. It has the disadvantage of gradually oxidizing in air, though, with corrosive results, and of course the soluble sulfides are toxic.

Liquid tank systems

It might be advantageous to work with a mixture that is liquid until the cement is precipitated, rather than consisting mostly of a packed granular filler. This doesn't exclude the use of fillers; especially bentonite clay can remain in suspension in water up to fairly high concentrations of clay without solidifying the water. It might be worthwhile to mix a little sodium or potassium alginate in with the phosphate so that the initial introduction of the calcium donor will gel things in place in milliseconds and prevent the liquid from flowing further, even if the calcium phosphate or other cement takes some time to fully crystallize. (This might be useful to limit diffusion even in a powder-bed system.)

(The advantage of Newtonian or at least non-thixotropic liquids is that their surfaces are reliably quite flat and horizontal; they have no angle of repose.)

Other plant gelling agents such as pectins and carrageenans can also be precipitated into a gel by pH control and in some cases by polyvalent cations (though there are many different types of pectin and many different types of carrageenan, and they can sometimes react in opposite ways to pH changes), and aluminum sulfate precipitates insoluble, gelatinous aluminum hydroxide when the water is insufficiently acidic.

Nucleation control

It may be desirable to prevent homogeneous nucleation in order for the cement particles to be big enough to bridge the gaps between grains of filler. For of these most cements, if the temperature is kept high enough, cement particles will only nucleate on the surfaces of grains of filler; this may help to produce a solid mass. (More speculatively, pressure control is another possible lever to control nucleation, but this would probably require a liquid-filled chamber.) It may also be possible to solve this problem by making the precipitation mass-transport-limited.

Filler particles with more extreme aspect ratios — clays such as bentonite being the champion here, though a less expansive clay may be more practical for this use — should lower the critical percolation threshold needed to form a solid mass, thus placing less stringent demands on the nucleation process.

Densification

Once you have the "green" article made out of filler grains cemented together, you can use water to wash off the unhardened mixture of filler ("powder") and unprecipitated solute, as well as washing out leftover reaction products other than the cement. Densification may be needed after the initial precipitation, since when the cement precipitates from solution, the water and other solute remain. (For example, if reacting aqueous dipotassium phosphate (which dissolves 150 g per 100 mℓ of water) with calcium chloride to produce hydroxyapatite, you have potassium chloride and water taking up space in the result.) Densification can be carried out by passing a supersaturated solution of the same cement, or a compatible cement, over the printed object once it is removed from the powder bed; or it can be carried out by infusing the pores with a different material, perhaps a melt, again after powder removal.

Electrolytic injection of cations

As an alternative source of polyvalent cations, you could use small anodes of suitable metals (zinc, copper, manganese, or iron, although maybe it might be possible with a suitable alloy of calcium or magnesium) with a controllable current; this might allow you to switch on and off the cementing action with much higher precision and frequency than pumping solute liquids in and out of a pipette or inkjet, and would avoid the need for the extra water content to maintain those cementing ions in solution.

This approach should be especially suitable to introducing controlled amounts of impurities into particular places in the printed object — for example, copper or iron ions would probably produce a bright blue color, or manganese ions a rose-red color. You could probably get a wide variety of other colors by using other metals not otherwise mentioned here; cations introduced for the purpose of adding color need not be polyvalent or form physically strong compounds.

More generally, the precise control of mixing provided by the electrolytic mechanism can be used to produce precisely controlled gradients of material properties in the cementing material, for example to produce controllable optical or acoustic refraction.

In theory you could also use a sacrificial cathode that released anions such as phosphate or carbonate when electrolytically reduced, but that seems much more difficult; I know of no such material.

Alternative solvents

Water is a terribly convenient solvent for facilitating such double-metathesis reactions, since it's capable of dissolving a very wide variety of ions, it's fairly nontoxic, and it is liquid at room temperature. But it has the major disadvantage that it contains oxygen, so to metals like calcium, water is utter death. Other polar solvents might be feasible alternatives; for example, anhydrous ammonia at low temperature and/or high pressure, or molten-salt

mixtures like FLiNaK and FLiBe at somewhat higher temperatures, or the truly outlandish polar organic solvent systems used in current lithium-ion batteries.

Bicarbonate as a hydroxyl donor

Cyanoacrylates polymerize in the presence of hydroxyl ions; dripping cyanoacrylate onto $NaHCO_3$, stealing hydroxyl ions and converting it to sodium carbonate, is a well-known manual additive manufacturing technique which can probably be improved by adding filler to the $NaHCO_3$.

Bicarbonate as a CO_2 donor

Waterglass (sodium or potassium silicate) forms a silica gel rapidly upon exposure to CO_2; maybe you can use $NaHCO_3$ as a CO_2 donor for this purpose. Certainly you can harden it with acids instead, or with ethanol.

There are other materials that harden or recrystallize upon exposure to CO_2, most notably $Ca(OH)_2$, slaked lime, which produces calcium carbonate. Normally they harden fairly slowly once wet by absorbing CO_2 from the air, but maybe you could get them to harden faster by supplying them with $NaHCO_3$.

Metastable redox systems such as thermites

Rather than using chemicals that react immediately on contact, as in the above, or initiating some kind of interaction by slowly heating the entire powder bed after careful deposition, as in Flux deposition for 3-D printing in glass and metals (p. 1366) and 3-D printing by flux deposition (p. 466), it might be worthwhile to use chemicals that can react quite energetically, but which remain almost completely inert during the printing process; and, once the printing is complete, ignite them and allow the self-sustaining reaction to run to completion. The trick is to identify reactions that would produce enough heat to produce interesting materials, but without producing enough gas to blow the nascent object to bits.

Thermites, such as the classic aluminum-powder/magnetite system formerly widely used for welding, are one example; you could selectively deposit aluminum powder into a bed of magnetite, and then ignite the thermite once the printing is done (traditionally, using magnesium ribbon). This produces molten iron and molten (!!) aluminum oxide, which I expect would then quickly quench in the much larger body of magnetite, producing a solid object consisting of a magnetite shell around a core consisting of phases of iron and amorphous or cryptocrystalline corundum; plausibly both phases might initially be continuous, as in an open-cell foam, but the corundum would almost certainly fracture severely during cooling. With some luck, the purified iron thus produced will be sufficiently ductile to remain intact.

(The temperature is 2500° when the oxidizer is hematite rather than magnetite, but I think this is limited by aluminum boiling at 2519° rather than by the energy available.)

Magnetite has some disadvantages; it will melt onto the outside of the printed object, its own properties are not all that desirable, and it

adds iron (thus, weight) to the piece. Other oxygen donors might solve or at least ameliorate these problems. However, the traditional alternatives are hematite (red iron oxide), silica, diboron trioxide (boria), a mixture of manganese dioxide with manganese monoxide, lead tetroxide, cupric oxide (CuO, the toxic tenorite), and viridian. Of these, I think silica is the one with the highest melting point (1600°), and it has the benefit of being transparent; but the metallic silicon thus formed is even more brittle than corundum. Viridian and cupric oxide offer the fascinating prospect of 3-D printing in purified chromium and copper, but cupric-oxide thermite can be explosive. Additionally, chromite ($FeCr_2O_4$) might work — I think aluminothermic reduction of chromite is used for commercial chromium smelting.

Sometimes people use teflon instead of an oxygen donor, thus producing a metal fluoride (and carbon) rather than a metal oxide.

Typically when burning aluminum with quartz as the oxidizer, sulfur is included in an aluminum–sulfur–sand composition; WP claims this functions as an extra oxidizer to add energy, as well as to ease ignition. Sulfur is sometimes used with magnetite, aluminum, and barium nitrate to make "thermate," a higher-temperature thermite with mostly military uses.

Aluminum is not the only possible fuel metal, only one of the cheapest and safest; other possibilities include zirconium, calcium (!), zinc, titanium, silicon, boron, and magnesium.

Common fillers for thermite welding include high-carbon steel, cast iron, or pig iron, which melt and mix with the purified iron to produce a steel with the desired level of carbon.

Alternatively, at somewhat higher cost, you could attempt to make the oxidizer rather than the metal the limiting reagent — for example, depositing a small amount of magnetite powder in a bed of aluminum powder, rather than the reverse; then, the newly formed material will quench in the aluminum, acquiring an aluminum coating rather than a magnetite coating. This is very risky, though, because the aluminum powder burns fiercely in air. You'd need to do it under an inert or reducing gas, or in vacuum.

The reaction between zinc and sulfur, every chemistry teacher's favorite, is another candidate. The sphalerite or wurtzite thus produced is a reasonably strong mineral (Mohs 3.5–4). Other metals, such as aluminum and I think iron, have similar reactions, but the sulfides thus formed are less stable and tend to hydrolyze.

Some materials pricing

Looking at Mercado Libre here in Argentina this weekend (2019-12-13 to 2019-12-15) I found some vendors for most of the materials I mentioned above; today the dollar is around AR$62 bid, AR$67 ask; I'm using AR$64.50/US$ for the conversion. I've ordered the materials I was able to price roughly by price.

(Addendum 2019-12-20: the dollar is AR$73 today. I spot-checked three of the prices below; none of the three have changed, in pesos, although this means they have fallen by something like 10% to 15% in dollars. This clearly means that the error bars on these prices are like 20% or 30%.)

• Silica sand for construction (not very pure, but without stones or

salt) has costs that vary greatly by location, but are generally around AR$1200/$m^3$, which works out to about AR$750/tonne at 1.6 g/cc, or AR$0.75/kg (US$0.012/kg).
• Portland cement costs US$9.64/kg (US$0.15/kg).
• Calcium hydroxide (slaked lime) costs AR$7.8/kg (US$0.12/kg).
• Fine pine sawdust costs around AR$10/kg (US$0.16/kg) although prices vary by a factor of 2 or 3.
• Coke (carbon) costs AR$45/kg (US$0.70/kg).
• Magnesium sulfate (Epsom salt) costs AR$116/kg for beer brewing or AR$112/kg as medicine or AR$83/kg or as low as AR$60/kg as a hydroponics fertilizer (US$0.90/kg).
• Diammonium phosphate costs AR$150/kg as a hydroponic fertilizer or AR$60/kg in bulk (US$0.90/kg).
• Magnetite is I think AR$70/kg ("oxido de hierro magnetico color negro") but it isn't totally clear whether the AR$350 price is for a 5-kg bag or not. (US$1.10/kg)
• Sodium silicate solution (waterglass) costs AR$72/kg for waterproofing and surface-hardening concrete ("curador silicato", in this case Sikafloor CureHard 24). (US$1.10/kg)
• Quicklime, calcium oxide, costs AR$75/kg. (US$1.20/kg). Note that this is almost ten times the price of slaked lime, suggesting that either the slaked lime is adulterated or the safe handling of quicklime is very costly.
• Bentonite clay costs AR$232/kg or more when food-grade, but as clumping cat litter, only AR$83/kg (US$1.30/kg). The cat litter might be contaminated with other clays, with silt, or with sand, but for these purposes that might be acceptable.
• Aluminum in ingots costs AR$100/kg (US$1.60/kg).
• Calcium chloride costs AR$106/kg as a desiccant (US$1.60/kg), or AR$125/kg if you only buy one kilo. For beer brewing they charge AR$165/kg which seems like it might be purer. For bath salts, AR$155/kg with a purity of 77-80%.
• Calcium nitrate costs AR$111/kg as a fertilizer (US$1.70/kg). It's deliquescent above 50% humidity.
• Green vitriol costs AR$111/kg (US$1.70/kg) to AR$165/kg as a fertilizer.
• Sodium bicarbonate costs AR$129/kg (US$2.00/kg).
• Monoammonium phosphate costs AR$143/kg as a hydroponic fertilizer (US$2.20/kg)
• Lead in ingots costs AR$145/kg from BATERIAS INDIANAPOLIS in Burzaco. (US$2.20/kg)
• Trisodium phosphate costs AR$150/kg for industrial use. (US$2.30/kg)
• 85% phosphoric acid costs AR$495/liter for beer brewing, or AR$350/liter in bulk or as low as AR$170/liter. That's 1.6845 g/cc so that's reasonably close to being AR$495 or AR$350 or AR$170 (US$2.60) per kg of phosphoric acid.
• Alumina costs AR$298/kg from Alcoa, or, in bulk, as little as AR$194/kg (US$3/kg).
• Magnesium chloride costs AR$221/kg or AR$249/kg for food or nutritional supplement use, or as low as AR$195/kg for bath-salts use (US$3/kg).
• Brass filings from keys (probably mostly free-machining brass) can cost AR$230/kg, (US$4/kg) although the price seems to vary quite a

bit; another listing has it at AR$1500/kg.
• Scrap brass in ingots costs AR$230/kg (US$4/kg).
• Powdered sulfur costs AR$329/kg or AR$244/kg in bulk (US$4/kg).
• Potassium nitrate costs AR$245/kg (US$4/kg) to AR$289/kg as a fertilizer.
• White vitriol costs AR$258/kg (US$4/kg) as a fertilizer.
• Magnesium nitrate costs AR$262/kg (US$4/kg) as a fertilizer.
• Sodium carbonate costs AR$690/kg for bath-salts use, or AR$860/kg, or AR$280/kg (US$4/kg) in bulk. It's also available in small quantities in supermarkets mixed with sodium percarbonate for laundry use.
• Manganese sulfate costs AR$381/kg (US$6/kg) as a fertilizer.
• Chopped 4.5-mm glass fiber costs AR$381/kg (US$6/kg) ("Hilo De Fibra De Vidrio Cortada 4,5 Mm - 20 Kg Carga Placas", "Comercial San José", 12 blocks from the Tigre station), or in 3-mm cuts, as low as AR$337/kg ("mecha cortada", "chemia.com.ar").
• Copper sulfate costs AR$400/kg (US$6/kg) as a swimming-pool fungicide and algicide.
• Silicon carbide costs AR$420/kg (US$7/kg) ("Carbeto de silicio", in Portuguese, imported from Brazil, brand Imerys Fused Minerals.)
• Monopotassium phosphate costs AR$483/kg (US$7/kg) as a hydroponic fertilizer.
• Calcium citrate costs AR$520/kg (US$8/kg) as a nutritional supplement. Its solubility is a bit under 1 g/liter: 100x higher than calcium carbonate, but 1000x lower than chloride and nitrate.
• Zinc oxide costs AR$1600/kg to AR$3300/kg for cosmetics use, or AR$540/kg (US$8/kg) for use in paint ("purity 99.5%, contact with skin dangerous"). Also zinc phosphate dental cement is for sale for AR$2080 for 90 grams; presumably this is the zinc oxide and phosphoric acid mentioned above.
• Iron filings ("limaduras de hierro") cost AR$600/kg (US$9/kg) to AR$850/kg.
• Magnesium citrate costs AR$650/kg (US$10/kg) as a supplement.
• Powdered lead costs AR$700/kg, (US$11/kg) but see above about lead in ingots.
• Magnesium oxide costs AR$845/kg (US$13/kg) to AR$1040/kg in 99.9% USP food-grade form.
• Copper filings ("polvo de cobre puro") cost AR$850/kg (US$13/kg).
• Potassium chloride costs AR$996/kg (US$15/kg) as a salt substitute.

• Fine brass powder costs AR$1200/kg (US$19/kg).
• Potassium silicate solution costs AR$1450/kg (US$22/kg) as a fertilizer.
• Calcium acetate hypothetically costs AR$2000/kg (US$30/kg) but a lot of people complain about that vendor.
• Powdered zinc costs AR$2600/kg (US$40/kg).
• 97%-pure powdered aluminum costs AR$2900/kg (US$40/kg) though I've seen other listings at lower prices.
• 99%-pure powdered tin costs AR$3700/kg (US$60/kg).
• Sodium alginate costs AR$40000/kg. or as low as AR$10050/kg (US$160/kg) for culinary use.
• Nitrates of iron and copper are not available.

• Zinc chloride is not available.
• Potassium carbonate is not available, which is a shame, since it's much more water-soluble than sodium carbonate.
• Ammonium carbonate is not available, although it's used in some cookies here in Argentina.
• Aside from the possibility of prying them out of dead batteries, carbon electrodes are available for arc gouging: 4 mm diameter, 305 mm long for AR$83, 6 mm diameter of some unknown length for AR$43, 10 mm diameter of some unknown length for AR$72, or 13 mm diameter of some unknown length for AR$136.
• Several vendors sell bars of magnesium as sacrificial anodes for solar hot water heaters for a few hundred pesos, and there are a few magnesium firestarters like the one I had as a kid.
• Someone is selling "glass basalt fiber" for "SMC roving", for AR$100 for 1.5 meters. I have no idea if this is actually basalt fiber but I suspect it's just glass fiber.

Some candidate mixtures explored in more detail

Although there are of course a very large number of combinations drawn from the above that are likely to work, I thought it would be useful to work out some properties and approximate recipes for a few of the variants.

Although mostly I'm considering a binder-jetting process here, keep in mind that in fact the "binder" being jetted is in most cases just water, or water thinned with alcohol, and its only function is to solvate the actual cement grains so that they can react, and in some cases to drive the reaction kinetics toward water-insoluble cement products. In most cases, another polar solvent such as ammonia, or heat from a laser or arc, could be substituted for the water "binder".

Also, many of these mixtures would benefit from additional ingredients; the U Washington Open3DP project has published a number of recipes they found worked well. In many cases, for example, they added carboxymethylcellulose or a similar plant gum to provide both wet strength and green strength.

Cat-litter bentonite or other clay body by itself

If we jet water, perhaps thinned with a little alcohol, onto a powder bed made of clumping cat litter, it will clump. If left to dry, perhaps without even depowdering, it will form a dried unfired clay object. If some sand or grog is included, this object can even be strong enough to survive handling, and such additives will also reduce shrinkage on drying, as would non-expansive clays.

Quartz sand and calcium hydroxide

This is the classic *cal y arena* mortar, cured by absorbing carbon dioxide from the air, mostly in 24 hours. It has the attractive feature of being bright white. I think U Washington Open3DP has done some work with this recipe.

Quartz sand and portland cement

This is the classic hydraulic mortar; it sets up faster if you add some slaked lime.

Quartz sand, wood flour, cat-litter bentonite, diammonium phosphate, calcium chloride

Upon jetting water thinned with a little alcohol onto this dry powder-bed mixture, ammonium chloride and calcium phosphate are formed; bentonite crystals serve to provide extra nucleation centers for the precipitating calcium phosphate, to bridge gaps between precipitate crystals (especially initially, when they are small), to add tensile strength to the weaker calcium phosphate crystals, and to stop the propagation of cracks through the calcium phosphate. The highly soluble ammonium chloride remains in solution in the pore water; if desired, it can be leached out later by immersing the finished part in water. The quartz sand fills the majority of the material and provides mostly compressive strength. The wood flour serves to reduce density and provide tensile strength, like collagen in bone.

The mixture is kept dry and must be protected from air exposure when not in use, because the calcium chloride is deliquescent at ordinary humidities; even then, the ammonium has a limited shelf life, especially when warm.

The needle-like morphology of typical apatite nanocrystals is well-suited for bridging gaps between clay particles and other fillers, and would pose no barrier to further diffusion to carry the reaction to completion; even the platelet-like morphology that sometimes occurs with apatites and often with triclinic octacalcium phosphate would work well. The spherical morphology that occurs with amorphous tricalcium diphosphate (called tricalcium phosphate, TCP) would be pessimal, and when TCP precipitates from aqueous solutions, it always precipitates in amorphous form, requiring heat-treatment to crystallize. Apatite is favored at high pH; TCP is favored at more acidic pH; and OCP is favored in between, at a slightly acidic pH.

Calcium chloride is $CaCl_2$, with a molar mass of 111 and a solubility of 650 g/liter of water at 10°; diammonium phosphate is $(NH_4)_2HPO_4$, with a molar mass of 132 and a solubility of 575 g/liter of water at 10°. Hydroxyapatite, which is the mineral cement we are hoping for, is $Ca_5(PO_4)_3OH$, with a Ca:P ratio of 5:3 and a molar mass of 502; Wikipedia says it is commonly prepared as nanocrystals from a mixture of calcium nitrate and diammonium phosphate, including at non-stoichiometric ratios. So for every 5 moles (555 g) of calcium chloride we want 3 moles (396 g) of diammonium phosphate and get some miscellaneous products plus one mole (502 g) of hydroxyapatite, 10 moles of chloride ions, and 6 moles of ammonium ions, which I think will result in 6 moles of ammonium chloride (53.5 g/mol, so 321 g) and 4 moles of excess chloride. Also we have a couple of extra hydrogens floating around, so maybe we'll get hydrochloric acid or something; might be a good idea to include some calcium hydroxide or something if that's happening. (I should work out the side products in more detail; the formation of chlorapatite rather than hydroxyapatite may be a possibility, and seems guaranteed if you heat the result to dissociate the ammonium chloride.)

Solvating that amount of calcium chloride simultaneously would take 850 g of water, and of diammonium phosphate, 690 g of water, at 10°. So, dividing, for every gram of hydroxyapatite, we need 3.1 g of water, 1.1 g of calcium chloride, and 0.79 g of diammonium phosphate. Actually we might need somewhat more or somewhat less

water than that: more because some of the water molecules are tied up by the "pore walls" of the bentonite, or less because when hydroxyapatite precipitates out of solution, the water remains to solvate new calcium chloride and diammonium phosphate. It will gradually become saturated with ammonium chloride (solubility: about 240 g/liter at 10°) and lose its ability to solvate more calcium and phosphate so they can react.

I'm not sure whether you would expect such a water deficiency to also slow the formation of the calcium phosphate crystals, allowing them to grow larger, by limiting the speed at which calcium phosphate can diffuse to the crystal growth sites, or to result in smaller crystals because the solution is more fully saturated. Both seem worth a try. Also, though, the papers I've seen on hydroxyapatite wet precipitation, like Poinern et al. 2009, required hours for the crystallization to produce particles of tens to hundreds of nanometers, and ideally we'd like it to happen at subsecond time scales, or in minutes at most. (But Victor Chen's YouTube demo of reacting sodium phosphate with calcium chloride produced a solid and completed within a few seconds; similarly Arieus Alcide's reacting calcium gluconate with potassium phosphate produced a white precipitate instantly.)

(Carbonates or hydroxides might work to liberate ammonium from the solution, and would prevent the pH from dropping (Hielscher's sono-synthesis report says they tried to keep their pH around 10 with NaOH in order to get hydroxyapatite instead of a different calcium phosphate) but the chlorides they formed would also be soluble, except in a few problematic cases like chlorides of silver, thallium, lead (plumbous, II), mercury (I) (calomel), and copper(cuprous, I), all of which are alarmingly toxic, absurdly expensive, or both. Also, I suspect any of these would form soluble complexes with the ammonium ligands, leaving us back where we started. Perhaps it would help to use trisodium phosphate, which is pretty alkaline, in place of some or all of the diammonium phosphate.)

The apatite crystals can incorporate a little magnesium, which can transform them into whitlockite, but it is reported to inhibit apatite nucleation and growth, as does carbonate. Magnesium I think favors the precipitation of tricalcium phosphate, since β-TCP shares whitlockite's crystal structure.

So, if the amount of water is about right --- as set by the amount of pore space available for the reaction --- then every 5 kg (or, say, 5 nanograms) of pore space will produce 1 kg (or, respectively, 1 ng) of cement. Because hydroxyapatite has a density of about 3.2 g/cc, this means that the cement will fill up only about 8% of the pore space, so we'd better hope that we can get by with a smaller amount of water.

(8% was arrived at as follows: anhydrous calcium chloride weighs 2.15 g/cc, and undissolved diammonium phosphate weighs 1.619 g/cc, so the 5 g of solution that yielded each gram of hydroxyapatite actually occupied 4.1 milliliters before the water began to solvate the salts, and the gram of hydroxyapatite occupies 1 ml/3.2 = 0.31 ml, which works out to about 7.6%.)

How much pore space is there? Building sand weighs d = 1.52–1.68 g/cc (see also rfcafe), which suggests a void fraction of (1 - d /2.4) = 30% to 37%, 2.4 g/cc being the density of quartz; let's say one

third. The bentonite particles might occupy 50% of the remaining space, one sixth of the total, and the phyllosilicate bentonite crystalline material might have a density of 2 g/cc (I'm not sure). Let's forget about the sawdust for the time being. So we have one sixth of the space available as pore space.

For each 4.1 milliliters of pore space, we need 1.1 g of calcium chloride and 0.79 g of diammonium phosphate. So for each milliliter of powder, we need 270 mg of calcium chloride, 190 mg of diammonium phosphate, 1600 mg of construction sand, and 170 mg of bentonite cat litter. Or, per liter of mix:

Ingredient	Mass/liter	US$/kg	US$/liter
Sand	1.6 kg	0.012	0.019
Cat litter	170 g	1.30	0.22
$CaCl_2$	270 g	1.60	0.43
$(NH_4)_2HPO_4$	190 g	0.90	0.17
Total	2.23 kg		0.84

It's probably important to make sure that the formation of the calcium phosphate take place mostly between the bentonite grains. In the powder bed, the bentonite is I think unavoidably going to be aggregated into clumps of tens to hundreds of microns in size, and water, when wetting the powder, will reach the centers of those clumps last. But the centers of those clumps are precisely where the calcium phosphate is most needed --- in other places it runs the risk of forming crystals that don't attach to anything. I think the way to solve this is to thoroughly wet-mix the calcium chloride into the bentonite before drying the bentonite and breaking it into those clumps; then mix the clumps with the sand and the crystals of diammonium phosphate. That way, when the water wets the powder, it will first dissolve all the diammonium phosphate, then begin to diffuse into the bentonite clumps, where it can cement them by forming calcium phosphate there.

If the bentonite in question is not already a calcium bentonite, it may eat some of your calcium in this process, diffusing out sodium to replace it, so you may need to use somewhat more calcium than suggested.

We can see that if we were to replace all the sand 1:1 with sawdust, assuming 1 kg/liter, it would add US$0.16 to the cost and bring it up to US$1/liter. However, sawdust has much higher porosity than sand, so it would also increase the amount of bentonite and cement that could be included; perhaps the cost might increase to as much as US$2/liter.

Phosphoric acid is not a cheaper phosphate source but might permit denser cement

The diammonium phosphate mentioned above contains one phosphorus atom per 132-dalton formula unit, and additionally it needs more than its own mass in water to dissolve it. Phosphoric acid also contains one phosphorus atom per molecule, but its molecules are only 98 daltons, and it only needs a very small amount of water. So if diammonium phosphate costs US$0.90/kg, phosphoric acid could cost as much as $1.20/kg and still actually be cheaper. But in fact phosphoric acid costs US$2.60/kg.

Jetting 85%-pure phosphoric acid out of nozzles is not going to work; it's too viscous. But you could maybe use powdered solid

phosphoric acid. However, although it's not toxic, it's pretty caustic; most of the other chemicals described above are less dangerous.

Calcium nitrate is more expensive than calcium chloride

Rey, Combes, Drouet, and Grossin explain that the usual way to deposit apatites in lab wet synthesis, with also some use in industry, is by reacting calcium nitrate with an ammonium phosphate; one reason for this is that nitrate and ammonium groups are easily driven out of the reaction product by heating. Calcium nitrate at US$1.70/kg would seem to be very nearly the same price as calcium chloride at US$1.60/kg, but calcium chloride's molar mass is 111 (or 219 as hexahydrate), while calcium nitrate's is 164 (or 236 as tetrahydrate), so you get considerably less calcium for your money unless that calcium chloride is fully hydrated and the nitrate is desiccated.

The bigger issue is that nitrates are a certain amount of hassle to deal with, due to both their toxicity to the humans and ongoing chimpanzee dominance games the humans like to play.

There are presumably cases where the greater ease of driving nitrate residues out of the structure is a decisive advantage, however.

Berlinite-bonded alumina

Grover et al. at Argonne published a paper in 1999 on this, reporting a rather astonishing result: "We hydrothermally cured a mixture of Al_2O_3 and H_3PO_4 solution between 130°C and 150°C to form a hard and dense berlinite-bonded alumina ceramic." I would not have thought that phosphoric acid could attack sapphire so easily, much less that the result would be a low-temperature way to bond alumina grains. They got a "putty-like" gel of aluminum phosphates after heating alumina in aqueous phosphoric acid, which could dry (to a hydrated xerogel I suppose) and then redissolve in water; by heating it to 150° they drove off not only the water but also the remaining hydrogen, converting the water-soluble aluminum phosphates into berlinite, aluminum orthophosphate, $AlPO_4$, which is covalently bonded to the remaining alumina.

This is such an astounding development that I wonder why I haven't heard of it before; perhaps it has some fatal flaw not mentioned in the paper. The cement described would cost close to US$3 per kilogram and requires baking to cure, so it's not going to replace portland cement unless some material prices change dramatically, but it's both cheaper and presumably much stronger than common petroleum-based plastics, while sharing most of their advantages, although requiring a slow curing process to reach its full strength.

It might work for a variant of this binder-jetting process, too. Although the soluble aluminum phosphates are probably too syrupy to squirt out of jets, you can reportedly dry them to a hard, rocky form that dissolves again in water; squirting water onto it may be sufficient to stick particles of a filler such as sapphire together into a green body that can then be baked at 150°, perhaps with a preliminary aging step. And, like the processes described in 3-D printing by flux deposition (p. 466), it might be possible to bake the whole powder

bed, since the water is an essential reagent in the hardening process; if this works, it would make the green strength irrelevant, but might irreversibly cure the unused aluminum-phosphate binder. And of course you can use an FDM-like selective paste deposition process like those used for adobe and clay-paste "3-d printing".

More on this berlinite-gel process in Berlinite gel (p. 848).

The double-metathesis-type reactions described above might be a more comfortable way to precipitate aluminum phosphates *in situ* than pressure-cooking alumina in strong phosphoric acid for several days. For example, you could produce an aluminum phosphate by mixing solutions of aluminum chloride and diammonium phosphate --- even if the aluminum phosphates you get are water-soluble, they won't be nearly as water-soluble as the reagents, so you might get enough precipitation. But it seems likely that, without baking, you'll only get soluble aluminum hydrogen and dihydrogen phosphates.

Sawdust, diammonium phosphate, sodium bicarbonate, and calcium chloride

You should be able to make a kind of inexpensive waterproof fiberboard by precipitating apatite between the wood fibers in the same way described above, but a larger fraction of the resulting substance will be made of apatite, because you don't have sand grains taking up two thirds of the volume. Sodium bicarbonate can keep the combination alkaline, like trisodium phosphate above, which not only favors the precipitation of apatite rather than less-stable calcium phosphates, but also protects the wood fiber from acid. Bicarbonate will buffer the system, preventing it from becoming too alkaline, and additionally serves as a fire retardant.

The elasticity of the mix may pose problems for a powder-bed 3-D printer, since it will spring back after you compact it. You can compact the whole mass at the end of the process, squeezing both air and water out of the mix and causing the water to spread somewhat. The alternative of maintaining the bed under compression while you squirt binder onto it seems impractical. Just adding binders like carboxymethylcellulose won't help because it's the dry part of the powder bed that causes the problem.

A reasonable mix might be 500 g sawdust, 100 g sodium bicarbonate, 235 g calcium chloride, 165 g diammonium phosphate; this works out to US$0.80/kg.

Magnesium sulfate, sodium carbonate, and silica sand

These two soluble chemicals (Epsom salt and washing soda) ought to form magnesium carbonate (magnesite). Magnesium sulfate is US$0.90/kg and sodium carbonate is US$4/kg. I haven't worked out the stoichiometry, but probably the article of commerce is the heptahydrate, which will have an impact on that.

Green vitriol and trisodium phosphate

At respectively US$1.70/kg and US$2.30/kg, with some luck, these two highly soluble salts should react to make an insoluble basic copper phosphate, the deep green pigment pseudomalachite and its polymorphs ludjibaite and reichenbachite, $Cu_5(PO_4)_2OH_4$. Again, I haven't worked out the stoichiometry.

Spot-welding brass filings with a carbon or TIG

electrode

Brass filings can be bought as cheaply as US$4/kg. There are a couple of ways you could easily melt a controlled-size spot on the surface of a bed of brass filings using a carbon-rod or TIG electrode. First, you could charge up a capacitor and move the rod closer to the surface until there is an arc, with the rod being positive and the filings being negative; this will deposit most of the energy into the filings. Second, you could bring the rod into contact with the filings, run a current through the rod and an inductor, and then break contact, again inducing an arc, again with the electrons impacting the rod and the ionized air or other gas molecules impacting the filings. In each of these cases the spot size is controlled by the amount of energy built up in the energy-storage device.

Third, you could run an arc more or less continuously from the electrode to the bed, as in normal TIG welding or carbon arc gouging.

Lead particles in the powder bed might help with the sintering; I think molten lead can dissolve a signficant amount of copper, and I don't know about zinc (see Filling hollow FDM things with other materials (p. 2119) and A phase-change soldering iron (p. 2270) for more on related systems). If so, as described in 3-D printing by flux deposition (p. 466), it might be possible to later bake the finished piece to induce the lead to diffuse away from what were initially the sintering boundaries, thus preventing the evolution of any liquid until a substantially higher temperature.

Other powdered metals, such as copper, lead, stainless steel, steel, or aluminum, would also work to a greater or lesser extent, but steel and aluminum are relatively hazardous and would probably need to be done under an inert gas such as argon.

Topics

Groping toward a high-efficiency speaker driver

Kragen Javier Sitaker, 2019-04-02 (15 minutes)

How do you do a very simple but high-efficiency audio output circuit on a microcontroller?

The simplest possible approach is to connect a speaker between a GPIO pin and a power rail, then generate a signal on the GPIO pin. Digital outputs are quite low impedance when generating either a 1 or a 0, so very little power is wasted in the pin driver in the microcontroller.

This has wonderful linearity, but it has a few different problems: power, DC waste, impedance matching, and high-frequency noise.

Power: the GPIO pin is typically capable of sinking or sourcing 5–50 mA, sometimes at 5 V but often at only 3.3 volts. 50 mA at 5 V is 0.25 W, which is not going to be very loud unless it's in an earphone.

DC waste: the average value of the GPIO pin when it's emitting a signal that's symmetric around some zero value is going to be 2.5 V or 1.7 V, half the supply voltage. This DC component of the signal is going to draw current through the speaker coil without producing any sound. Worse, it will typically see a lower resistance than the speaker's nominal impedance.

Impedance matching: when you have a source with an internal impedance driving a load, the maximum power applied to the load is when the load's impedance equals the source impedance. If the load's impedance is too low, most of the voltage is dropped in the source impedance, and the voltage that actually reaches the load is too low. If the load's impedance is too high, all the voltage reaches the load, but only a fraction of the source's current-delivery capacity comes into play. To be concrete, driving 20 mA (ac!) through a garden-variety 8Ω dynamic speaker is only going to give you 160 mV (ac) across it, rather than the 3.3V or 5V your microcontroller can theoretically deliver.

High-frequency noise: GPIO pin voltage transitions are fairly sharp and so potentially have a lot of power in the ultrasound part of the spectrum. For directly driving a speaker, this may not be a problem, but under some circumstances the high-frequency signal can result in excessive heat dissipation in the speaker as well, especially once you solve the above power delilvery problems.

So I was thinking about these ideas:

First, how about just driving the speaker through an audio transformer? This ensures no dc reaches the speaker, allows the microcontroller to see an effectively higher speaker impedance (by the square of the turns ratio) and the waste of driving dc through the transformer's winding is potentially smaller than the waste of driving it through the hair-fine speaker winding. Transformers may even reduce the transmission of high-frequency noise, though probably not in a very well controlled way.

Second, how about using a transistor switch as an amplifier? An N-channel MOSFET or NPN transistor to ground would work, as

would a PNP BJT to a positive voltage supply. The MOSFET case can work with extremely high efficiency! A 2N7002 can switch 200 mA at up to 60 V with only 5Ω of on-resistance; in theory that would be up to 12 W of output, at which point it would be dissipating 1 W, requiring the TO-92 package for heatsinking rather than the SOT23-3L.

To solve the high-frequency noise problem, though, we need analog filtering — and we need it *after* the transistor switch, because if we do it *before*, we lose the efficiency that brought us here in the first place. If we use mostly LC filtering, rather than RC, maybe we can get high efficiency.

More elaborately:

Put a speaker in series with a capacitor to ground to block DC and stuff below, say, 20 Hz; this way the power dissipated by the speaker is almost all in the right frequency band to be turned into sound, even if the speaker's best-case efficiency is low, like 5% or so. Put another, much smaller capacitor in parallel with the speaker-capacitor combination in order to attenuate high frequencies, like above 20kHz or so. Put an inductor in series with the capacitor-speaker-capacitor combination in order to attenuate high frequencies further and prevent short-circuit currents. Put a Schottky clamping diode in parallel with the inductor-capacitor-speaker-capacitor combination, from ground up to the inductor's "input", to prevent inductive voltage spikes to large negative voltages. Drive the diode-inductor junction with a MOSFET to the positive voltage power supply — either a GPIO output from the microcontroller itself (which perhaps avoids the necessity for the Schottky diode), a logic-level-input P-MOSFET with its gate hooked up to the GPIO, a regular power P-MOSFET with its gate driven by a logic-level-input MOSFET or by a BJT, or maybe even a bootstrapped N-MOSFET.

Basically this is a buck converter driving a speaker through a DC-blocking capacitor.

So far so good, but there's one missing piece still: when there's a signal, the input has a positive DC level relative to ground, and there's no DC path to ground for it. So stick an additional humongous inductor in parallel with the original capacitor-speaker-capacitor combination in order to provide a non-dissipative DC path.

It might be more convenient to turn the whole passive-and-diode output part of the circuit upside down, hooking it up to the positive power supply rather than ground, so we can use an N-channel MOSFET to drive its input by shorting it to ground intermittently, rather than any of the annoying options for CMOS high-side switching.

Some rough numbers. Suppose the speaker is a standard 8Ω half-watt dynamic speaker. To make the whole mess feasible, let's use a 10:1 audio transformer to drive it; that way 1VAC 125mA on the output works out to 10V 12.5mA on the input, an impedance of 800 Ω instead of 8 Ω. The DC-blocking capacitor then shouldn't have too much more than 1kΩ of impedance at 20Hz ($1/\omega C < 1k\Omega$) which gives us 8μF, a blessedly quite feasible value. The parallel capacitor to short out stuff above 20kHz also shouldn't have too much more than 4kΩ of impedance, but at 20kHz, so 8nF is adequate.

The impedance of the whole capacitor-transformer-capacitor thing is going to be in the neighborhod of 2–3kΩ, so we want the input inductor to be in the neighborhood of that when we hit 20kHz; this requires a relatively large inductor of around 16mH in series. However, the much worse problem is that our parallel inductor to ground — the one that's supposed to drain off our DC voltage so we can keep running the circuit — is supposed to have an impedance in the neighborhood of 2–3kΩ for frequencies of **20Hz**. And *that* would require a humongous monster **16 HENRY** inductor.

So, what's the problem here? I think that maybe I've made the load impedance too high, requiring high-impedance capacitors (which are cheap) and high-impedance inductors (which aren't). Maybe I don't really need the transformer; then both the capacitors and inductors could be 100× lower impedance. That means the capacitors would need to be 800µF (annoying but fairly commonplace) and 800nF (totally normal), while the inductors would need to be 160µH (totally normal) and 160mH (also annoying but not exotic).

With ideal components, all the energy here would be dissipated either by the diode or the speaker. In practice, the inductors in particular will have significant parasitic resistance, depending on how much copper you're willing to lavish on them. One 150µH inductor I have a datasheet for here, the toroidal SMD PM2110-151K-RC from Bourns, has 0.049Ω of DC resistance; another, Bourns's "dual-winding SRF0703-151M" (really a transformer), has 0.986Ω. So the losses should be manageable.

Simulation shows reasonable results with these values and 100kHz PWM, though there's some serious harmonic distortion on the "negative-polarity" side of the wave at high amplitudes (presumably from the diode), and there's about 6 dB attenuation already at 8 kHz, and significant bleedthrough with a 40kHz PWM carrier. Probably a more judicious choice of component values would yield a sharper cutoff at a more appropriate frequency. It also seems to have rather large currents through the diode at times, not to mention the other passive components.

In Falstad's circuit format:

```
$ 1 1.0E-7 10.20027730826997 50 5.0 43
r 928 336 928 384 0 8.0
c 928 384 928 464 0 7.999999999999999E-4 0.060217163598211165
w 928 304 992 304 0
l 992 304 992 464 0 0.16 1.0273205335302287
w 928 304 864 304 0
c 864 304 864 464 0 8.000000000000001E-7 0.10880129845424946
l 864 464 768 464 0 1.6E-4 1.0359085688452483
d 768 464 768 304 1 0.305904783
w 864 304 768 304 0
w 928 304 928 336 0
w 864 464 928 464 0
w 928 464 992 464 0
R 768 304 768 256 0 0 40.0 5.0 0.0 0.0 0.5
f 688 480 768 480 0 1.5
g 768 496 768 544 0
a 592 480 688 480 1 15.0 0.0 1000000.0
```

```
R 592 496 544 496 0 4 100000.0 5.0 0.0 0.0 0.5
170 592 464 544 448 3 20.0 40000.0 2.0 0.2
o 0 64 0 35 0.15625 0.025 0 -1
o 13 16 0 35 10.0 1.6 1 -1
o 17 64 0 35 2.5 9.765625E-5 2 -1
```

I was thinking that maybe I could use an electrolytic capacitor for the series capacitor for the speaker, since one end of it is periodically almost shorted to ground, but now I realize that won't actually work; the capacitor-speaker combination is in parallel with an inductor, which means its average voltage over time must be zero. (Otherwise the current through the inductor is growing without limit!) On the minus side, this means that you can't use an electrolytic capacitor. On the plus side, it means that by the same token, you don't need any capacitor, because the parallel inductor itself guarantees a zero DC component to the signal as seen by the speaker. And that, in turn, means that we don't really need such a large inductor; its impedance above 20Hz only needs to be large compared to 8Ω, not 20–30 Ω. (160mH at 20Hz gives us an impedance of $2\pi fL = 20.1\ \Omega$.)

Fuck inductors, though. We probably can't get by with *just* capacitors because any just-capacitor circuit of this sort is going to have a massive shoot-through current when it first turns on, and your MOSFET is maybe going to explode. But what if the only inductor in the system is a little choke that's there to stop that from happening? Maybe with a diode or capacitor in parallel with it to keep it from generating massive voltage spikes that blow up the transistor when we turn it off.

But then maybe we can use a capacitor in series with the speaker to high-pass filter the signal (at 20 Hz or so) and another in parallel with either the speaker or the capacitor-speaker combination to low-pass filter it (at 20 kHz or so). Maybe we'd like the time constant of the high-pass-filtering capacitor to be around 50 ms, and the low-pass filtering cap to be around 20 µs. Now we really do want to use a transformer, say 10:1, making the effective speaker impedance 800 Ω, so our high-pass-filtering cap in series with the transformer can be 68 µF (and either electrolytic or MLCC ceramic), while the high-pass filter in parallel with that can be 22 nF.

Let's say 100kHz PWM with a 50%-max duty cycle is our working assumption, and we're feeding the whole shebang from five volts. So the transistor is on for up to 5 microseconds, then off for at least 5 microseconds. And let's say we don't want more than, say, 200 mA going through the MOSFET, because it's a 2N7002 or something. How big should the choke be?

We want the current to ramp up to 200 mA in 5 microseconds when the choke sees all 5 volts, since that's probably the worst case. That's 125 µH. Let's use a capacitor in parallel with it to keep its voltage spikes limited without wasting any energy or introducing any nonlinearity; if we want the combination to resonate at 1MHz, we want the capacitor's reactance to exactly cancel the inductor's reactance at that frequency: $2\pi fL = 785$ ohms $= 1/(2\pi fC)$, giving 202 pF (and indeed 1 MHz ≈ (125 mH 200 pF)^-½). This does reintroduce the startup short-circuit path we were hoping to get away from, but now it ends 4000 times faster. However, in simulation, this only limits the inductive voltage spike to 140 V, which is still too

high. So we probably need some kind of more aggressive damping. A 4.7nF capacitor instead, in series with a 100Ω resistor, keeps the spike down to -23 V.

An interesting thing is that whatever the ringdown network for the choke is, whatever losses it has are mostly switching losses, i.e., they happen after each pulse, so using more pulses means more losses there. So to the extent that your normal filtering is adequate to keep your PWM or whatever out of your audio, you can lower the PWM frequency to reduce the ringdown losses.

If you wanted to keep the voltage spike down to 5 volts at 200 mA, you could use a 21.5Ω resistor in series with a regular silicon 700mV diode. Except that of course the resistor's voltage drop will fall as the current does, so it's more of an exponential decay than a linear one, and so it doesn't come close to stopping the choke current before the transistor turns back on. But maybe that's actually what we want? In simulation, it does keep the voltage spike to 4.6 volts.

Hmm, I just realized that maybe the body diode in the 2N7002 would have a similar effect. Maybe it points the wrong way, though.

Also I think the whole idea of capacitors on all the paths from power to ground is a dumb idea. Sooner or later all those capacitors are going to be charged up to 5V and then no more current will flow.

Topics

Relational modeling and APL

Kragen Javier Sitaker, 2019-05-20 (updated 2019-05-21) (5 minutes)

I think there's a kind of logic-programming or constraint-logic approach that preserves most of what's good about array languages, while adding the kind of multidirectional inference languages like Prolog and especially miniKANREN have.

I've noticed these notes are getting repetitive as I write down the same ideas over and over again, having forgotten them; I'll try to link them here.

I was thinking about object-oriented equational rewrite rules and IRC bots with object-oriented equational rewrite rules (p. 838) today or yesterday, thinking about how it would be nice to define properties like ".vol = π.r².h; .area = 2π.r(.h + .r)" so that "foo.area" would do a search for formulas that could be applied, and use that one if it happened that "foo" has properties .h and .r. (I vaguely handwaved in my head that some kind of namespacing could alias this to "foo.cylinder::h" so it wouldn't collide with, say, "foo.planck::h".) And it occurred to me that this is precisely the same thinking in A principled rethinking of array languages like APL (p. 1995) about array conformability. (And in OMeta contains Wadler's "Views" (p. 842), I opined that they were the same thing as Wadler's Views.)

A difference, though, is that in the rewrite-rule thinking, a single property can have multiple definitions, like methods overridden in different OO classes, of which normally only one is applicable, while in A principled rethinking of array languages like APL (p. 1995) no such merger was contemplated, except through explicit conditionals.

In IRC bots with object-oriented equational rewrite rules (p. 838) I'd suggested resolving conflicts through a specificity ordering, like CSS or Aardappel. But thinking of the rewrite rule as a *deduction rule* suggests another alternative. Suppose we read ".vol = π.r².h" as specifying an equation in the usual sense — a relationship that is known to hold in all situations where the variables are defined. In that case, it defines a *constraint*, which means that not only can we use it to compute .vol, but we can use it to compute .h if we happen to know .vol and .r from someplace else. That means that it's also an *assertion* — if we have a different computation of .vol from some other definition, the two values must agree, or we have discovered an inconsistency in our model!

So that's a different way of dealing with "rewrite rule" conflicts — crash the program if two conflicting definitions give different results.

(And Prolog, of course, considers conflicting definitions as equally valid possibilities, though there is a definite order to them; the second is only used if we backtrack out of the first.)

But we can get that without giving up the yumminess of implicit loops we get in A principled rethinking of array languages like APL (p. 1995) (and APL with typed indices (p. 3264) and Index set inference or domain inference for programming with indexed families (p. 1434)) by virtue of saying something like .r = [1mm 2mm 5mm 10mm], meaning that there are four situations of interest defined by different radii. And these "different cylinders" can all "inherit" a

common .h, have a .h that varies together with .r, or have a .h that varies independently of .r. The underlying logic is conditional deduction over various different situations.

I talked a bit about this connection in More thoughts on powerful primitives as well.

I wrote down almost exactly these ideas two years ago in Relational modeling (p. 1102).

By using the more powerful kind of relational programming miniKANREN provides, rather than the more limited kind used by Prolog, we may be able to solve more models. Also, for numerical relationships like the ones I used as examples above, interval arithmetic or affine arithmetic may be very useful for two reasons — first, in order to make progress toward solving or proving insoluble a system that starts out underconstrained (consider .x = exp(.x), which has no solution, or .x = exp(.x)/4, which has two solutions), and second, for determining whether two values of a numerical quantity reached through different computational paths are in fact equal or not. (Interval arithmetic can't prove that the values are equal, but it can reliably tell whether an apparent difference is too big to be due to rounding error.)

There's a potential conflict here between the use of implicit patterns of known values to distinguish situations where a property doesn't *exist*, like trying to find the radius of a cube, and where a property *isn't known yet*. I suspect that there may be different valid design choices one can make to resolve this conflict which lead to interestingly different languages that work for different purposes.

Topics

- Programming (p. 3658) (286 notes)
- Programming languages (p. 3656) (47 notes)
- Interval and affine arithmetic (p. 3528) (24 notes)
- Arrays (p. 3326) (17 notes)
- Constraint satisfaction (p. 3387) (9 notes)
- APL (p. 3320) (9 notes)
- Prolog and logic programming (p. 3667) (8 notes)
- Predicate logic (p. 3644) (6 notes)
- miniKANREN (p. 3585) (6 notes)

Microlens vibrating lightfield

Kragen Javier Sitaker, 2018-07-14 (updated 2018-07-15) (11 minutes)

Microlens arrays for 4D lightfield displays are now being shown publicly — more or less, a brute-force realization of "holographic displays". Essentially, this is garden-variety lenticular 3D, but with a computer screen rather than a printed image behind the lenses. But this places new demands on the underlying screen resolution. To reliably get human binocular vision at a distance of one meter, I think you need new images every 50 milliradians, half the distance between human eyes. With two dimensions of parallax and a radian or so of viewing angle, this works out to a few hundred images, let's say 256. For a 512×384 image, which I think is what the original Macintosh had, that means you need 50 megapixels.

(As a point of comparison, Thomas Burnett's FOVI3D display displays a 180×180 image with 50×50 viewing angles across 90° (10π milliradians), for a total of 81 megapixels, using 20 4K OLEDs tiled without very good brightness correction; another model uses 108 megapixels. He's doping all the pixels to yellow-green to get spatial resolution and brightness at the expense of color.)

It's challenging to fabricate so many pixels. One possible solution is to temporally multiplex a smaller number of pixels, scanning each pixel over a significant area with a vibrating or rotating mirror. DLP and inorganic LED pixels are capable of response times sufficiently fast to make this a viable option; LCD, CRT, and plasma pixels are not. DLP pixels are too close together, though, which leaves inorganic LED pixels as the only option. I'm going to consider only monochrome displays for now.

Doing this kind of thing with a vibrating or rotating mirror requires some kind of scanning pattern that scans the (image of the) physical pixel over all the spatial locations it's responsible for illuminating; the traditional approach from analog TV is a pair of sawtooth signals, a slow one (≈60Hz) for Y and a fast one (≈15kHz, with important harmonics up to, say, 105kHz) for X. Other scan patterns, such as Lissajous, are possible but do not remove the necessity for a rather high scanning speed in one dimension.

Vibrating or spinning a mirror at low speed is much easier than vibrating a mirror at high speed, especially if the mirror is large. Consider a small area of 393 216 spatial locations — this could be 512 lines of 768 spatial locations, 768 lines of 512 spatial locations, 128 lines of 3072 spatial locations, 48 lines of 8192 spatial locations, and so on. In the last case, supposing the Y dimension is scanned at 60 Hz, the X dimension will need to be scanned at 2880 Hz, which is considerably less demanding than 15kHz. At some cost in effective resolution and complexity of brightness control, you can use a sinusoidal scan for the X, avoiding the need for responsivity at higher harmonics such as 8640 Hz. Alternatively, you could use a spinning mirror such as those used in supermarket scanners and laser printers; a hexagonal mirror would give you a 2880 Hz scan at 480Hz or 28'800 rpm, which is challenging but feasible.

That is for a single LED; you need to position it at least 48 LED heights above the next LED below.

Ideally you'd like to be able to change all 50 megapixels for each frame, but even without that ability, you can make the display work for less-frequently-updated images. For example, you could assign a microcontroller to each 8192×48 area (512×3 macropixels), which it would have to refresh at least 60 times a second: 23'592'960 pixel outputs per second. This is within the capability of STM32F microcontrollers, which cost 59¢. Moreover, they can control 16 lines at a time at this speed, so a single microcontroller (and perhaps three associated 40¢ ULN2003 seven-Darlington low-side switching chips) can control a 8192×768 area (512×48 macropixels). Eight such microcontrollers would suffice for the whole 6144×8192 display (512×384 macropixels), with a total BOM cost of US$14.32 for the silicon, not counting the 128 tiny LEDs and 128 resistors, and actually wasting a bunch of the Darlingtons.

However, an STM32F0 microcontroller typically only has 4K of RAM. Rather than economizing so much on microcontrollers, it might make more sense to economize on mirror X scanning. For example, if we use 64 microcontrollers instead of 8, with 16 LEDs per microcontroller, we can control 1024 LEDs via 147 ULN2003s, and each LED only needs to cover an 8192×6 area rather than 8192×48, so the X scan on the mirror can slow from 2880 Hz (28'800 rpm with a hexagonal mirror) to 360 Hz (3600 rpm), which is much easier to achieve. Moreover, the signal to each LED need only change at 2'949'120 Hz rather than 24 MHz — this still poses signal integrity challenges but is dramatically simpler. You need 147 ULN2003s and 64 STMF0s, for a silicon BOM cost of $96.56.

(I should check the ULN2003 datasheet to see if it can handle a 3MHz or 24MHz signal.)

Note that this still only leaves you with 256K of RAM to hold the data to display on a 6144×8192 display, i.e. 192 pixels per byte. I think there are some items in the STM32 line with more RAM, up to 2 megabytes per chip. DRAM would likely be fast enough. Also, external RAM chips with their data lines connected to the driver inputs might work, as long as the drivers have a disable input. But it might be adequate to generate a single 8-kilobit scan line (1 kilobyte) on each scan, starting with some kind of heightfield model or something that you could reasonably rasterize a line of in 48MHz/360Hz = 133'333 32-bit CPU cycles.

By using the double-parabolic mirror trick used for the famous floating coin illusion, you can cause the microlens-array-generated lightfield image to appear to float in midair instead.

If we wanted to reduce this approach to an absolute minimum demoable product, maybe we could start with something the size of an 80×25 terminal with a narrow viewing angle. Let's say 80×5 = 160 macropixels horizontally and 25×8 = 200 macropixels vertically. And let's say we are willing to accept fewer viewing angles: 8 horizontally and 4 vertically, for example. And let's lower the refresh rate to a cinema-flickery 24Hz. And let's use a mirror that only scans in one dimension, horizontally. Now we need 800 fricking LEDs, but we can probably multiplex them a bit, because the whole matrix is only 1280×800, so we only need 61'440'000 pixels out per second, or 3'840'000 16-bit updates per second. If you run 32 of the LEDs at any given time using 5 ULN2003s, you can use 25 high-side switches (what are these called?) in, say, 4 chips. To get these 57 GPIOs you

might need, say, four microcontrollers, with very lax constraints on their output timing. This works out to 13 chips that collectively cost US$6. The 800 fricking LEDs may cost more than that, but probably not more than US$16.

The optics may be somewhat more of a problem. You only need 24Hz scanning (240 rpm) and possibly some kind of magnification in order to be able to use a manageably small scan mirror.

You might really want RGB LEDs and multiple brightness levels, which, at these speeds, are probably best achieved by linearly controlling current sources rather than PWM. These are probably achievable but may be difficult.

To investigate

• ULN2003 speed: Is 3MHz OK? 24MHz? The ULN2803A datasheet (the one with 8 Darlingtons instead of 7) says 130 ns propagation delay low to high, 20 ns high to low, which suggests it ought to be able to make it up to about 6MHz, but the ULN2003A gives a max of 1 µs for each of these, suggesting a limit of 500kHz, despite a typical propagation delay of 250 ns (thus 2MHz).
• Does the ULN2003 have chip enable? No.
• What's the high-side equivalent of the ULN2003? There used to be a UDN2891 but it's obsolete. As a sort of replacement, TI has come out with the US$1.81 TLC59123 and TLC59123A 8-input 500mA 13.2V latching synchronous high-side drivers, which claims it can only be clocked up to 1 MHz. It doesn't have a chip enable either. It also has 100–200 ns propagation delays, and requires clock pulses of at least 100ns, so you would think it ought to be able to do several MHz. There are other alternatives, like the US$2.52 Allegro A2982 (8 parallel 50V 500mA high-side drivers, but up to 10µs turnoff delay).
• Is PSRAM fast enough for framebuffers? The US$3.20 ISSI IS55WV51216EBLL-55TLI 8-megabit 16-bit-parallel 44-pin TSSOP PSRAM claims 55 nanoseconds, which is plenty fast enough, though the datasheet only claims 60ns; the US$2.10 ISSI IS62C1024AL-35QLI-TR is a 128K × 8 (1-megabit) real SRAM 32-SOP that claims 35ns.
• original Macintosh screen size: 512×384? No, 512×342.
• How much do SMD LEDs cost? How small do they come? 4.0¢ in metric 1608 packages (archaic 0603) in quantity 1000. These are the Rohm SML-D12x1 series, which come in five colors: V and U reds (630 and 620 nm), D orange (605 nm), Y yellow (590 nm), and M yellowish green (572 nm). They can withstand 100mA peak current (at 10% duty cycle 1kHz), 20mA continuous current, and 54mW power dissipation, with respectively 40, 63, 100, 100, and 30 millicandelas at 20mA. They drop 1.7–2.2 V. The Lite-On LTST-C191KGKT, another archaic 0603, costs 6.0¢; these can handle 30mA continuous, 80mA peak, and 75mW dissipation, and claim a max of 71 mcd at 20 mA instead of 30. These are binned by brightness and color. (But the min is only 18.) These LEDs (both types) turn off after dropping by about 200 mV; neither specifies a junction capacitance or response time in its datasheet.

Topics

Text relational query

Kragen Javier Sitaker, 2019-08-28 (10 minutes)

I talked a little about Lotus Agenda in Agenda hypertext (p. 1836). Today I want to explore a different related idea: a flexible tool for making relational views of text file contents, whether written by hand or generated by machine.

An Agenda file was, primarily, an unordered bag of text snippets. A lot of Unix utilities treat text file lines as sort of database records — roughly speaking, `sort` sorts the records, `uniq` eliminates duplicates and possibly provides you with a count, `grep` selects records matching a pattern, `cut` projects away some of the columns, `look` does a lookup in a sorted index, `join` and `paste` do relational joins, and so on.

These utilities are often the fastest way to get certain tasks done, but they're kind of hard to use (especially robustly), and they don't deal with ad-hoc-formatted text at all, which was Agenda's strength. However, Agenda's queries and views used a sort of tagging system that wasn't very powerful.

The basic idea: create relations by pattern-matching text lines

What if we had a system that also treated a text file as a bag of lines, but permitted relational queries on those lines, similar to how Sniki permitted relational queries on its link graph (see Prolog table outlining (p. 2837))? For example, consider these lines:

```
-rwxr-xr-x   1 user user     19096 Dec 30  2018 xshmucalc_fb
-rw-r--r--   1 user user       254 Dec 30  2018 erosion1d.c.~1~
-rwxr-xr-x   1 user user      8984 Dec 30  2018 erosion1d
-rw-r--r--   1 user user       534 Dec 30  2018 erosion1d-log.c.~1~
-rw-r--r--   1 user user      1122 Dec 30  2018 erosion1d.c
-rw-r--r--   1 user user      1362 Dec 30  2018 erosion1d-log.c
-rw-r--r--   1 user user     40055 Dec 30  2018 erosion1d-log.lst
```

Suppose you match them against this pattern, using two PCRE regexps:

```
-rwxr-xr-x 1 user user $size:\d+ $date:.* $executable
```

(`executable` gets the default pattern `\S+`; unquoted space matches any amount of space.)

This produces the following relation:

size	date	executable
19096	Dec 30 2018	xshmucalc_fb
8984	Dec 30 2018	erosion1d

The other lines don't match the pattern, so they don't enter into the result.

How about files that have an Emacs backup file, the thing with the version number between ~ marks?

```
$_ $_ $_ $_ $size1:\d+ $date1:.* ${fname}.~$v~
```

```
&$_ $_ $_ $_ $size2:\d+ $date1:.* $fname
```

This is a kind of self-join, joining the set of records with itself, but using two different patterns. They might both match the same line in different ways, but they each need to match some line for the query as a whole to succeed. Those lines may not be consecutive — in the above data they are not.

Inference rules or views

A better way to formulate that query is through a *view*, or inference rule. Here's an inference rule that factors out the common part of the above; it consists of a query, followed by one or more templates that transform the query results into inferred lines:

```
$_ $_ $_ $_ $size:\d+ $date:.* $fname
.: File $fname is $size bytes, modified $date.
```

This inference rule causes a panoply of inferred lines to implicitly spring into existence:

```
File erosion1d.c.~1~ is 254 bytes, modified Dec 30  2018.
File erosion1d is 8984 bytes, modified Dec 30  2018.
File erosion1d-log.c.~1~ is 534 bytes, modified Dec 30  2018.
File erosion1d.c is 1122 bytes, modified Dec 30  2018.
```

And so on. This allows us to write the self-join query from before much more conveniently, now matching these inferred lines:

```
File $fname $_:.*
&File ${fname}.~$v~ $_:.*
```

Such inference rules have the potential to produce infinite loops of inferring more and more lines:

```
GNU$stuff:.*

.: GNU's Not Unix$stuff
```

In general I think it is undecidable whether this is happening, so probably the best solution is to ensure that if it happens, it doesn't lead to disasters.

JSON?

This is messy, like the Unix shell utilities that inspired it — concatenating raw strings and then trying to parse the results is a recipe for disaster — so maybe inferring JSON and using destructuring matches like modern JS would be better than inferring text lines; then we only need to use text matching when it's part of the problem statement:

```
$_ $_ $_ $_ $size:\d+ $date:.* $fname
.: {fname, size, date}

{fname}
&{fname: "${fname}.~$v~"}
```

Here `{fname}` is syntactic sugar for `{"fname": "$fname"}`, and similarly for `{fname, size, date}`.

Grouping

Suppose you specify an aggregate operation on a field:

```
$perm   1 user user $size.sum:\d+ $_:.*
```

Aggregation implies that you need to do some kind of grouping in order to have groups to run the aggregation over. How do you do the grouping? Well, in SQL, if you use grouping, the only fields you can legally SELECT without an aggregate operation are the ones you're going to GROUP BY. (MySQL is historically lax about this.) So implicitly we could just use the fields that don't have an aggregate operation attached to them, so the above will group by permissions!

```
      perm    count size.sum
---------- -------- --------
-rw-r--r--        5    43327
-rwxr-xr-x        2    28080
```

Multiple aggregations on the same field don't fit easily into this approach, but they are rare.

A useful aggregation that makes sense in this context is `.join(sep)`.

Sorting

By default aggregation results should be sorted descending, but by which field? You should be able to tag a field in your query as the sort key; consider this data from `ps ux`:

```
user     28358  0.7  7.8 2114240 310916 ?      Sl   Aug22  59:20 /usr/lib/firefox
/firefox -contentproc -childID

user     28905  0.0  1.2 122440 49028 pts/2    S    00:49   0:04 xpdf.real sensor
s-18-00768.pdf
user     29260  0.0  0.0  22460  3516 pts/4    Ss+  Aug26   0:00 bash

user     29299  0.0  0.0   8340  1892 pts/4    T    Aug26   0:00 less sensors-18-
00768.pdf
```

You could match it with:

```
$uid $pid $_:.* $vsz:\d+ $rss:\d+ $tty $stat $start $cpu $cmd:.*
```

But maybe you'd like to sort by RSS:

```
$uid $pid $_:.* $vsz:\d+ $rss+:\d+ $tty $stat $start $cpu $cmd:.*
```

Or sort ascending instead:

```
$uid $pid $_:.* $vsz:\d+ $rss-:\d+ $tty $stat $start $cpu $cmd:.*
```

Of course more sophisticated sorting requires multi-field keys, specifying numeric versus ASCIIbetical versus locale versus mixed-text-and-numbers, field ordering in the key, and so on. But the simple case should be easy.

Non-equality criteria

The above allows you to select lines where a field is equal to a constant or to some concatenation that includes some other field value. So you can find your zero-byte files. But what if you want to find your files that are over a mebibyte? You need to be able to say something like this:

```
$perm   $nlinks user user $size[>1048576] $_:.* $name
```

Line sequences

Most text files aren't purely unordered bags, and if you can grok a bit of their structure, you can do much more useful things. There's a whole ladder of language power to ascend here — first strict sequences of line templates:

```
commit $commit
Author: $author:.*
Date:   $date:.*
```

Then some kind of vague contextual line-template thing for simple hierarchical structures with header lines at the top, where each occurrence of the *last* line in the template creates a row, but implicitly pulls in all the fields from the most recent occurrence of each of the others:

```
* $h1:.*
...
** $h2:.*
...
*** $h3:.*
...
${_:.*}<${url}>$_:.*
```

This would be especially useful for things like `ls -lR` output:

```
-rw-r--r-- 1 user user 1060956 Dec  2  2018 e0000ea7.au
-rw-r--r-- 1 user user 1060956 Dec  2  2018 e0000ffe.au

./whatsapp-mockup:
total 104
-rw-r--r-- 1 user user  1192 Oct 23  2016 camera.png
-rw-r--r-- 1 user user   591 Oct 23  2016 check.png
```

Here you want to use a pattern something like this, so that you can know what directory each file is in:

```
${dir}:
total $dirtotal
...
$perm   $nlinks $user $group $size $date:.* $filename
```

Alternatively, you could tag some kind of thing onto the end of the line pattern that asks for the *most recent* matching line, without implying a hierarchical structure.

Then some kind of regular expression system for line templates, then a full grammar system (which is getting into Tagging parsers (p. 208)), which could also extend down into the lines themselves, so that you can do things like parse the ingredients out of "Neapolitan ($17): Ham, tomato, garlic, and mozzarella". The only trouble with that is that an AST isn't very similar to an N-ary relation.

Web scraping

Probably the largest current source of structured data that needs to be parsed out of big text files, and for which regexps are currently the best choice.

Data entry

If you were writing a file to be parsed by such a query system, you could imagine a text editor that would make it very easy by offering syntax highlighting and contextual autocomplete — maybe any word that was the same as in the line above it would be grayed out, and TAB at the end of a line that was shorter than the previous line would copy down whatever text was the same on those two lines. So for example at the end of this:

```
Barbara is Joanie's mother.
Joanie is Ryan's mother.
Ryan
```

TAB would insert a grayed-out "is", taking you to the next "field", at which point an autocomplete dropdown combo box thing would suggest that you might want "Joanie" or "Ryan". (Maybe it could guess that the two data columns were the same type and suggest "Barbara" too.)

Topics

CIC-filter fonts

Kragen Javier Sitaker, 2017-06-28 (1 minute)

The traditional way to compose the graphical things that make up fonts, whether as outlines or as strokes, is to concatenate sequences of lines, arcs, and Bézier curves.

I've thought about alternatives including decomposing those shapes into splines and sine waves. Now it occurs to me that maybe CIC filters would also be a reasonable kind of way to decompose those shapes.

The idea is that the path of a paintbrush painting the letter is some kind of f(t) = (x, y) parametric curve, and CIC filters are capable of producing fairly interesting parametric curves at pretty low computational expense. Maybe stroke width should also be part of it.

Topics

Salt slush refrigeration

Kragen Javier Sitaker, 2017-08-22 (updated 2019-10-08)
(12 minutes)

Salt depresses the freezing point of water. When you freeze salty water, the behavior depends on whether you're at, above, or below the eutectic point, -21.1° and 27% NaCl. If less saline than the eutectic point, first you get ice crystals sucking up heat and growing without much salt until the remaining brine is at the eutectic point, at which point it freezes. If more saline than the eutectic point, instead you get salt crystals sucking up (much less) heat and growing without much water until the remaining brine is at the eutectic point, at which point it freezes.

On the other hand, if you have salty ice and you melt it partway, the part that melts first will be the eutectic phase, leaving the remaining phase (either water or salt) more concentrated in the remaining crystals. At this point you can mechanically separate the two with an arbitrarily small energy input.

I was thinking that you could use this behavior somehow to get a refrigerator, but now I'm not sure you can. It seems like the movement of mass into the eutectic phase is kind of unrecoverable by freezing and melting. Maybe you could separate them with a pressure swing or by distillation.

Frozen salt water as a freezer thermal reservoir

(This comes from a discussion with SpeedEvil on Freenode.)

But suppose you have a conventional freezer, cooled using a conventional refrigeration approach like the compression-condensation cycle or an ammonia-absorption cycle, and you'd like some resilience against problems --- either power outages, as in Balcony battery (p. 2377), or mechanical failures. A phase-change thermal reservoir inside the freezer is one way to achieve this.

However, just putting water bottles in the freezer is suboptimal. Normally the freezer is at -20° or so, and the water bottles won't melt until 0°, which means that all your food also heats up to 0°, possibly allowing it to spoil or get soggy. A lower-temperature phase-change material would work better, ideally something that melts just above the freezer's normal temperature, like about -18° or so.

You could try using just near-eutectic NaCl salt water, but you run the risk that, with repeated freeze-thaw cycles, it will separate --- I didn't understand this until SpeedEvil explained it to me.

Suppose it's a little lower in salt content than the eutectic 27%. Low-salt-content crystals will freeze first, concentrating the salt in the remaining solution, and at -20°, you will be left with a large amount of lower-salt-content crystals floating on top of the remaining very-nearly-eutectic solution at the bottom. If this system (because "mess" sounds too unappealing) melts, the higher-salt-content water is denser and thus will tend to stay at the bottom, although there will be a small amount of diffusion at the boundary layer. If it then refreezes, the same thing will happen again, but this time the

initially-freezing crystals will be even lower in salt content, and thus will freeze at an even higher temperature.

Repeated cycles of this will eventually separate the mixture into a large amount of very-nearly-eutectic mix at the bottom, which never freezes, and a small amount of very-nearly-pure water at the top, which freezes far too easily, separated by a very thin diffusion layer.

I think one possible cure is to make it a little *higher* in salt content than the eutectic 27%, so that the crystals that freeze first are higher in salt content than the eutectic, concentrating water in the bottom. (I'm assuming that the crystals will still be water crystals and not salt; if not, this won't work!) This way, when the system melts again, the denser, saltier water liberated from the crystals at the top of the solution will not be stable in that position --- it will produce convection cells that carry it back toward the bottom of the tank. This should provide some mixing, but I'm not sure if it will be enough.

Another possible cure, of course, is to use an impeller or a spoon to mix the liquid when there is liquid.

A third possibility is to immobilize the whole solution in some kind of gel, such as agar, so that the crystals cannot float or sink. This way, the physical distance between the different phases of crystals remains small enough to prevent diffusion. Over time, though, I think crystal formation in repeated freeze cycles will cut the gel matrix to shreds.

A fourth possibility is to use a Peltier cooler on top of the salt-water tank inside the freezer to drop it an extra 2° relative to the rest of the freezer. This is a relatively small temperature difference and a small heat flux, so the Peltier cooler should be relatively efficient, although its waste heat does add to the freezer's load.

A fifth possibility is to find another material whose aqueous solution has a slightly warmer eutectic point, one toward the bottom end of the freezer's normal temperature range. There are an abundance of inexpensive and nontoxic salts, acids, and bases that might work. Surely there is a database of their eutectic points somewhere.

A sixth possibility is to just run the freezer a bit cooler, perhaps cooling the brine tank directly and arranging things so that heat from the outer walls diffuses first through the food and then into the brine tank, thus keeping the food a bit warmer than the brine tank.

Candidate eutectic systems

If $NaCl{\cdot}nH_2O$ is barely unacceptable, what might work better? Ternary aqueous salt systems (e.g. NaCl, KCl, H_2O) are probably going to have *lower* eutectic temperatures rather than higher ones. For example, according to a paper by Hall, Sterner, and Bodnar in 1988 on "Freezing point depression of $NaCl–KCl–H_2O$ solutions", this system has a eutectic at -22.9° at 74% water, 20.5% NaCl, and the remainder KCl. (I think those are weight fractions, not mole fractions.) So we can probably mostly restrict our attention to binary systems of water plus a single solute.

For storing in conjunction with food we probably want something nontoxic and non-caustic; even nitrates, bromides, or soluble hydroxides would be questionable. I think this mostly restricts us to soluble chlorides, acetates, formates, phosphates, bicarbonates,

carbonates, and sulfates of potassium, sodium, magnesium, calcium, and perhaps ammonium and aluminum, plus nontoxic water-soluble organic compounds like urea.

Material Eutectic temperature with water
$CaCl_2$ −51°
$MgCl_2$ −33°
KCl −11° at 19.5% (plus additives)
Or −10.8° at 19.1% in Willem van der Tempel's 2012 thesis on "Eutectic Freeze Crystallization"; eutectic given as 19.4 wt% KCl.
Urea
(NH_2CONH_2) −12°, Frisbeetool (plus additives)
Na_2SO_4
(mirabilite) −1.24° I think at about 3.98%
$CaSO_4$ Insoluble, that's plaster, dude
$MgSO_4$ about −5° at about 19%
NaH_2PO_4 about −10° at about 35%?
$NaNO_3$ −17.45° at 37.93% (Holmberg, 1968, AE-340)
Na_2CO_3 −2.06° at 5.70% (Holmberg)
KNO_3 −2.87° at 10.29% (Holmberg)
K_2SO_4 −1.59° at 6.48% (Holmberg)
NH_4Cl −15° (Frisbeetool)
$AlCl_3$ caustic and somewhat toxic, but also I don't know
Sodium acetate
(CH_3COONa) about −18° at about 22%
CH_3COOK −60° at 49%
$NaHCO_3$ −2.23° at about 6.2%
Na_2CO_3 −2.13° at about 5.5% (same source)
$NH_4H_2PO_4$
(monoammonium phosphate) −4.15°?
$Al_2(SO_4)_3$ No freaking idea

From this I think we can conclude that potassium chloride is a very promising choice, despite the dozens of other compounds I haven't investigated yet; and you could probably use urea, ammonium chloride, or sodium acetate. (Urea might require some extra measures to prevent bacterial growth; ammonium chloride might outgas ammonia, perhaps eventually exploding if sealed.) Ternary eutectics involving some of the salts in the −10°–0° range might be interesting, too — perhaps they could either combine with one another or with something like potassium chloride to give a more ideal eutectic point.

Some of these systems have a serious supercooling problem — sodium acetate is famous for this property. If the solution is sufficiently capable of supercooling, you might be able to cool it to the freezer's minimum temperature of −20° or whatever while the reservoir remained entirely liquid — and then, if it started freezing, it would *heat the freezer back up* to its eutectic temperature until it finished freezing. I think some of the "additives" mentioned above by Frisbeetool are actually for nucleation sites to prevent supercooling. You'd think boiling stones would be a sufficient solution to this problem, but maybe not?

Non-aqueous solvents such as propylene glycol might be worth investigating, but giving up water's enormous enthalpy of fusion of 333 kJ/kg seems like a sacrifice that's probably not worth making. Amusing thought: heavy water has both a larger enthalpy of fusion

and smaller freezing point depression, and it's not sufficiently toxic to rule it out for this purpose; it's just far too expensive at present.

System sizing

The worst power outage I've had to endure was three weeks at Christmas, in the worst heat of the summer. Suppose that a normal refrigerator requires about 220 W, as suggested in Household thermal stores (p. 1533), so a normal chest freezer might require 300 W, or perhaps 100 W if superinsulated. Three weeks of 200 W is 360 MJ, so keeping food frozen through such an outage would require on the order of one tonne of salty ice — half a tonne if you could hit the 100-W figure, or only 256 kg or so if you could insulate all the way to 50 W. By contrast, keeping food frozen for an extra day (at 200 W) would require only 17 MJ or 50 kg of ice, and keeping it frozen during the night when the solar panels are off would require only about 25 kg of ice.

At the tonne level, the cost of the phase-change material becomes significant, potentially hundreds of dollars for the salt.

Why freezer thermal stores are important

Solar photovoltaic energy is on track to be by far the cheapest source of energy to date, but energy storage — especially kilowatt-scale energy storage — is not cheap. If you can run your freezer only on sunny days, when energy is abundant, you can save yourself or your municipality the need to use scarce cloudy-day or scarcer nighttime energy to keep your food cold. The kind of phase-change thermal reservoir discussed above is about two or four orders of magnitude cheaper than batteries, which are still US$100 to US$200 per kilowatt hour (US$30 to US$60 per megajoule).

Why do I say such a thermal store would be cheaper? According to Thermodynamic systems in housing (p. 2804), water's heat of fusion is 333 kJ/kg, so 3 kg per megajoule — at the US$0.58/k$\ell$ cost for expensive reverse-osmosis water from Sorek cited in Calculations about desalination in Israel (p. 2827), water costs US$0.0017 per 3 liters and thus per megajoule. Adding a controlled amount of salt might add a bit to that cost, but I don't think it's more than an order of magnitude. According to A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206), supermarket salt cost AR$41.65 per kg in 2012 when AR$16 was US$1, so that's almost US$3 per kg, or US$1 per 3-kg megajoule; but I'm pretty sure bulk rock salt is at least one order of magnitude cheaper than that.

A freezer designed for use in this way would probably be a bit larger than the freezers we currently use.

Topics

Pattern matching and finite functions

Kragen Javier Sitaker, 2017-05-10 (14 minutes)

What if we unified function definition, pattern matching, and dictionaries?

Consider this table:

```
| Input | Output |
|-------+--------|
|     1 | "st"   |
|     2 | "nd"   |
```

In JS or Python, we could encode it as a hash table: `{1: "st", 2: "nd"}`. This has the advantage of being "just data" and therefore easily changed, for example to add more items, and more easily optimized, for example by using a hash table.

But we could also encode it as code instead of data; in JS, for example, `x => x === 1 ? "st" : x === 2 ? "nd" : null`. OCaml has a syntax that allows us to encode it as code while we omit the variable name: `function 1 -> "st" | 2 -> "nd"`.

All of these cases benefit from having a fallback:

```
|   Input | Output |
|---------+--------|
|       1 | "st"   |
|       2 | "nd"   |
| (other) | "th"   |
```

In Python, we can write this, more or less, as `collections.defaultdict(lambda: "th", [(1, "st"), (2, "nd")])` (although `defaultdict` has a fatally bug-prone interface; you shouldn't use it). As code, this function could be written in JS as follows:

```
x => x === 1 ? "st"
   : x === 2 ? "nd"
   :           "th"
```

Or, in OCaml, as follows:

```
function
    1 -> "st"
  | 2 -> "nd"
  | _ -> "th"
```

The OCaml approach works by pattern-matching the argument against each of the "patterns", potentially binding variables (just as in passing arguments to a function) that can then be used in the consequent of that pattern match. In this case, there happen to be no variables either on the left or the right side.

Really simplified overview of type

inference in OCaml

OCaml does type inference to figure out what the argument type — which is to say, the domain of the function — should be. To simplify things and avoid subtyping relations, normal ML type inference requires that the domain be a known type, not some arbitrary collection of values. Consequently, the first example above produces a warning:

```
# let suffix = function 1 -> "st" | 2 -> "nd" ;;
Characters 13-43:
  let suffix = function 1 -> "st" | 2 -> "nd" ;;
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val suffix : int -> string = <fun>
```

However, OCaml's type inference algorithm has been extended in ways that do require subtyping, including "polymorphic variants" and objects with methods. Here's an example with polymorphic variants:

```
# let rec int_of_peano = function `Z -> 0 | `S(x) -> 1 + int_of_peano x ;;
val int_of_peano : ([< `S of 'a | `Z ] as 'a) -> int = <fun>
# let rec peano_of_int = function 0 -> `Z | n -> `S (peano_of_int (n-1)) ;;
val peano_of_int : int -> ([> `S of 'a | `Z ] as 'a) = <fun>
# peano_of_int 11;;
- : [> `S of 'a | `Z ] as 'a =
`S (`S (`S (`S (`S (`S (`S (`S (`S (`S `Z))))))))))
# int_of_peano (`S (`S (`S (`Z)))) ;;
- : int = 3
# int_of_peano (peano_of_int 11) ;;
- : int = 11
```

Note that the return type of `peano_of_int` is not quite the same as the argument type of `int_of_peano`:

```
([> `S of 'a | `Z ] as 'a)
([< `S of 'a | `Z ] as 'a)
```

That's because the argument type is an upper bound, while the return type is a lower bound. It's valid to pass a value of a subtype, for example only `` `Z ``, as an argument; the types don't have to match exactly. Similarly, it's valid to pass the return type into a context that could handle some wider array of possibilities, perhaps `([`S of `a | `D of `a | `Z] as a)` if we want to support efficient binary representation through both successors and doublings. But the converse operations are not valid — we cannot pass `` `D (`Z) `` to `int_of_peano`, because it will crash, and we cannot use `peano_of_int` in a context that can only handle `` `Z ``.

Of course, it's not just a matter of the polymorphic variant tag matching; the type of its argument has to unify, too:

```
# int_of_peano(`Z (`D (`Z)));;
Characters 12-26:
  int_of_peano(`Z (`D (`Z)));;
```

```
              ^^^^^^^^^^^^^^
Error: This expression has type [> `Z of [> `D of [> `Z ] ] ]
       but an expression was expected of type [< `S of 'a | `Z ] as 'a
       Types for tag `Z are incompatible
```

Making it work with infix operators

In OCaml, the `|` and `->` tokens are not infix operators; you cannot make expressions with them alone. They are part of the `function` and `match` productions in the grammar. But what if they were infix operators? What if we want the following expression to parse as infix, work by pattern-matching, and evaluate the function outlined above?

```
  1 -> "st"
| 2 -> "nd"
| _ -> "th"
```

Or, using operator tokens that make it look like JS or Python data:

```
1: "st", 2: "nd", _: "th"
```

The first is fairly straightforward: `x -> y` is, more or less, JS's `x => y`, a λ-abstraction. It's a function that takes `x` as an argument and returns `y`, evaluated on a context augmented with whatever names are bound in `x`. But we kind of want it to be able to *fail*, either at compile-time or at run-time, so that we can union `1 -> "st"` with `2 -> "nd"` and get `1 -> "st" | 2 -> "nd"`. And, for this kind of case, we really want the failure semantics to be *ordered* — we don't want to return `"th"` for everything, just everything not otherwise covered.

Leaving aside the very interesting type inference question for now, let's consider what this gives us. We have a single symbol here for defining pattern-matching lambda-expressions that can fail, and a second one that extends it to pattern-matching ordered conditionals, which also work as dict literals. This (plus some syntax for applying a function to an argument) is already more than enough for Church numerals, and so we're already past the Turing-completeness line; but can we make it more convenient?

In Perl, Python, and especially JS and Lua, it's very common to use dicts as general-purpose data structures. But here our "dicts" are functions (whose domain we perhaps have some way of computing, or at least conservatively approximating). Can we pattern-match on those dicts?

In ES6, pattern-matching on a "dict" (called, confusingly, an "object") looks like this:

```
const {a: b, c: d, e: f} = {c: 1, e: 2, g: 3};
```

This binds the variables `d` and `f` to `1` and `2`, and the value `b` to `undefined`. We could imagine that, given the option to declare a failed match, the missing value for `a: b` would cause the pattern match to fail, but the missing binding for `g: 3` would not — the point of storing your data in dicts in the first place is so that you can add new properties to it without breaking existing code, which is how email

headers have remained backwards-compatible for forty fucking years.

If the "dict" you're matching is a function (that can fail), this corresponds to the pattern-match requiring it to succeed on some finite set of arguments mentioned in the pattern. And those semantics are straightforward to implement, although maybe inefficient.

This means that to try to invoke a function, you're taking the function that is its argument and trying to invoke it (possibly several times) with the keys in the argument pattern. This sounds like it's going to be an infinite regress, but presumably at some point you will bottom out in symbols and other atoms, which, say, cannot be invoked as functions.

If we try to keep our syntactic noise to a minimum, we could declare that lowercase barewords are symbols and variables need uppercase (as in Erlang), use the `:` and `,` symbols from Python/JS instead of `->` and `|`, and use the infix-syntax and function-application-by-juxtaposition syntax from OCaml. So then our `peano_of_int` function ends up as:

```
Peano_of_int = 0: z, N: (s: Peano_of_int (N-1));
```

And then its counterpart should look like this:

```
Int_of_peano = z: 0, (s: N): 1 + Int_of_peano N;
```

And the example we started with would be written:

```
1: "st", 2: "nd", _: "th"
```

Arguments first!

At this point, we start to want local variables or let-expressions.

But let-expressions are just, in some sense, syntactic sugar to paper over the unfortunate fact that in `(λx.some long thing involving x)3`, the `x` argument and the `3` are spatially far apart, and we want them to be close together. That is, we have the order `x Y 3`, and we want `x` and `3` to be adjacent. Of the six possible orderings of the three items, only `x Y 3` and `3 Y x` do not have them adjacent, and `let` rewrites that to `x 3 Y`, omitting the literal tokens.

But there are three other orderings that would also solve the problem, which are `Y x 3`, `Y 3 x` and `3 x Y`. Since `Y` is an expression written in terms of `x`, it's desirable for readability for `x` to precede it and be adjacent to it, so `3 x Y` would perhaps be the clearest order.

Which is to say, perhaps arguments should precede functions in a function application expression, as, in a sense, in the ς-calculus and other object-oriented languages — even though, in this case, the functions are first-class values rather than merely selectors.

This leads us to write our examples above as follows, with a needless Dijkstra-like `.` introduced to improve familiarity:

```
Int_to_peano = 0: z, N: (s: (N-1).Int_to_peano);
Peano_to_int = z: 0, (s: N): 1 + N.Peano_to_int;
```

This pants-on-head syntax now gives us the sugar to avoid let-expressions in programs like this:

```
Sort = nil: nil,
    (car: P, cdr: Xs):
        Xs.(P.Lt.Filter).(Lesser:
        Xs.(P.Ge.Filter).(Greater:
        Lesser.Sort.((car: P, cdr: Greater.Sort).Append)));

Filter = F: (nil: nil,
            (car: X, cdr: Xs): X.F.(true: (car: X, cdr: Xs.F.Filter),
                                    false: Xs.F.Filter));

Append = nil: (Ys: Ys),
         (car: X, cdr: Xs): (Ys: (car: X, cdr: Xs.Ys.Append));
```

Here my `Filter` and `Append` functions are curried, and I'm presuming that `Lt` and `Ge` are curried primitives such that, for example, `4.Lt` evaluates to a function that returns `true` for arguments that are less than 4.

If you could arrange for the `X/Xs` destructuring pattern-match to fail in `Filter` when `X.F` doesn't return `true`, maybe you could get a cleaner program. I am not clear that there is a way to do that within the semantics I have described.

If we wanted it to be more Prolog-like, we could use ; instead of ,; to be less Prolog-like, we could interchange the sense of upper and lower case, or use Ruby-like prefix colons for literal symbols instead of case, or interchange = and :. And we could potentially dispense with the . for function application. Four of the 11 other possibilities arising thus:

```
sort = Nil: Nil,
    (Car: p, Cdr: xs):
        xs (p lt filter) (lesser:
        xs (p ge filter) (greater:
        lesser sort ((Car: p, Cdr: greater sort) append)));

Sort = nil: nil;
    (car: P; cdr: Xs):
        Xs.(P.Lt.Filter).(Lesser:
        Xs.(P.Ge.Filter).(Greater:
        Lesser.Sort.((car: P; cdr: Greater.Sort).Append)))

sort: Nil=Nil;
    (Car=p; Cdr=xs):
        xs (p lt filter) (lesser:
        xs (p ge filter) (greater:
        lesser sort ((Car=p; Cdr = greater sort) append)));

sort = :nil: :nil,
    (:car: p, :cdr: xs):
        xs.(p.lt.filter).(lesser:
        xs.(p.ge.filter).(greater:
        lesser.sort.((:car: p, :cdr: greater.sort).append)));
```

Yeesh, that last one looks unparseable.

Raph Levien's Io language used, approximately, -> and a right-associative ; where I am using .(and :, thus avoiding the pileup

of right parentheses at the end. If we tried that approach and terminated each alternative in the pattern match with . (that being the only symbol whose precedence in prose is lower than ;), we might end up with something like this:

```
sort = Nil; Nil.
    (Car; p. Cdr; xs);
        xs -> (p -> lt -> filter) -> lesser;
        xs -> (p -> ge -> filter) -> greater;
        lesser -> sort -> ((Car; p. Cdr; greater -> sort) -> append.
```

This does not seem more readable to me.

Backtracking, generators, and binary relations

All of the above is predicated on the idea that a function produces at most one value — so, for example, `_ -> "a" | _ -> "b"` will never return "b" because the first match will never fail. But a natural extension of the approach is to allow any expression to yield any number of values, as in Icon, rather than just zero or one. (Partly by coincidence, Icon uses | to do something closely analogous to chaining together cases of a pattern-match.)

This, in effect, means that we are no longer talking about functions; we are talking about binary relations. With an order imposed on their pairs, perhaps.

There is a rather nice algebra of binary relations; they form a lattice (considering them as sets of (input, output) pairs with the usual set operations), a monoid (under composition), and they have a nontrivial isomorphism — the converse or inverse, which, unlike function inverses, is defined for all binary relations.

One approach to programming with general binary relations would be to attempt a binary-relation-based pattern-matching version of miniKANREN, abandoning the ordering of the possibilities. And that would be cool.

Another approach would be to follow Icon (and Python and JS) in using these generators for iteration. Io does this using no new operations, just a little bit of syntactic sugar for continuations. Python, JS, and Icon do not allow multi-shot continuations; Python and JS, like Io, reify the suspended generator, while Icon usually does not, instead identifying the generator with the expression. I'm not sure how many operations you really need to control the backtracking, but there are at least two: one that prevents further backtracking (Prolog's cut) and one that backtracks repeatedly until no further backtracking is possible, then continues.

Topics

• OCaml (p. 3602) (8 notes)
• Predicate logic (p. 3644) (6 notes)
• miniKANREN (p. 3585) (6 notes)
• Binary relations (p. 3342) (6 notes)
• Backtracking (p. 3338) (3 notes)
• Icon
• Generators

Illumination cost

Kragen Javier Sitaker, 2017-05-31 (3 minutes)

Went on a trip to Carrefour the other day and walked down the residential illumination aisle. A 500W 118mm halogen tube went for AR$25 (US$1.60) with no visible indication of its brightness, the same price as 300W, 150W, and 100W versions. An OSRAM 13W LED bulb using 13 watts, supposedly equivalent to 100W incandescent, providing 1350 lumens, was priced at AR$289 (US$18), while OSRAM compact fluorescents using 20W, supposedly equivalent to 85W incandescent, cost AR$90 (US$5.60) and provide some illegible number of lumens which I suppose is about 1150. (The iPhone Notes app peremptorily downrezzed my photos and destroyed the information I was trying to acquire.)

If we assume a luminous efficacy of 18 lm/W for the halogen, it should be 9000 lm.

Historically, the most inexpensive way to do lighting has been with straight fluorescent tubes, like the T8 size. I didn't happen to see these at Carrefour, but DMX SRL in San Nicolás in Buenos Aires is offering 58W GE T8 tubes for AR$59 (US$3.70). These are F58T8 tubes; one such on GE's site has 4900 mean lumens. Additionally, for these, you need a ballast; Tienda de LED in Chacarita offers a two-tube RTM electronic ballast for 58W T8 tubes for AR$75 (US$4.70), or US$2.35 per tube.

type	US$	W	lm	lm / W	lm/US$
halogen	1.60	500	9000	18	5625.
LED	18	13	1350	103.84615	75
compact fluorescent	5.60	20	1150	57.5	205.35714
T8 fluorescent	6.05	58	4900	84.482759	809.91736

#+TBLFM: $5=$4/$3::$6=$4/$2

The very low cost of the halogen bulbs surprises me, but it seems like it might make feasible a project I've been thinking about for some time, which is making a daylight booth in my house for mood control. Direct sunlight is about 100klux, while indirect daylight is in the 15klux range. In a reflective-walled booth, like an indoor-growing apparatus, almost all of the light would be absorbed by my skin unless the booth got very large; my skin absorbs about ⅓ of the light that falls on it, and has a body surface area of about 2m², so effectively I absorb about 0.7m² of light. Illuminating 0.7m² with 15klux requires a bit over 10klm, which means one or two of these 500W bulbs would be totally adequate. I'd need three if I were black.

Now, 1000W of illumination will require a certain amount of cooling and fire-danger attention. You could get the same 18klm out of LEDs with only 173W, but you'd need 14 bulbs like the one I saw, costing US$240 instead of US$3.20.

Topics

Free space optical coding gain

Kragen Javier Sitaker, 2019-05-08 (updated 2019-05-09) (4 minutes)

I was thinking about Wi-Fi and infrared remote controls for TVs and air conditioners today. (Did you know there used to be infrared 802.11 access points too?) Remotes don't work if you don't point them at the air conditioner (or TV or whatever), although the infrared light they emit is pretty bright, as bright as a bright flashlight. But it's modulated at a barely ultrasonic frequency; the first TV remote controls were in fact ultrasonic chimes pinged with a hammer, and bizarrely manufacturers have continued using the same frequencies despite using a completely different medium.

A remote-control receiver receives about -54 dBm of signal

I'm not totally sure, but I'm guessing that these remotes use infrared LEDs with a forward drop of about 1.5 volts, a current of about 40 mA, and an efficiency on the order of 4%. (Typical LEDs have a luminous coefficient around 3%, although modern illumination white LEDs can reach 25%, but I'm assuming that these advances haven't translated to the infrared LEDs used in remotes.) That would work out to 2.4 mW of light energy, or +3.8 dBm, which is then emitted with an "antenna gain" of about 6 dBi (thus EIRP ≈ 6.2 dBm) spread over a sphere of perhaps two meters radius (50 m² of surface) and detected by a phototransistor of radius perhaps 8 mm (5e-5 m²), so the received signal should be about 60 dB below the EIRP, or about -54 dBm.

But Wi-Fi works with orders of magnitude less than that

Wi-Fi often achieves sustained, consistent data transmission despite signal levels below -70 dBm, sometimes below -100 dBm. GPS receivers detect even weaker signals.

What does the optical environment look like for that kind of thing?

What if you used the spread-spectrum coding-gain approach used by GPS receivers and Wi-Fi receivers, but for local optical communication? You can easily modulate ordinary LEDs at over a megahertz, and now that CRT TVs are dying out, I think there aren't many sources of megahertz-range optical noise in households. Common illumination and taillight LEDs are typically pulsed, but only on the order of 100 Hz, in order to reduce switching losses; I think their 10,000th harmonic is going to be very weak, precisely because of those switching losses (aside from having a very low duty cycle even if the switching were perfect).

There *is* still the problem that direct sunlight is 100 kilolux, and even indoor room lighting is 50–500 lux, while the light you're able to transmit from an infrared LED is also only about 100 lux a few dozen millimeters in front of the LED, maybe 1 lux or less at the

sensor. Even if the infrared light coming in an open window is effectively constant over microsecond-scale time periods, it still gives rise to shot noise at your infrared sensor. We usually think of shot noise as going away as signal levels rise, but actually it increases; it's just that it increases proportional to only the square root of the signal, so the signal increases even more. But in this case the sunlight "signal" carries no information, and an increase in shot noise proportional to its square root could eventually swamp the information-carrying signal.

In the 1000-ms response time of a conventional remote control, you could transmit literally a million bits, giving you the possibility of up to 60 dB of coding gain. (I think? I'm kind of guessing about the limits of coding gain here. I should go back and really understand information theory.)

The key problem with light is that it is easily blocked, and even when multipath transmission can get it around obstacles, it's heavily attenuated. But, by the same token, there's no need to share bandwidth or avoid interfering with reserved bandwidth.

Topics

Some notes from playing 20q.net

Kragen Javier Sitaker, 2007 to 2009 (22 minutes)

Summary: 20q.net seems to do badly at choosing questions. It was able to guess the axolotl and the domestic cat; but it missed the sailboat, a gallon bottle of milk, one million, basil, a knife-sharpening stone, methyl ethyl ketone, Che Guevara's corpse, Mare Imbrium, and a leather high-heeled tango shoe.

Q24. I am guessing that it is an axolotl (Mexican salamander)?
Right, Wrong, Close
23. Can you find it in a house? Sometimes.
22. I guessed that it was a largemouth bass? Wrong.
21. Is it soft? Yes.
20. I guessed that it was a tadpole? Close.
19. Does it have a tail? Yes.
18. Does it have an exo-skeleton? No.
Is it tasty? Unknown.
17. I guessed that it was a cichlid (tropical freshwater fish)? Wrong.
Can it cheer you up? Unknown.
16. Is it larger than a pound of butter? No.
15. Does it weigh more than a duck? No.
14. Is it ferocious? No.
Does it bring joy to people? Unknown.
13. Does it live in the ocean? No.
12. Does it go in your mouth? Doubtful.
11. Does it live in salt-water? No.
10. Can it be hunted? Yes.
9. Do you know any songs about it? No.
8. Does it live in fresh water? Yes.
7. Is it a wild animal? Yes.
6. Is it an appliance? No.
5. Is it larger than a microwave oven (or bread box)? No.
4. Can it live out of water? No.
3. Does it swim? Yes.
2. Does it have whiskers? No.
1. It is classified as Animal.

Yes, it was:

You were thinking of an axolotl.
Can it live out of water? You said No, 20Q was taught by other players that the answer is Yes.
Can it be hunted? You said Yes, 20Q was taught by other players that the answer is No.
Does it bring joy to people? 20Q was taught by other players that the answer is Yes.
Can it cheer you up? 20Q was taught by other players that the answer is No.
Is it tasty? 20Q was taught by other players that the answer is No.
Contradictions Detected
The opinions of the 20Q A.I. are its own, and are based on the input of players. 20Q's answers reflect common knowledge. If you feel that 20Q is in error, the only way to correct it is to play again!
Similar Objects
a tadpole, a cichlid (tropical freshwater fish), a largemouth bass, a newt, a salamander, a paramecium (protozoa), a guppy (fish), a bullfrog, a trout (fish), a koi (fish), a leech, a Siamese fighting fish.

Well, it's hardly surprising that it had so much trouble guessing in that case. I had two "wrong" answers.

Still, it doesn't seem like 20Q is doing the best possible job of choosing questions. How often is an animal an appliance, and how often does an animal that doesn't live in salt-water live in the ocean?

I'm pretty sure that it can only live out of water if given an iodine injection to induce metamorphosis.

Thinking of a tango shoe (leather, high-heeled):

Q30. I am guessing that it is a camera?

Right, Wrong, Close
29. I guessed that it was a sneaker? Close.
28. Is it human powered? Yes.
27. Is it an absolute state? No.
26. Is it comforting? No.
25. Can it be used for recreation? Yes.
24. Is it brown? Sometimes.
23. Would you give it as a gift? Yes.
22. Would you find it on a farm? No.
21. Does it taste good? No.
20. I guessed that it was vomit? Wrong.
19. Does it open? No.
18. Can you smell it? Yes.
17. I guessed that it was a steak (beef)? Wrong.
16. Can it be painted? Doubtful.
15. Is it round? No.
14. Is it found on a desk? No.
13. Can it fit in an envelope? No.
12. Do you make something with it? No.
11. Is it living? No.
10. Does it have eyes? No.
9. Does it live in the jungle? No.
8. Is it smaller than a loaf of bread? Yes.
7. Does it live in salt-water? No.
6. Does it have a shell? No.
5. Can you play games with it? No.
4. Can it climb? No.
3. Is it black? Sometimes.
2. Is it heavy? No.
1. It is classified as Animal.

Eventually:

You were thinking of a high-heeled shoe.
You said it's classified as Animal, 20Q was taught by other players that the answer is Other.
Can you smell it? You said Yes, 20Q was taught by other players that the answer is No.
Can it be used for recreation? You said Yes, 20Q was taught by other players that the answer is No.
Is it human powered? You said Yes, 20Q was taught by other players that the answer is No.

Thinking of Mare Imbrium:

Q30. I am guessing that it is an angel?
Right, Wrong, Close
29. I guessed that it was the Moon? Close.
28. Does it contain a liquid? No.
27. Can it help you find your way? Doubtful.
26. Does it produce gas? No.
25. Would you use it daily? Doubtful.
24. Can it affect you (cause an effect to you)? Yes.
23. I guessed that it was a coral reef (marine habitat)? Wrong.
22. Can it be stolen? Doubtful.
21. Is it considered valuable? Yes.
20. I guessed that it was a snow man? Wrong.
19. Is it brown? No.
18. Is it used for entertainment? Doubtful.
17. Can you use it 24 hours a day? No.
16. Does it bring joy to people? Yes.
15. Is it hard? No.
14. Can you put something into it? Doubtful.
13. Can you smell it? No.
12. Does it make noise? No.
11. Is it larger than a microwave oven (or bread box)? Yes.
10. Do people sit on it? No.
9. Does it provide shade? No.
8. Is it stationary (movement)? Yes.
7. Is it orange colored? No.
6. Would you eat its fruit? No.
5. Is it found in mines? No.
4. Is it a geological feature? Yes.
3. Is it a metal? No.
2. Does it weigh more than a duck? Yes.
1. It is classified as Mineral.

It wasn't able to get it.

Thinking of Che Guevara's corpse:

Q30. I am guessing that it is a mermaid?
Right, Wrong, Close
29. I guessed that it was a skeleton? Close.
28. Can it be hunted? No.
27. Does it have a backbone? Yes.
26. Can you see it in a zoo? No.
25. I guessed that it was an angel? Close.
Can any age group use it? Unknown.
24. Does it like to clean itself? No.
23. I guessed that it was a composer? Close.
Can it affect you (cause an effect to you)? Unknown.
22. Does it live in the forest? No.
21. Does it like to run? No.
20. I guessed that it was Superman (fictional)? Close.
19. Is it worth a lot of money? Doubtful.
18. Can you smell it? No.
17. I guessed that it was Spiderman (superhero)? Close.
16. Does it bring joy to people? Yes.
15. Is it tall? Yes.
14. Do you find it in the sky? No.
13. Is it outside? Yes.
12. Is it cold? No.
11. Does it come in different colors? No.
10. Does it have teeth? Yes.
9. Does it have four legs? No.
8. Does it like to be petted? No.
7. Is it spotted? No.
6. Do you find it in space? No.
5. Is it useful? Yes.
4. Is it dangerous? No.
3. Is it larger than a microwave oven (or bread box)? Yes.
2. Can it growl? No.
1. It is classified as Animal.

Thinking of methyl ethyl ketone:

Q30. I am guessing that it is a stick of gum?
Right, Wrong, Close
29. I guessed that it was air? Wrong.
28. Does it make music? No.
27. I guessed that it was electricity? Wrong.
26. Is it used for communications? Yes.
25. Does it communicate? No.
24. Is it human powered? No.
23. I guessed that it was energy? Wrong.
22. Do you clean it regularly? No.
21. Can you make sounds with it ? Yes.
20. I guessed that it was mineral water? Close.
19. Does it contain words? No.
18. I guessed that it was water? Close.
17. Would you find it in an office? Yes.
16. Can it be used for recreation? Yes.
15. Would you use it daily? Yes.
14. Does it bring joy to people? Yes.
13. Is it hard? No.
12. Does it have a hard outer shell? No.
11. Can it be placed on your head? Yes.
10. Do you eat it with Thanksgiving dinner? Sometimes.
9. Is it tasty? Sometimes.
8. Is it usually sliced or carved? No.
7. Can you walk on it? No.
6. Does it flower? No.
5. Can you add it to a salad? Yes.
4. Can you use it with your friends? Yes.
3. Is it larger than a pound of butter? Irrelevant.
2. Is it decorative? No.
1. It is classified as Mineral.

Thinking of a rectangular knife-sharpening oilstone:

Q30. I am guessing that it is lipstick?
Right, Wrong, Close
29. I guessed that it was an electric toothbrush? Wrong.
28. Does it have a hole in it? No.

27. Is it made out of wood? No.
Is it pleasurable? Unknown.
26. Is it larger than a microwave oven (or bread box)? No.
Can it cheer you up? Unknown.
25. Would you find it in an office? No.
24. Can it bend without breaking? No.
23. I guessed that it was dill? Wrong.
22. Is it straight? Yes.
21. Is it worth a lot of money? No.
20. I guessed that it was a piccolo? Wrong.
19. Is it connected to a wire? No.
18. Does it cut? No.
Can you play games with it? Unknown.
17. Do you wear it? No.
16. Would you wear it on your wedding day? No.
15. Does it come in a pack? No.
14. Does it get shorter from using it? Yes.
13. Would you use it daily? Yes.
12. Would you give it as a gift? Yes.
11. Does it smell sweet? No.
10. Does it bring joy to people? Sometimes.
9. Does it come in many varieties? Yes.
8. Can you walk on it? No.
7. Is it made of crystals? Yes.
6. Can it blink? No.
5. Does it come in a box? Yes.
4. Is it round? No.
3. Is it used to make jewelry? No.
2. Do you hold it when you use it? Yes.
1. It is classified as Mineral.

Thinking of basil:

Q30. I am guessing that it is breakfast?
Right, Wrong, Close
29. I guessed that it was lunch? Wrong.
28. Would you find it on a farm? Yes.
27. Do you use it at night? Sometimes.
26. I guessed that it was marmalade? Wrong.
25. Would you find it in an office? Doubtful.
24. Is it a specific color? Sometimes.
23. Does it go inside other things? Sometimes.
22. I guessed that it was dinner? Wrong.
21. Does it contain a liquid? Yes.
20. I guessed that it was ballet? Wrong.
19. Can it be used for recreation? Sometimes.
18. I guessed that it was an ear of corn? Wrong.
17. I guessed that it was salad? Close.
16. Does it smell sweet? Yes.
15. Does it live in the forest? No.
14. Is it flat? Sometimes.
13. Is it red? No.
12. Does it bring joy to people? Yes.
11. Does it get really hot? No.
10. Is it colorful? Yes.
9. Is it used in Oriental cooking? Yes.
8. Does it taste good with butter? Yes.
7. Is it a leafy vegetable? Yes.
6. Does it have a backbone? No.
5. Is it round? No.
4. Can you peel it? No.
3. Does it come in different colors? Yes.
2. Is it crunchy? Yes.
1. It is classified as Vegetable.

However, apparently my opinion of basil is different from most people's:

You were thinking of basil.
Is it crunchy? You said Yes, 20Q was taught by other players that the answer is No.
Does it come in different colors? You said Yes, 20Q was taught by other players that the answer is No.
Is it used in Oriental cooking? You said Yes, 20Q was taught by other players that the answer is Doubtful.
Is it colorful? You said Yes, 20Q was taught by other players that the answer is

No.
Is it flat? You said Sometimes, 20Q was taught by other players that the answer is No.
Does it smell sweet? You said Yes, 20Q was taught by other players that the answer is No.
Can it be used for recreation? You said Sometimes, 20Q was taught by other players that the answer is No.
Does it contain a liquid? You said Yes, 20Q was taught by other players that the answer is No.
Do you use it at night? You said Sometimes, 20Q was taught by other players that the answer is No.
I don't know what kind of people think that basil doesn't smell sweet and isn't used in Oriental cooking.

Thinking of a gallon bottle of milk:

Q30. I am guessing that it is a phone book?
Right, Wrong, Close
29. I guessed that it was a cigarette case? Wrong.
28. Do you open and close it? Yes.
27. I guessed that it was a tissue box? Wrong.
26. Is it white? Yes.
25. Is it black? No.
24. Is it round? No.
23. I guessed that it was a bottle of water? Close.
22. Would you use it in the dark? Yes.
21. Do you put things in it? No.
20. I guessed that it was a revolver (hand gun)? Wrong.
19. Would you give it as a gift? No.
18. Can it be refilled? Yes.
17. I guessed that it was a notebook (paper)? Wrong.
16. Can you use it with your friends? Yes.
15. Does it have writing on it? Yes.
14. Is some part of it made of glass? No.
13. Does it have a cable? No.
12. Is it used by the police? Yes.
11. Is it used for entertainment? No.
10. Do you use it at work? Yes.
9. Is it alive? No.
8. Can it fit in an envelope? No.
7. Does it dig holes? No.
6. Can it be used to put things together? No.
5. Is it small? Yes.
4. Does it have short fur? No.
3. Is it brown? No.
2. Can you see it in a zoo? No.
1. It is classified as Animal.

Q17. I am guessing that it is a domestic cat?
Right, Wrong, Close
16. Does it have a long tail? Yes.
15. Is it small? Yes.
Does it come from something larger? Unknown.
14. Does it live in large populations? Sometimes.
13. Do you know any songs about it? Yes.
12. Is it a carnivore? Yes.
11. Can it be trained to obey commands? Doubtful.
10. Does it eat fish? Yes.
9. Is it spotted? No.
8. Is it a herbivore? No.
7. Does it usually live on a farm? No.
6. Does it come from space? No.
5. Does it serve a purpose? No.
4. Is it awake at night? Yes.
3. Is it a small mammal? Yes.
2. Can you find it in a house? Yes.
1. It is classified as Animal.
You were thinking of a domestic cat.
Does it live in large populations? You said Sometimes, 20Q was taught by other players that the answer is No.
Does it come from something larger? 20Q was taught by other players that the answer is No.

Thinking of the number 1 000 000:

Q30. I am guessing that it is cyberspace?
Right, Wrong, Close
29. I guessed that it was a language? Wrong.
28. Is it printed? Sometimes.
27. Is it used during meals? Irrelevant.
Can it add? Unknown.
26. I guessed that it was information? Close.
25. Is it healthy? Irrelevant.
24. Is it used by the police? Yes.
23. Can you buy it? Irrelevant.
22. Can it be used for recreation? Yes.
21. Does it have cash value? Irrelevant.
20. I guessed that it was knowledge? Close.
19. I guessed that it was mathematics? Close.
18. Is it taught in school? Yes.
17. Can it live out of water? Irrelevant.
Does it involve contact with other humans? Unknown.
16. Can you read it? Irrelevant.
Was it invented? Unknown.
15. Is it used for entertainment? Yes.
14. Does it have a title? Yes.
Is it man made? Unknown.
13. Is it something you bring along? Irrelevant.
12. Do you look at it? No.
11. Does it move? No.
10. Is it used to calculate? Yes.
9. Is it mechanical? No.
8. Do you use it at work? Yes.
7. Does it need love? No.
6. Does it have a cold nose? No.
5. Can it be heard? Irrelevant.
4. Is it small? No.
3. Does it get wet? No.
2. Do you hold it when you use it? Irrelevant.
1. It is classified as Other.

Thinking of a sailboat:

Q30. I am guessing that it is a power boat?
Right, Wrong, Close
29. I guessed that it was a boat? Close.
28. Can it float? Yes.
27. Would you pay to use it? Sometimes.
26. Is it mechanical? Yes.
25. Is it very large? Sometimes.
24. Do you hold it when you use it? Sometimes.
23. Was it used over 100 years ago? Yes.
22. Does it have four wheels? No.
21. Does it open? No.
20. I guessed that it was an outboard motor (motorboat engine)? Close.
19. Does it roar? No.
18. I guessed that it was an accordion? Wrong.
17. Does it move? Yes.
16. Do you use it at night? Yes.
15. Does it roll? No.
14. Would you find it on a farm? Sometimes.
13. Does it have lots of buttons? Sometimes.
12. Does it have buttons? Yes.
11. Can it be heard? Yes.
10. Is it black? Sometimes.
9. Is it connected to a wire? No.
8. Does it break if dropped? Yes.
7. Does it have a trunk? No.
6. Does it grow in the Southern U.S.? No.
5. Is it larger than a microwave oven (or bread box)? Yes.
4. Is it used for entertainment? Yes.
3. Is it flat? No.
2. Is it something you bring along? Yes.
1. Is it hard? Yes.
It is classified as Unknown.
You were thinking of a sailboat.
You said it's classified as Unknown, 20Q was taught by other players that the answer is Other.
Is it something you bring along? You said Yes, 20Q was taught by other players that the answer is No.
Is it black? You said Sometimes, 20Q was taught by other players that the answer

is No.
Can it be heard? You said Yes, 20Q was taught by other players that the answer is No.
Does it have buttons? You said Yes, 20Q was taught by other players that the answer is No.
Does it have lots of buttons? You said Sometimes, 20Q was taught by other players that the answer is No.
Would you find it on a farm? You said Sometimes, 20Q was taught by other players that the answer is No.
Do you hold it when you use it? You said Sometimes, 20Q was taught by other players that the answer is No.

At this point it has played 64 million games.

Topics

TV oscilloscope

Kragen Javier Sitaker, 2017-04-10 (updated 2017-06-20) (4 minutes)

Electronics hacker "GreatScott!" made a video using a TV CRT as a "crude" oscilloscope.

In the video he shows that the horizontal scan normally runs at some 15kHz, and its return at the end of the scan is not very clean. He demonstrates visualizing waveforms of a few times that frequency.

One difficulty is that the deflection coils in a TV CRT produce a deflection that's proportional to their current, but he's driving them with a voltage signal, and they are primarily inductive loads — one consequence is that they more strongly attenuate higher frequencies, but another is that they produce frequency-dependent phase shifts (of nearly 90°) and thus badly deform the waveforms.

It seems like a straightforward solution to this problem is to drive the coils from a current source rather than a voltage source, so that up to some potentially fairly high voltage the magnetic field will faithfully reproduce the desired signal.

Scott's TV originally used a -42V to +30V signal for the horizontal scan at a bit over 15 kHz (with a voltage waveform that was a distorted square wave, thus producing a distorted triangle wave of current for the scan), with a 600mΩ resistance and 210μH of inductance. The vertical scan used -10 to +40V at 43 Hz, with 21Ω resistance and 20mH of inductance.

In both cases there were strong harmonics in the signals.

It seems to me that if you want to make this a useful oscilloscope tube, your best bet is to turn the screen sideways, because the "vertical" deflection of your signal basically always needs to have higher frequency components than the "horizontal" scan. If a ±40V signal is adequate to recognizably reproduce a deflection of some 150kHz, then by using a current source with compliance up to 40kV, we should be able to successfully reproduce 150MHz. The coil probably would not survive 40kV, though, as special coil construction is needed to handle voltages over a few hundred volts, so we could probably only reach a few MHz in practice without rewinding the coil.

At the 15kHz fundamental, the 210μH inductance represents about 20 ohms (making its 0.6Ω real resistance insignificant at that frequency or above), so getting a full deflection requires on the order of an amp or two through the coil.

Purpose-designed oscilloscope tubes differ in a few ways: they use electrostatic deflection, which I think is not a thing you can retrofit to the tube afterwards; are longer so that smaller angles are feasible; and have an axial voltage gradient to progressively accelerate the beam so that the electrons are moving more slowly as they go through the deflection plates than when they hit the screen. (In part this is to compensate for some disadvantages of electrostatic deflection: it deflects fast electrons less than slow ones, so the speed must be relatively low and consistent across all the electrons in order to get a large and consistent deflection. Magnetic deflection does not have these problems; the radius of the electron track depends only on the magnetic flux density, not on its speed.)

A computer monitor might be more promising. This TV had only about 350 lines per scan and a very low refresh frequency, so it could get by with a low horizontal scan frequency. By contrast, the LG 710e monitor someone threw out the other day is reported to support 1024×768 at 85Hz. This means that its normal horizontal scan must run at at least 65kHz. I don't know if it uses a higher voltage than ±40V or not; it would seem like using a coil with half as many turns would give you a quarter the inductance but also half the field, which would be a win.

(See also files VCR oscilloscope (p. 213), Laser printer oscilloscope (p. 449), and CCD oscilloscope (p. 1861).)

Topics

Full res globe

Kragen Javier Sitaker, 2014-02-24 (1 minute)

Suppose you want to make a globe with outrageously high resolution. Maybe you want to map every street in the world, or every car. How much space do you need?

The Earth is almost exactly 40 000 000 meters around (the small error in this number is due to long-ago measurement mistakes), and to see every street, you need no worse than, say, 15-meter resolution — which you can get from, say, Landsat. OpenStreetMap may be a more interesting data source.

40 million divided by 15 gives you 2⅔ million pixels around your globe. If you're using a regular 600dpi laser printer, your globe is 370 feet in circumference, or 18 meters in radius, 36 in diameter. Your globe will be about 14 stories tall.

To reduce that to a single story tall, you'd need to reduce the size by 16×, to 9600dpi. At this resolution, your pixels are only 2.6 microns wide: within the range of what you can see with a light microscope, but not easily.

However, 9600dpi is somewhat difficult to achieve. 2400dpi is supposedly achievable on high-end inkjet printers, and you might be able to use photographic reduction processes to reduce them 4:1.

Topics

• Graphics (p. 3483) (91 notes)
• Microprint (p. 3582) (8 notes)
• Printing (p. 3649) (7 notes)
• Geographical information systems (GIS) (p. 3473) (3 notes)
• OpenStreetMap (p. 3615) (2 notes)
• Landsat

Scriptable windowing for Wercam

Kragen Javier Sitaker, 2018-10-26 (updated 2019-07-24) (26 minutes)

One of the designs I'm thinking of for window systems for BubbleOS (see Speculative plans for BubbleOS (p. 2128)) is sort of based on MGR, the Blit, and Plan 9, with some ideas from NeWS (as filtered through the UNIX-HATERS Handbook), Bitcoin, eBPF, and Dan Luu's studies of user interface latency. It's also partly inspired by thinking about might-have-beens about the VT100, the H19, and the Datapoint 2200 from the 1970s.

However, see also A nonscriptable design for the Wercam windowing system (p. 3092).

Basically, the issue is that user interface latency is a really big deal when it comes to the "feel" of an interactive system, and modern computer software design is making our user interface latency suck shit, to the point that most modern computers are dramatically less responsive than an Apple //e. The Apple //e takes 30 ms to get a keystroke on the screen, while a typical modern machine takes 100–160 ms, and a tuned modern machine takes 60 ms. And this gets worse rather than better when we try to operate our systems over long-latency links, which are, despite all predictions, increasingly common in today's world, despite the concurrent proliferation of low-latency links.

One approach to the link latency problem is the Lotus Notes strategy: partially replicate the application state on different nodes, including some as close to the user as possible, and reconcile the occasional conflicting update after the fact. This is more or less the modern AJAX strategy, but it doesn't help much with user-interface latency as such.

A different approach, which is sort of the original use case for JS, is to put a small amount of application-specific code as close as possible to the client to react quickly to input events, while delegating more complex processing to the client application. As Don Hopkins explains:

> At the mere mention of network window systems, certain propeller gheads who confuse technology with economics will start foaming at the mouth about their client/server models and how in the future palmtops will just run the X server and let the other half of the program run on some Cray down the street. They've become unwitting pawns in the hardware manufacturers' conspiracy to sell newer systems each year. After all, what better way is there to force users to upgrade their hardware than to give them X, where a single application can bog down the client, the server, and the network between them, simultaneously!
>
> The database client/server model (the server machine stores all the data, and the clients beseech it for data) makes sense. The computation client/server model (where the server is a very expensive or experimental supercomputer, and the client is a desktop workstation or portable computer) makes sense. But a graphical client/server model that slices the interface down some arbitrary middle is like Solomon following through with his child-sharing strategy. The legs, heart, and left eye end up on the server, the arms and lungs go to the client, the head is left rolling around on the floor, and blood spurts everywhere.
>
> The fundamental problem with X's notion of client/server is that the proper division of labor between the client and the server can only be decided on an application-by-application basis. Some applications (like a flight simulator) require that all mouse movement be sent to the application. Others need only mouse

clicks. Still others need a sophisticated combination of the two, depending on the program's state or the region of the screen where the mouse happens to be. Some programs need to update meters or widgets on the screen every second. Other programs just want to display clocks; the server could just as well do the updating, provided that there was some way to tell it to do so.

The right graphical client/server model is to have an extensible server. Application programs on remote machines can download their own special extension on demand and share libraries in the server. Downloaded code can draw windows, track input eents, provide fast interactive feedback, and minimize network traffic by communicating with the application using a dynamic, high-level protocol.

As an example, imagine a CAD application built on top of such an extensible server. The application could download a program to draw an IC and associate it with a name. From then on, the client could draw the IC anywhere on the screen simply by sending the name and a pair of coordinates. Better yet, the client an download programs and data structures to draw the whole schematic, which are called automatically to refresh and scroll the window, without bothering the client. The user can drag an IC around smoothly, without any network traffic or context switching, and the server sends a single message to the client when the interaction is complete. This makes it possible to run interactive clients over low-speed (that is, slow-bandwidth) communication lines.

From a security point of view, the idea of uploading arbitrary application code to the window server is somewhat alarming. What if it hangs the window server, or makes it run slowly or unresponsively? What if one application uploads a malicious version of a widget used by another application?

Putting a small amount of application code into the input event handling path is also the approach taken by BPF for packet filtering: tcpdump uploads a tiny bytecode program into the BSD kernel to discard uninteresting packets *in interrupt context*, forwarding the interesting packets to the user-level program. This avoids context-switching to the tcpdump process for every network packet, which was seriously important in 1992 when BPF was designed, and even more important in 1980 when its stack-machine predecessor, CSPF, was designed.

BPF's instruction set has 22 instructions and two registers, but only supports forward jumps, so it cannot express loops. eBPF, which includes things like functions and user-accessible data structures, is now used by Linux not only for packet filtering, but also for sandboxing processes (a purpose for which MacOS uses a tiny Scheme), inserting debugging and performance probes into the kernel, XXX

Uses of scriptability in Wercam

Suppose that application programs can upload event-handler scripts for a similar virtual machine into the Wercam window server, which then compiles and runs them. Then maybe those scripts can provide quicker responsiveness than the applications themselves, because they're running in the process that directly draws into the framebuffer. Indeed, maybe at some point we can move that process out of the time-sharing operating system the applications are running on, into a real-time executive like the one in RTLinux, or even onto dedicated display-handling hardware. Then we can get the latency of user interface responsiveness down into the submillisecond range where the delay really is undetectable by humans.

But to guarantee this kind of responsiveness, especially without rewriting major parts of your application in this strange virtual

machine, we must limit the amount of context available to the event-handler script; and it might not produce the ultimately correct display immediately. In a lot of cases, this is probably okay. Mosh does something similar with typing: when you type a printable character, it immediately adds it to your screen, and then it reconciles the screen image with the remote host, so if the character didn't get echoed by the application there, it disappears.

What kinds of things might you reasonably add?

Input event selection

Well, one simple example is that you can specify which events you would like the window server to send to the client. Don's example of how some applications would like all mouse movement, while others just want clicks, is the simplest example. You can implement it easily if your event-handler script is in charge of sending the events back to the client; it can simply return without doing anything when the mouse-button state hasn't changed since the last event. This mechanism is more complicated than the X11 mechanism:

```
/* Apparently StructureNotifyMask is what you use to get MapNotify
 * events? */
XSelectInput(w->display, w->window, KeyPressMask
             | KeyReleaseMask
             | ButtonPressMask
             | ButtonReleaseMask
             | PointerMotionMask
             | StructureNotifyMask
);
```

But it's probably a lot easier to debug!

Bit gravity

X11 has a "bit gravity" feature which modifies window resize handling. If the bit gravity of a window is set to, for example, `NorthEastGravity`, then when a window is resized, the previous window contents are in the upper right corner of the screen ("northeast" in the conventional mapping of compass directions to maps). This matters because there is a span of time between the window resize on the server and when the application may be able to respond by redrawing the window. Indeed, when X was designed in the 1980s, window redraws often took several video frames, even when there were no issues of system load.

Input event translation

A more interesting example — and this is where we get into MGR and Blit territory — is that if the client's event-handler script is responsible for sending the events to the client, maybe it can also determine the form of those events. Like, maybe you could add a mouse-compatibility layer in front of some program written for a character-cell terminal, just by defining a few event handlers.

Application-to-display stream parsing?

Of course, that suggests that the event-handler script could *also* be in charge of the handling of data *from the client*. And then maybe you could, as the Blit does, start out in a glass-tty mode with some default

event handlers that provide glass tty behavior, and then upload new event handlers with escape sequences.

That's probably not actually ideal, for a couple of reasons. First, graphical applications can easily involve sufficiently massive data volumes that we want to minimize the number of copies of that data. (I'm typing this on a 1920×1080 32bpp 60Hz screen, which multiplies out to 497,664,000 bytes of pixel data per second. mpv can actually reliably get that data on the screen, using OpenGL. We probably don't want that data flowing through an unaligned FIFO, and we probably don't want client bytecode to be in charge of parsing it, because despite the potential advantages of XXX

It's probably better to maintain the application-to-display protocol fixed, but include in it the ability for the application to directly invoke previously-uploaded event handlers, including event handlers that won't fire on any window-system-generated events.

Vertical sync for tear-free redrawing

Another interesting example is the unfortunately-named† XSYNC extension, intended to provide tearing-free and flicker-free double-buffering for X11, as well as audio/video synchronization. It was proposed in 1991 but the Linux implementation still lacks access to the vsync signal needed to make it work, despite some work on this in 2003. XSYNC works by corking up the pipeline of drawing requests from a given client until some condition becomes true, such as a given timestamp passing, a counter incrementing, or the monitor going into vsync. Then the pipeline gets uncorked and updates the screen during the vertical blanking interval (VBI) or renders the next animation frame or whatever.

† "Unfortunately named" because an almost, but not completely, unrelated core X11 function is called `XSync()`.

Aside from the clumsy special-casing in the `XSyncWaitCondition` struct (which you can read about in the docs if you're interested), implementing vertical synchronization just by uncorking a drawing request pipeline excludes many optimizations we would like to do. Consider what Massalin said about terminal emulation in Synthesis:

> The numbers in Table 7.5 can be used to predict the elapsed time for the "cat /etc/termcap" measurement done in Section 7.2.2. Performing the calculation, we get 3.4 seconds if we ignore the invocation overhead and use only the per-character costs. Notice that this exceeds the elapsed time actually observed (Table 7.2). This unexpected result happens because Synthesis kernel [sic] can optimize the data flow, resulting in fewer calls and less actual work than a straight concatenation of the three quajects would indicate. For example, in a fast window system, many characters may be scrolled off the screen between the consecutive vertical scans of the monitor. Since these characters would never be seen by a user, they need not be drawn. The Synthesis window manager bypasses the drawing of those characters by using fine-grained scheduling. It samples the content of the virtual VT100 screen 60 times a second, synchronized to the vertical retrace of the monitor, and draws the parts of the screen that have changed since the last time. This is a good example of how fine-grain scheduling can streamline processing, bypassing I/O that does not affect the visible result. The data is not lost, however. All the data is available for review using the window's scrollbars.

This kind of optimization — which, incidentally, improves predictability and reduces user-interface latency — can't be done simply by uncorking a drawing pipeline at a predetermined time. You have to avoid sticking useless drawing operations into the drawing pipeline in the first place.

It would, on the other hand, be easy to implement this drawing approach as an event-handler script, although not without some kind of looping capabilities. Note that in this case the loop indexes over the terminal screen contents, which is a finite-size data object, and so it's amenable to worst-case execution-time guarantee calculation — just not in such a straightforward way as BPF's loop-free virtual-machine instruction set. (And you could limit execution time by, for example, inserting an Ethereum-style gas check before each backward jump or potentially-recursive function call, although a static check seems perhaps more desirable, since it moves the error reporting to a client request instead of a failed window redraw.)

(Despite the complaining in the links above, it's actually reported to be possible to get tear-free redraws under X11; GLX is reported to be adequate when used properly, but I think I do see tearing in mpv using an OpenGL visual.)

Terminal emulators may be a somewhat unusual case, though. The "source" contents of a terminal window is small (typically a few kilobytes) and bounded, so it's safe for a window server to traverse it during the VBI; and it's much smaller (typically 200× smaller, though this varies by font size and pixel depth) than the pixel data in the window, updating the "source" is inherently much cheaper than updating the pixel data; and the source may suffer massive quantities of updates while it's not visible. Other kinds of applications (hypertext browsers, movie players, PDF viewers) might not have all these properties.

So it may not be ideal to design Wercam around what's best for terminal emulators.

However, tear-free vertically-synced redrawing is desirable for many applications.

Window-server-side text editing

However, modern GUI programs are full of text fields, and each text field is a full-fledged text editor. Much of the time, the app doesn't care about the intermediate values of the text field, only its eventual value. Rather than simply doing local echo of keystrokes, ideally the vast majority of keystrokes and mouse clicks could be handled by the window server without any interaction with the application process at all, until it wants the contents of the text field, or the user sends some keystroke that the text field doesn't handle.

This requires the window location and shape, font metrics, font glyphs, and buffer contents to be all maintained in the window server, where they're accessible to the client-sent event handlers. The font glyphs in particular are a potential problem, given the rendering time and total size of full Unicode fonts, so probably some kind of glyph-rendering-on-demand is needed, with some kind of placeholder displayed for not-yet-loaded glyphs.

With Unicode, combining characters and bidirectional text rendering are another big ball of hassle that is probably best not uploaded in tiny event handlers.

But even handling the most basic actions with low latency — cursor positioning with the mouse or arrow keys, inserting characters, deleting characters with backspace — would go a long way to reducing the latency of user interaction with text fields.

Keyahead and mouseahead for menus

One of the benefits of Don Hopkins's implementation of pie menus in the window server in NeWS was "mouseahead": since mouse movements that followed the click that brought up the pie menu weren't processed until the pie menu was already active, there's no window of time during which a too-fast click or release will be lost. This is analogous to how you can "key ahead" or "type ahead" in keyboard-driven menu or data-entry systems; you don't need to wait for the application to display a menu before selecting from it, or to move the cursor to the next field before typing into it. The same sequence of actions will always have the same result regardless of details of timing or how fast the application manages to respond, which makes the system feel better, allowing the user to control it quickly with open-loop control.

A very simple event-handler script is adequate to track the user's position in a menu tree and navigate it by mouse movements.

"Keyahead" is a rather old term, appearing for example in the TRS-80 Model II DOS manual:

Keyaheads

TRSDOS allows keyaheads of up to 80 characters. This means that you can type in the next command while previous ones are being executed.

Note: A keyahead will not be displayed until TRSDOS or the application program is ready to interpret it.

Unfortunately I don't know of a modern term for the phenomenon.

MATE, the desktop environment I'm using at the moment, has a serious failure of both keyahead and mouseahead. It has a Microsoft-Windows-95-style "Start" menu, which can be activated either by clicking the menu button or by pressing the Microsoft Windows key on the keyboard. When it hasn't been used for a while, though, it gets slow (presumably relevant stuff needs to be read from disk) and subsequent keys or mouse actions are captured by whatever the foreground application window is, rather than being routed to the menu. This means that, even when the menu opens quickly, you have to wait for it to appear before starting to select things from it.

Animation, compositing, and filtering

It's beneficial for filtering (for e.g. text drop shadows) to run in the display server rather than the app.

Other protocol considerations

Applicaiton embedding (e.g. a Flash player or movie player in a web browser) will really benefit a lot if there's a way to avoid copying all the pixel data first from one application to the other, then to the server.

Tentative protocol design

The primitive unit of drawing is a 32×32-pixel tile in premultiplied 32-bit RGBA format with the bytes in that order. This representation is chosen for a few reasons:

• It's 4096 bytes, the usual memory page size on i386 and amd64 systems. That means that in theory you should be able to transfer tile buffers between clients and servers on the same machine without copying them, just by remapping memory, at least if they're properly aligned. It may have undesirable cache effects for operations on

corresponding pixels in multiple tiles, but typically these are done on only three or four tiles at a time, and all modern L1D caches are at least four-way set-associative.

• A single scan line of it is 32 pixels, which is 128 bytes. This is larger than, and divisible by, the cache line size on currently popular desktop CPUs, which is 64 bytes. It's also larger than any of the SIMD registers in current popular CPUs, which max out at 512 bits (64 bytes). A single color plane of it would be only 32 bytes, which would be smaller than a 512-bit AVX512 register, but would fit in the more commonly used 256-bit SIMD registers.

• It's smaller than typical L1D cache sizes (≈64 KiB), so you can fit several tiles in the L1 cache at once.

• It's large enough that most pixels (87.9%) are interior pixels, not border pixels.

• It's large enough that unaligned blitting from one tile to another will still only run into tile boundaries every 16 pixels on average.

• It's small enough to keep wasted space in tiles that run off the edge of a window to a tolerable level. To take a random window size that would fit on this display, a 949×1067 window is 30×34 tiles, or 31×35 tiles if none of its edges are aligned on tile boundaries. That's a total of 1,111,040 pixels, while the window as such contains 1,012,583 pixels, so you have 9% waste. Windows I actually have open on this display have sizes like 800×482, 656×362, and of course 1920×1080. Many of these are divisible by 16; some are even divisible by 32.

• It's the pixel format this video system uses. Most video systems, in fact. And it's easy to work with. And it has alpha. It has the great disadvantage that it's twice as big as the 16-bit and 15-bit formats common in mobile phone displays and video formats.

Tiled images can efficiently support filtering and rescaling operations, which, like alpha-compositing, are fundamental to currently fashionable GUIs.

There are two ways to draw things: either send a tile from the application to the display server, sticking it into a tile-aligned position on some pixmap, or blit some pixels from one pixmap to another. Blitting uses the premultiplied version of the Porter–Duff "over" operator to modify the destination pixmap with the source pixmap, as in Plan9's 8½ or the usual use of the XRENDER extension.

Blitting coordinates can exceed the boundaries of the source pixmap, which is not an error; instead, the pixmap wraps. Destination pixmap coordinates are clipped as you would expect.

Note that this leaves out the "hold out of" and "add" Porter–Duff operations, as well as the possibility of using an alpha channel from another source, which are present in the XRENDER Composite request.

When the application creates a thing, such as a pixmap, a data buffer, or an event handler, it specifies the identifier it will use in the future to identify that thing. This eliminates the round trips that would be needed if the display server were to be responsible for allocating these identifiers, as in X11 and 8½. The application must first request resource allocations to be used by these later creations; the resource allocation may be denied, but if it is granted, the creations are guaranteed to succeed precisely when they do not exceed the allocated quantity.

Duplicate tiles are not coalesced in the memory of the display

server, as they are in XRENDER. This is because, in any case, this could not be visible to applications, for security reasons. Reducing the charge against the application's pixel quota would make applications unstable in a difficult-to-debug way; they would fail to allocate sufficient pixels only when certain other applications were running. Pixmaps are writable, and Wercam is designed to provide guaranteed real-time responsivity, which means that even copy-on-write tile sharing is probably a bad idea, because it means that writing a pixel might vary in speed by orders of magnitude depending on whether that tile was shared. (And this is a side-channel information leak if the timing is observable to the application, which it surely would be.)

Another desirable property of the system is that you should be able to capture and replay a protocol stream, as you can with ttyrec, and in particular that it should be possible to generate a "checkpoint" containing just the protocol messages necessary to get the window into the current state, so that a checkpointed application can be reconnected to a fresh window server, if desired.

Event types:

- VSync
- Timer expiration
- Mouse events
- Keyboard events
- Touch events (start, end, move)
- Screen resize
- Error
- Window close

Topics

$1 recognizer diagrams

Kragen Javier Sitaker, 2019-08-11 (updated 2019-10-24) (15 minutes)

I read the 2007 "$1 recognizer" paper the other day. It describes a simple "unistroke recognizer" you can implement in 100 lines of code or so to experiment with pen-based gestures, like PalmPilot Graffiti; they report on a user study on an Itsy (or rather an iPAQ) where it compared reasonably well to a couple of other stroke recognition algorithms popular in the research literature.

It occurred to me that something like the $1 recognizer might be a viable solution to my problem in Dercuano drawings (p. 64), namely, how to illustrate the notes in Dercuano without using either too much of my time or too many bytes.

The $1 recognizer algorithm

For the $1 recognizer, a pen gesture consists of a sequence of (x, y) points; its task is to match this sequence against a collection of "template" gestures and provide a list of gestures that match most closely.

The first point in the sequence is where the pen first touched the tablet, and the others are samples of the pen's position taken over time until it is lifted. These trace out some kind of more or less continuous curve on the display, but the position of each individual point along that curve is just a matter of how far the pen had gotten when the sampling clock fired.

So, the first thing the algorithm does is resample the curve to a uniform point spacing, using a fixed, predetermined number of points that is the same for every gesture. They found that numbers in the range 64–256 worked best. The resampling algorithm they used is fairly naïve, just linearly interpolating between points. This step normalizes the drawing speed and the relative phase of the sampling clock and the drawing operations, so that exactly the same curve traced at different speeds will result in very nearly the same set of points.

People are also not absolutely precise about the angle, position, scale, and aspect ratio of their pen gestures, so the next steps in the algorithm are to try to normalize out these features. The points are rotated around their centroid so that the gesture starting point is at 0 degrees. The X and Y coordinates are then given independent affine transforms to fit them into a predefined 1×1 bounding box, thus eliminating variations of position, scale, and aspect ratio. Finally, this normalized sequence of points is compared against each (normalized) template by calculating the average Euclidean distance between corresponding points — not once, but with a series of different candidate rotations using golden-section search, based on their observation that the rotation-distance function had no local minima on correct matches.

Remarkably, they reported 99% accuracy on a vocabulary of 16 unistroke symbols in their tests, varying very little with stroke speed; the fastest strokes were around half a second, so this amounts to around 8 bits per second of input bandwidth.

The nature of diagrams

Diagrams have a few freehand lines, but are mostly symbolic — aside from textual labels, they mostly consist of many repetitions of a relatively small number of different symbols, at different positions, orientations, and sometimes scales or aspect ratios. So bubble charts contain bubbles, connecting lines, and arrowheads; circuit diagrams contain resistors, capacitors, inductors, grounds, horizontal and vertical wires, rectangles representing chips, junctions, and so on; various kinds of diagrams contain different kinds of boxes. Some of these kinds of symbols can be freely rotated, while others cannot. Other variations between different instances of the same symbol are usually random and need not be stored.

Also, diagrams commonly contain significant topology, quite aside from their possibly significant geometry, and often it's helpful to "snap" certain connection points together, and if a symbol is later moved, to update connected symbols so that they remain connected.

Diagrams in such a form can be quite reasonably represented as a set of references to symbols, each associated with a position, orientation, scale, and possibly aspect ratio. The symbols themselves can possibly be shared between multiple diagrams or embedded in one diagram.

Drawing diagrams with pen gestures

It occurs to me that a pen gesture interface is probably one of the most straightforward ways to create diagrams; it offers the possibility of smoothly scaling from a pure sketching interface to a much more formal interface. The basic idea is that you put a series of strokes on a canvas, and the system (initially devoid of symbol definitions) tries to figure out which strokes belong to the same symbol, using something like the $1 recognizer; when it gets it wrong, you correct it after the fact.

By translating, rotating, and scaling the graphic for each symbol to match the original stroke you input it with, the system redraws your diagram. By remembering all of your strokes, it gradually improves the definition of each symbol, improving the appearance of your diagram, without overly interfering with the sketching process.

Furthermore, you can explicitly edit symbols; for example, in a circuit diagram, you might want your ordinary wire symbols to be only horizontal or vertical, so you might want non-rotatable horizontal and vertical line symbols. A small vocabulary of such non-rotatable symbols (boxes, ellipses, lines, arrowheads) would be adequate for many quick diagramming tasks. You might want to add or move attachment points, add synonym templates, toggle rendering of noise, add additional rendered decorations you don't need to sketch explicitly (including other symbols, as in Recursive curves (p. 1948)), and so on.

If you were to write text in such an interface, it would ideally discover the relatively small number of letters you were using and represent your "diagram" as a list of letters and their positions.

Aside from the above fuzzy spectrum between defining symbols and freehand drawing, you'd probably want the usual kinds of drawing operations: undo, redo, move, rotate, scale, multiple selection, and so on.

Doing it on hand-computer touchscreens

Realizing such a fluent interface with the non-ideal hardware available is a substantial challenge. In practice multitouch cellphone touchscreens are what I have available, and these have relatively crude touchstart and touchend resolution, although they're fairly precise during the touch (and some of them are even fast, like 60 Hz, while others are more like 10 Hz); moreover, they have major finger occlusion problems.

Using the quasimodal multitouch ideas outlined in Interactive calculator (p. 2771), Two-thumb quasimodal multitouch interaction techniques (p. 1765), and Interactive geometry (p. 508), I think this can be overcome: a transparent virtual stylus projects from your finger upon first touch, and a button elsewhere on the display starts the ink running out of it. This allows you to reposition the stylus before you start drawing and stop drawing before you lift your finger, and it greatly reduces the finger-occlusion problem. It also conflicts less with the now-standard one-finger-drag scrolling gesture.

Rounding

For Dercuano, as mentioned in Dercuano drawings (p. 64), I want to round off coordinates to reduce the amount of space they take up in the rendered output, although more bloat can be accepted in the source code. Symbol definitions that have been drawn many times offer a way to do this: the Platonic location of a point whose drawn location is highly variable the various times I drew the symbol can be safely rounded to any convenient precision that doesn't take it too far outside the zone where it's being drawn. Moreover, maybe we should weight that point less heavily when we're matching templates. The user should have the option as to whether to draw that point with per-symbol-instance noise or not.

Other points whose position can be interpolated to reasonable precision (either with a spline or with a line) also do not need to be stored for graphical display.

The problem of coming up with a minimal-length description of a polyline that stays within the usual limits of drawn instances of the symbol is an optimization problem that can be solved using the usual kinds of search approaches for offline optimization problems.

Repetition

If you draw the same symbol several times in a row, perhaps in a linear or circular path with systematic variation in position, angle, or size, it's possible that you would like to continue drawing more of it; since the system is categorizing each stroke as a symbol and redrawing it, a reasonable thing for it to do in such a case is to offer further repetitions, perhaps in a different color, with a slider to accept one or more of those repetitions.

Interpolation

One of the attractive features of resampling all the strokes to a uniform number of points is that it makes it easy to interpolate between them, for example, linearly. In drawing editing, this could be used for several different things:

• By defining a subspace from two or more templates, you can draw a stroke that indicates simultaneously a location in that N-dimensional subspace as well as a location, rotation, scale, skew, and stretch in the

display space. In this case, the final operation from the $1 recognizer of calculating the sum of Euclidean distances to measure the distance from a stroke to a template is replaced by projecting your stroke onto that subspace, then measuring its distance from that projection.

• By using two or three templates to define a subspace of one or two dimensions, you can map an area of the display to a subspace of the many-dimensional stroke space. By dragging around this subspace, you can explore variations within that space to instantiate in your drawing.

• By drawing a path through such a subspace, you can define an *animation*. This may work better with a K-nearest-neighbors kind of interpolation so that you can place more than three templates into a two-dimensional space. This path might be drawn synchronously as a draft animation plays — unlike the stroke used for stroke recognition, its timing *is* important.

• Of course you can also do animations morphing strokes with standard tweening functions such as ease-in/ease-out and linear interpolation.

• When doing repetition, as described in the previous section, if the repetitive strokes map onto one of the one-dimensional subspaces found between existing templates, they could indicate a gradual transition; for example, a series of curved lines gradually becoming more straight could indicate a morphing progression that could be continued to straightness and possibly beyond.

Improving the $1 recognizer

Alternative template matching algorithms

The $1 recognizer mostly tries to normalize rather than using search, but even so, the researchers apparently found it necessary to use search for angular alignment to get competitive results. In some cases, they seem to have used fairly fragile statistics in order to keep the algorithm accessible to mathematically naïve users: the normalization of X and Y coordinates using the bounding box means that noise in the X coordinate of the single leftmost and rightmost points will be distributed across all the points, and the linear-interpolation resampling scheme guarantees that such noise will occur.

The search used for the optimal rotation is the old-fashioned golden-section search algorithm, which has the advantage of being derivative-free, but has quite slow convergence (slower than binary chop!) and also makes rather strong assumptions about the input.

(An alternative way to match against a template that is insensitive to translation, scaling, and rotation, though not aspect ratio, would be least-squares linear regression in $\mathbb{C}$, the complex-number field. I've never done linear regression in $\mathbb{C}$, but I think it's a straightforward extension of linear regression in $\mathbb{R}$.)

One approach to improving the algorithm would be to use more robust statistics. For example, you could use the standard deviation or quantiles to determine the X and Y scaling factors, and you could use an angle that depends in some way on all the points instead of just the first point to get the initial rotation.

Perhaps a simpler approach, though, is to use a generic optimization algorithm. The function to optimize is already present: it's the

average Euclidean distance from the points of the transformed input stroke to the corresponding points of the template stroke. The objective is to find the transformation that minimizes it; the translation in X and Y, rotation around some arbitrary center, and scaling in X and Y, form five continuously variable parameters out of the six in an arbitrary 2-D affine transformation. (The sixth missing parameter is diagonal shear, and I'm not sure it should be omitted.) With modern automatic differentiation, it should be straightforward to use a generic optimization algorithm like Adam or a quasi-Newton method to search this parameter space for the best fit. This would probably result in a much simpler algorithm, and possibly a faster one as well.

This also allows extreme aspect ratios and rotations to be penalized in a smoother fashion than the original $1-recognizer algorithm.

Using a better resampling algorithm would probably help somewhat as well.

Indexing

Another issue with the algorithm is that it doesn't really permit any kind of indexing of the templates; if you are matching each new unistroke against a database of 10,000 64-point templates, it is going to take 640,000 Euclidean-distance computations.

The above hacks don't help much with indexing, but perhaps absurdly-downsampled versions of the gesture and templates could be used to get a linear speedup on to searching a large index of candidate glyphs — like 4 or 8 points. Some kind of interval-arithmetic categorization or something is needed if you're going to get a superlinear speedup — some way to put a lower bound on the best distance to any template in a given group, so that you can avoid iterating over the templates in the group.

Alternatively, you might be able to use linear-algebra techniques to speed up the search; if a smallish number of low-dimensional subspaces nearly contain most of the templates (as determined by PCA on subsets of the templates), you can project a user stroke onto each of those subspaces to find out which are plausible candidates (and which are too far away), then perhaps use a k-d tree on the remaining principal components.

Topics

You're pretty much fucked if you want to build an oscilloscope on the AVR's ADC

Kragen Javier Sitaker, 2013-05-17 (3 minutes)

An oscilloscope is a crucial tool for serious circuit hacking. The minimum serious oscilloscope input bandwidth is 20MHz, which is to say, it attenuates 20MHz signal components by 3dB (and probably phase-shifts them), 40MHz signal components by 9dB (and phase-shifts them more), 80MHz signal components by 15dB, and so on; so on a "20MHz" oscilloscope you should still be able to *see* if there's an 100MHz signal, even if you can't measure it very well or see what its shape is.

It would be nice if it were possible to improvise a 20MHz oscilloscope out of inexpensive electronics.

Suppose, though, you're trying to work with an AVR ATMega328. This chip can only run at a max of 20MHz clock, and its ADC can only manage some 77000 conversions per second, and those at 7.5 bits of resolution. It would seem that you're pretty much fucked for seeing anything above 35kHz, which is some 500 times slower than you would like, and 500 times slower than the processor's own clock. That is, you're not going to diagnose much of anything related to an AVR with another AVR, right?

Well, consider this. It's common, although far from universal, for the 1MHz or 3MHz or 10MHz signal you're interested in to repeat many times. So if you can figure out its period, you can sample different points from many waves, then reassemble them into a picture of what one period of the wave looks like. Each sample-and-hold cycle should take place in well under a microsecond, and a faster external sample-and-hold circuit can be built, perhaps with an op-amp on its input to raise the input impedance.

(You probably can't show more than about 1000 samples horizontally on the screen at a time anyway, and you can't do that more than 60 times a second.)

Suppose, though, that it doesn't repeat many times? There's not a whole lot you can do about that if you don't have some way to store the analogue signal until you have a chance to digitize it. All the options I know of (Williams tubes, magnetostrictive delay lines, acoustic delay lines) seem improbable.

As for guessing the period, you can bandlimit your input signal with an analog input filter, sample the waveform at more or less random intervals, and then seek the period that provides the best continuity.

That won't give you a 20MHz oscilloscope but it should give you a 200kHz oscilloscope. One order of magnitude down, two more to go.

Topics

The Suburbean: a minimally-mobile dwelling machine with months of autonomy

Kragen Javier Sitaker, 2019-11-24 (updated 2019-12-03) (32 minutes)

Suppose we take seriously Le Corbusier's idea that a house is a "dwelling machine"; what would it look like? I propose the Subur-bean or Suburbean, a sort of immobile or minimally mobile alternative to a submarine, designed to give the humans maximal control over their living situations. Previous notes on this subject include Thermodynamic systems in housing (p. 2804).

I have been careful in this note not to rely on any novel technology that the humans do not have working already; it's all just a straightforward application of known, working processes to the problem.

My Vanagon experience

The moment in my life when my house was the most convenient was the three months in 2006 when I lived in a Volkswagen Vanagon with Beatrice, which we dubbed the Magic Bus. This is sort of counterintuitive: you would think that living in such a tiny space would be very inconvenient, and indeed accessing some kinds of things was somewhat inconvenient, requiring disassembly of the bed for example; but the day-to-day tasks in it were quite convenient. At first we had a terrible time finding things, but after a few days we got into the habit of always putting things away as soon as possible, and it became *easier* to find things than in a larger house, which allows you to get into bad habits.

I say it was a tiny space because it was a little under two meters wide, about four meters long, and about a meter and a half tall; I couldn't stand up inside unless we cranked up the pop-top, which raised the roof to a height of over two meters in the center portion of the vehicle (say, 1.5 meters by 1 meter), also providing access to another bunk up there that was two meters by 1.5 meters long, about 0.8 meters tall. (This two-meter length included the 1-meter hatch from below, which you would close with the second half of the bed if you were going to sleep up there, which we never did.)

Many camper Vanagons had the front "bucket" seats mounted to swivel around toward the back, so that when it was parked, you could turn them to face the center of the interior. Ours was an aftermarket conversion that didn't have that, so the front driving area was nearly partitioned off from the main interior area, making it even smaller than it would be otherwise. So the only seating in the main living area was the vinyl-padded bench seat that the bed folded down into, which could seat about three people.

As a vehicle, the Vanagon was a terrible piece of shit, but as a house it was wonderful. Unfortunately we drove it a lot in kind of an abusive way and thus had to spend a lot of time fixing it.

It had a vinyl awning or canopy that could be rolled out from one side of it to provide a shaded area to set our chairs in.

One thing it couldn't provide internally was sewage facilities; for this we relied on external bathrooms or, occasionally, Gatorade bottles. And of course the external scenery was usually gorgeous and changed often.

So the inner sanctum of the Magic Bus was about 12 cubic meters, much of which was storage cabinets (and the refrigerator, sink, stove, and batteries), and there were another couple of cubic meters of machinery mounted below, including the hundred-kilowatt air-cooled boxer engine, wimpier than many modern motorcycle engines; and the additional bunk was about 2.4 cubic meters more. The canopy occasionally enclosed another 10 cubic meters or so, and another 12 cubic meters or so were needed for access to the back lift gate.

And this was very comfortable and convenient for two people nearly 24 hours a day for several months, with occasional breaks. Without, it should be emphasized, any drugs other than the occasional ibuprofen or beer for one of us, or any particular level of spiritual enlightenment, since of course drugs or enlightenment can make you comfortable under almost any circumstances.

There was something very empowering about having everything you need within arm's reach, and it also had very nice acoustics and a seriously nice sound system.

Cars are comfortable

Back when I had cars, I often enjoyed sitting in them listening to music. Bucket seats in cars are more comfortable than many chairs, though not all, and acoustics inside of cars are generally much better than in houses, offices, or classrooms, largely due to the necessity of damping engine and road noise; although false-ceiling acoustic tiles help significantly, it's very expensive to buy and install the tens or hundreds of square meters of acoustic foam needed to give really good acoustic characteristics to a house, office, or classroom. If the car engine is running and you have an air conditioner, you have tens of kilowatts of power available for heating or cooling the interior, so within a minute or two you can get the temperature to a comfortable level, whatever your preference is.

Small spaces are more controllable, but large spaces are more efficient

When larger spaces go out of control and become unlivable, it happens more slowly than in smaller spaces; and bringing them back under control takes longer.

CO_2

For example, consider CO_2 homeostasis. Mina's bedroom is 36 cubic meters, three times the size of the Vanagon, and in Reducing nighttime bedroom CO_2 levels (p. 478) I calculated that if Mina and I are in there for 12 hours without any airflow, we could raise the CO_2 levels from their default 400 ppm to 14000 ppm; it would take us 20 minutes to double the CO_2 level. Her whole apartment is 14 m × 3 m × 4 m, or 168 m³, so if it were sealed, it would take us 90

minutes to double the CO_2 levels. By contrast, in the Vanagon, it would take only 6 minutes. So if you manage to seal the Vanagon up too tightly, you'll notice within minutes that it's getting uncomfortably stuffy, and you'll be able to tell within a few minutes if improved ventilation measures are successful.

Climate control

Similarly, if the air conditioner in Mina's apartment breaks, it may take a few hours for temperatures to reach the level where you realize this has happened; and if the house has gotten too hot while you were out, it can take half an hour or so for the air conditioner to bring the temperature down to a livable level.

Per-person efficiency

However, larger spaces generally require more energy and more material to remain under control, although as the scale increases, many of the things you might want to control become more efficient to control.

Acoustic conditioning

There is an example above of acoustic tiles: a single-person listening booth of 0.8 m × 0.8 m × 1.5 m has a total surface area of 6.1 m², which works out to 6.1 m² of expensive acoustic materials per person. By contrast, a 5 m × 5 m × 3 m listening room has a total surface area of 110 m² and can hold about 35 people, so it needs only about π m² of expensive acoustic materals per person.

Climate control per person

A more extreme example has to do with climate control.

A minimal private bunk, like the upper level of the Magic Bus, might be about 2 m × 1 m × 0.8 m, with a surface area of 8.8 m². (This is a bit cramped; this body is 0.95 m high, seated; but we can imagine that there's a bit of ceiling height variation to accommodate such things.) If you're trying to maintain an insulated tent of this size at 22° at a time that the outside temperature is 0°, and your insulation is 100 mm of 0.04 W/m/K insulation (typical for insulation materials; see Air conditioning (p. 1665)), you have roughly 10 m² of effective surface area, so you need 88 W and 1 m³ of insulation, both per person.

If, by contrast, you have a sort of barracks with two stories, each 2.4 m tall, containing a 5-m-long corridor whose walls are covered with three levels of 0.9 m × 0.7 m doors leading into such capsules --- sort of like a morgue or mausoleum, but for the living, like a Japanese capsule hotel --- the overall building is 5 m × 5 m × 5 m, not counting the 100 mm of insulation around the outside. Its surface area is about 160 m², so under the same conditions it loses about 1400 W and needs 16 m³ of insulation. But it holds 48 people (assuming the last meter of corridor is used for travel between the levels and whatnot) so this 29 W and 0.33 m³ of insulation per person, one third of the individual tent. In fact, if occupancy is over about 30%, it won't need active heating; it will need cooling.

There is no limit to this kind of increase in efficiency from scale, assuming the whole interior can legitimately be at the same temperature.

Lighting

Another example has to do with lighting. As described in Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder (p. 527), a few years ago you could supply artificial daylight (10 kilolux) to an individual human in a sort of tanning booth or SAD-treatment pod at a cost of about 200 watts and US$150 of LEDs. But obviously illuminating a normal-sized living room with 200 watts will not come close to daylight. (More information about lighting costs is in Illumination cost (p. 1242), including an option that uses many times more energy but uses only US$3.20 of quartz-halogen bulbs; notes on using lightpipes instead are in Can artificially-lit vertical farming compete with greenhouses? (p. 2064), Subterranean glazing (p. 1126), and Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566).)

Indoor pot growers often use portable closets illuminated in such a way to daylight levels, sometimes with low-pressure sodium bulbs; modern high-efficiency LEDs are also apparently considerably better than the ones I mentioned above. (See Can artificially-lit vertical farming compete with greenhouses? (p. 2064).)

The outline of the Suburbean

The Suburbean is a self-contained living space capable of months-long autonomous habitability for three people under a wide variety of external conditions, much like a nuclear submarine, but with the objective of bringing habitability to urban rather than marine environments. It is built into a standard-width refrigerated "high-cube" TEU shipping container, 6.1 m × 2.44 m × 2.90 m, so that it can be easily loaded onto a truck, train, or ship for moving, although it's better not to do this while people are inside of it. The refrigeration system is used to maintain the interior at a comfortable temperature chosen by the inhabitants, rather than a temperature suitable for preserving food. The Suburbean has very limited autonomous mobility --- it's more of an autoimmobile than an automobile.

Unlike a nuclear submarine, the Suburbean does need access to external air, at least to exhaust its waste products into and usually also to burn its diesel fuel with. When no external power is available, its refrigeration unit and other internal energy consumption is powered by two 1-kW onboard diesel engines, providing some 2 kW of mechanical or electrical power.

The whole Suburbean weighs some 20 tonnes, well short of the 30-tonne limit for a loaded TEU. This means that when lifting itself vertically with its six built-in winches with its 2-kW-output generator engine running at 100% duty cycle, it can rise only about 10 mm/s, although when running off batteries or external power, it can manage about 130 mm/s, limited only by the 25 kW of the winches. In horizontal or downward movement, when it need only overcome friction, whether suspended in the air by its winch cables or running on wheels, it may be able to move several meters per second over limited distances under its own power. This is what I mean by "very limited autonomous mobility".

How realistic is this?

Amazon lists a 6 horsepower (=4500 W) winch for US$320. eBay

lists a bunch of 1kW handheld portable generators for around US$200 to US$300, but they all run on gasoline; diesel generators seem to start around US$1200 and 7kW. A plain TEU-sized shipping container seems to cost around US$2000, but refrigerated ones seem to cost around US$6000, or up to twice that new.

Thermal homeostasis

Like some refrigerated shipping containers, the interior of the Suburbean is superinsulated with vacuum insulated panels (< 8 mW/m/K) to reduce the power necessary for refrigeration; it uses 50 mm of them in two staggered layers, reducing the interior dimensions by 100 mm in addition to the 200 mm or so consumed by corrugation, to about 5.7 m × 2.04 m × 2.5 m (32.3 m³, of which 20 m³ is air). Over the effective surface area of 70 m² or so, this works out to about 11.2 W/K, so over the 0° to 44° temperature range, only a maximum of 250 W of forced heat flow is needed at steady state, or about 85 W electric to power the heat pump with its CoP of 3, which works out to about 210 W of diesel fuel. If three people are actually present inside, this adds about 300 W of forced heat flow when the outside temperature is too hot rather than too cold, increasing these numbers to 550 W of heat flow, 180 W electric, and 460 W fuel.

The usual wooden flooring of a shipping container is not present, since the vacuum insulation layer includes the floor as well, requiring a rigid protective support surface to prevent damage to it.

The air conditioner's CoP of 3 is achieved by liquid-cooling its condenser coils with antifreeze that is circulated through pipes that heat the metal outer walls of its entire 35-m² lateral area, providing a large surface for radiative and convective cooling. When the air conditioner is operated in heating mode as a heat pump from outside to inside, this operates as liquid-heating the evaporator coils with the same antifreeze circulated through the same pipes, where they absorb heat from the environment.

When the Suburbean is being heated rather than cooled, the diesel generators' exhaust is routed through a heat exchanger to transfer the heat they produce to the interior as well through a closed-circuit heat-transfer fluid, the same way that car heating systems typically work. This reduces the energy demand of the heat pump, possibly to zero.

What existing reefers are like

The Container Handbuch says that typical power consumption for cooling a ThermoKing Smart Reefer TEU is around 3.6 kW, which is about 43 times what I've calculated above; EPRI's "Electric Refrigerated Container Racks: Technical Analysis" says that diesel reefers have 2-liter engines with 30 to 40 horsepower, or in modern units, 20 to 30 kW. Container Handbuch section 3.1.1.2, "Container design and types, Part 2", suggests perhaps unintentionally that typical mechanically powered refrigerated containers are *not* insulated, and that insulated containers typically use 50–100 mm of polyurethane foam, reducing heat conductance to 0.4 W/m²/K (type code H5) or 0.7 W/m²/K (type code H6). Instead, the 50-mm thickness of VIPs described above would have heat conductance of some 0.16 W/m²/K. This explains a discrepancy of about 3.4, leaving another factor of 13 or so from my calculation above, to be explained by people using uninsulated containers (!?) and/or the much lower temperatures.

It seems that self-powered refrigeration units are (or recently were) dominant in land transport, but plug-in electric refrigeration units (three-phase 400VAC) instead are dominant for sea transport. Carrier and ThermoKing make all the refrigerated container machinery for sea transport.

I found a "Technical Specification for Refrigerated Container Model No. SS1WN1" of half a TEU from Shanghai Reeferco. It claims they use a Carrier refrigeration unit and their insulation is 63–80 mm in thickness everywhere except on the floor where it ranges up to 135, and that the heat transfer rate U_{max} is 20 W/K at 20°C. Its surface area is about 43 m² so that works out to 0.47 W/m²/K, which is in the range of the Container Handbuch numbers above. Yet the Carrier 69NT40-541-300 it ships with can do 3.2 to 10 kilowatts of cooling with inside-outside temperature differences of 67° (3.2 kW) to 36° (10 kW). You would think that with a temperature difference of only 36° you would only need 720 W, not 10 kW. Maybe it's shipped with this huge air conditioner for the initial pulldown?

Energy storage and solar panels

The Suburbean carries half a tonne of diesel fuel, which weighs 0.832 kg/ℓ and provides 43.1 MJ/kg, so this occupies 0.6 m³ of the 32.3 m³ available and stores 21.6 GJ. At the outdoor high-temperature extreme of 44°, when it needs to use 460 W of diesel fuel to maintain thermal homeostasis, this provides almost 18 months of autonomous operation before needing to refuel, as long as it has access to oxygen from air and somewhere to release exhaust.

The Suburbean also includes half a tonne of lithium-ion batteries, which at some 500 kJ/kg amounts to 500 MJ, about 2% of the energy content of its diesel fuel tank. Still, at the 180 W electric needed to maintain thermal homeostasis at extreme temperatures, this amounts to a bit over a month of autonomy without access to air; for example, during a flood or while buried under rubble, although assumptions about external temperature and insulation may be called into question in such circumstances. Lithium-ion batteries have a self-discharge rate on the order of 2% per month, which is insignificant in this context.

The batteries are rated for a discharge rate of 5C (12 minutes), which is lower than the 15C discharge rate (4 minutes) used for quadcopters and the like, but higher than the lowest-end batteries. At a conservative discharge rate of 3C (20 minutes), the Suburbean can muster a peak output power of some 400 kW, a bit over 500 horsepower; for example, for arc welding.

To recharge the batteries without consuming diesel fuel when sunlight is available, 6 m² of the 15-m² roof of the Suburbean contains 22%-efficient SunPower Maxeon Gen II monocrystalline solar panels under an openable protective cover. This provides nominally 1300 W peak of solar power, providing 260 W as a 24/7 average, at a typical 20% capacity factor. Moreover, the back of the cover is mirrored, and it is positionable under motor control, permitting higher capacity factors than are possible with statically positioned panels, and potentially even providing over 1300 W at times.

Atmospheric life support

See also House scrubber (p. 248) and Notes on a possible household air filter (p. 1961).

Of course, when air is available from the outside, the Suburbean will use it, after appropriate filtration and temperature control. But above, it is explained that the batteries can provide a month's worth of climate control without air for the diesel engine. So, what about air for the humans inside in such a situation?

Oxygen

Remaining human-habitable without access to air requires very roughly 600 g of oxygen per person per day, which is the amount in about 2 m³ of air (2.4 kg) or 0.4 m³ of oxygen, at 1.429 g/ℓ. You'd think you could generate oxygen on demand in such situations through electrolysis of water, and although that does work, it turns out to be very expensive, about 250 MJ/kg O_2 (see Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345)), which at 600 g/day/human works out to about 1700 W per person, a large energy drain which also adds to the cooling load when the situation is hot. The 20 m³ of air contains enough oxygen to last only 10 person-days, or 3 days with 3 people, and storing compressed air at ordinary pressures such as 15 atmospheres (1.5 MPa) would only give you 15 normal m³ of air per m³ of compressed-air storage.

Instead, the Suburbean stores oxygen as relatively nontoxic sodium chlorate, $NaClO_3$. Aircraft decompression masks are supplied from sodium-chlorate chemical oxygen generators; a 63-mm-diameter, 223-mm-long canister (0.0007 m³) generates enough oxygen for two humans for about 15 minutes, which suggests you'd need 1 m³ per person per month, which is closer but still too bulky. A simpler chemistry is used in chlorate candles, which are mostly sodium chlorate (2.5 g/cc) but with iron powder to produce heat, and provide 6.5 person-hours of oxygen per kg, which works out to 110 kg per person-month or 330 kg per 3 person-months.

However, the Suburbean's sodium chlorate is not mixed with a fuel in this way, so there is no risk of an oxygen-candle explosion. Instead, heating the sodium chlorate to the requisite 300° is done with an electrical heating element. This produces NaCl and 48 daltons of oxygen per 106.4 daltons of $NaClO_3$, so the 55 kg of oxygen required for three person-months of autonomy require only some 122 kg of $NaClO_3$, occuping 49 ℓ.

(I think it should be possible to run the diesel engines from stored sodium chlorate and diesel fuel, but the amount of sodium chlorate required is rather large, and the reduction to NaCl consumes some of the energy; 106.4 daltons of $NaClO_3$ yield 48 daltons of oxygen, which can oxidize only 14 daltons of diesel, so you need 7.6 kg of sodium chlorate for every kg of diesel, so you only get 5 MJ/kg from the total mixture (minus whatever the endothermic chlorate decomposition energy is), and only 1.5 to 2 MJ/kg of exergy. Still, that's three or four times the exergy density of the batteries, and it might provide a viable approach to multi-month underwater autonomy. See Underwater energy autonomy (p. 1662) for more.)

The Suburbean generates its own sodium chlorate, when water and energy are abundant, from electrolysis of an aqueous solution of NaCl, with some HCl to lower the pH, at 90°. Because this reaction also produces some hydrogen, the reaction chamber is outside the

interior space, so that if there is a hydrogen leak it dissipates into the environment rather than making the internal atmosphere explosive. Normally the hydrogen is fed into the diesel engines through a secondary injector to burn it, or burned in external air with a spark igniter if the diesel engines remain off for too long. If there is no external air, the hydrogen is just released into the environment.

CO_2

It isn't enough just to add oxygen to the atmosphere, though; CO_2 must also be removed. I reviewed the possibilities in House scrubber (p. 248); the Suburbean's CO_2 scrubber uses six small beds of high-surface-area caustic magnesium oxide to absorb the CO_2, which it regenerates by heating them to 450° by recirculating a stream of CO_2 through the bed and through a heating element. One person-hour of CO_2 is 25 grams, if the figures in Reducing nighttime bedroom CO_2 levels (p. 478) are correct; MgO weighs 40.3 daltons; $MgCO_3$ weighs 84.3 daltons; CO_2 weighs 44 daltons, which is the difference. So each bed is sized to be able to hold 75 g of CO_2 during its active hour, which makes the conservatively 150 g of $MgCO_3$ (2.96 g/cc, so 50 cc) which becomes 71.7 g of MgO after heating. Excess CO_2 is vented to the environment.

(Hmm, I'm not sure this is actually the right solution; I can't find any notes about anyone using MgO as a CO_2 scrubber, and Wikipedia's CO_2 scrubber article says the Space Shuttle's metal-oxide-adsorbent scrubber was regenerated with 10 hours of 200° air through the "Metox Canister", which it turns out actually used *silver* oxide, perhaps in a thin layer on the surface of some kind of inert support material that comes with a high surface area. More recent design proposals use solid amine adsorbents and Fe_2O_3 but I don't know if they've flown or if they're suitable for the Suburbean.)

As a backup in case all of this fails, the Suburbean is also equipped with lithium-hydroxide curtains which provide a few days of autonomy before needing to get access to air.

Other contaminants

As mentioned in House scrubber (p. 248), there are other contaminants that must also be removed, and a variety of means to remove them.

Subdivision into compartments

The reason there are six separate CO_2 scrubbers is that the Suburbean is divided into two compartments with a hermetically-sealed bulkhead between them, connected through an airlock; experience with the ISS has shown the importance of being able to isolate the effects of accidents such as fires and refrigerant leaks to only part of the environment.

If the air quality in one compartment is becoming bad because of a broken or inadequate pollution control system rather than because of a polluting accident, the inhabitants can open the airlock to allow air to flow.

Doors and connectivity

Of course, the Suburbean has the usual big doors that are the entire rear end of the shipping container. But it has some other features as well.

The Suburbean is stackable; each of its two independent compartments has a hermetically-sealed hatch in one corner of the top surface and the bottom surface, making it possible to connecting multiple Suburbeans into a single larger dwelling-machine by stacking them on top of one another and twistlocking them together, then opening the hatches and sealing them together. ISO 1496(1) requires that TEUs be tested for stacking nine containers high, but that's only safe when the doors are closed! The Suburbean has extra bracing just inside those rear doors, part of which is the hatch shaft itself, so that it can safely support that amount of weight even with the doors open.

This allows you to construct an ad-hoc autonomous minimally-mobile hermetically-sealed apartment building with space for up to 26 people to live comfortably, assuming you use the rear compartment of the bottom Suburbean as a common entry and exit.

When no other Suburbean is atop yours, these hatches can also serve as entry and exit, if you either have a way to get on and off the roof, or space underneath. Unlike the rear doors, the hatches cannot be locked by people outside, only by the Suburbean's computer systems, ensuring that escape is possible in an emergency (even if the rear compartment is on fire) and preventing outsiders from locking the doors as a way to apply coercion to the inhabitants.

The shaft connecting the top and bottom hatches is also hermetically sealed from the compartment it runs through, with a door providing access to the compartment. A positive-displacement ventilation pump runs only when air-quality sensors report that the air in both the shaft and the compartment are not contaminated. This means that air-contamination incidents in compartments do not discourage you from traveling vertically past those compartments in a stack of three or more Suburbeans. It also makes it possible for the inhabitants of a compartment to lock it against access from their neighbors, while still allowing those neighbors to pass through.

If there are stable high points to attach the winches to, no separate crane is needed to stack the Suburbeans; once some winches are attached, the upper Suburbeans can move themselves into position. The winch cables can also guy the stack of Suburbeans to points on the ground or elsewhere to reduce the risk of falling in wind. (See Bootstrapping rope bridges and other tensile structures with UHMWPE-bearing drones (p. 2950) for one way to bootstrap the winch rope attachment.)

If your Suburbean is hanging from its winches out of reach above you, you can command it to descend to within reach and open a bottom hatch so that you can enter; then, once you have entered, you may want to command it to ascend again.

Both of the hatch shafts are on the right side of the Suburbean if you're facing its nose; this means that if you're facing the rear door, the rear hatch shaft is on your left. This means that if one Suburbean is yawed 90° on top of another, one twistlock point and adjacent hatch shaft can dock, but it's the *rear* hatch of one connecting to the *front* hatch of the other. With some additional support beyond that provided by the twistlock points, this design permits the assembly of much larger and more stable ad-hoc hermetically-sealed interconnected Suburbean Voltrons.

Flotation

Because the Suburbean is hermetically sealed and has a volume of 43 cubic meters, but a gross weight of only 20 tonnes, it floats if it falls into water as long as it doesn't leak, like other shipping containers.

The sensation of space: it's done with lights, mirrors, fans, and acoustic foam

Humans who feel that they're in a small, cramped space for a long period of time will be unhappy. So within the Suburbean there are areas with "infinity mirror" illusion ceilings and parallel mirrors on the walls to create the illusion of a large horizontal space. Extra acoustic foam on the other walls and behind holes in the mirrors provides the acoustic sensation of being outdoors; fans silenced by baffles followed by laminar ductwork provide a breeze.

Garbage and sewage

These are potentially a big issue, as they are in space travel and nuclear submarines.

Food

Water

Pets

Topics

Stereographic map app

Kragen Javier Sitaker, 2018-12-02 (2 minutes)

Maybe a stereographic projection, or Ptolemaic planisphere, would be a fun way to display a local map on your cellphone. You'd still have different possible zoom levels, but all of them would show the whole world on your screen, except for an arbitrarily small area just around the point of projection. You could move the point of projection as you moved around.

Planisphere projection has many interesting properties. It's conformal, which is to say that it's locally planar — it doesn't distort the shapes of small objects — but it's not isometric or equal-area, which is exactly what would seem to offer the possibility of making a potentially useful local map that includes the entire planet.

There's the question of how to orient things. You can try to orient the map with north at the top, or you can try to orient it according to the cellphone's compass, but either way you have a contradiction. Suppose you orient it with north at the top. In the center of the screen you have the antipodes, and around it the rest of the world, and you are positioned at the point at infinity. Suppose you're in a cul-de-sac with a street leading out to the north. Does this street appear at the top of the screen or the bottom?

If it appears at the top of the screen, the more northerly points on the street are further down the street. The least northerly point, the end of the street where you are, is at infinity. More and more northerly points are further down the screen toward the antipodes, until finally the street ends. So in fact north is down.

Consider, though, a parallel street far enough to the west that its end appears on your map, which also ends in a cul-de-sac just to the west of you, and proceeds north from there. On *this* street, more northerly points are further up the screen — until they start to curve in towards the center of the screen, and then start to curve back down, and north is down again.

Topics

Interval filters

Kragen Javier Sitaker, 2015-09-17 (2 minutes)

What if we consider the particle-filtering problem from the perspective of interval arithmetic? Instead of approximating a multidimensional prior probability distribution using an exponential number of particles, we could approximate it with an exponential number of irregular partitions of the multidimensional space, and then subdivide the high-probability partitions and fuse the low-probability ones in order to give each partition equal probability, then do a Bayesian update. It seems like this could work and might be more broadly applicable (and rigorously justifiable) than the Monte-Carlo approach.

Like raw particle filters, this won't work well on high-dimensionality spaces, but perhaps it can, like them, be extended to work on such spaces.

Interval arithmetic can be used to efficiently calculate that the posterior probability of being inside a large region of the space is negligible.

Often, divide-and-conquer algorithms are more efficient when the division is by three instead of by two — dual-partition Quicksort being one notable example — and I suspect that might be the case here too. If combined with random rather than optimal positioning of the partition, in particular, it would help with the case where we really do kind of need particles — where there is a tiny local maximum probability lost in a large, low-probability partition. Eventually, random choice of three-way partitioning will happen to snag the tiny local maximum between its pincers, and have a chance to track it down further.

You could store for each partition not only the total probability (or, equivalently, average probability density) of the partition, but also the average slope or derivative of the probability density inside of it. This would dramatically reduce the error in the estimate of the PDF that all the intervals comprise together.

Topics

An algebra of textures for interactive composition

Kragen Javier Sitaker, 2019-05-08 (4 minutes)

I was thinking about the problem of composing graphical textures in a user interface, and it occurred to me that a neat formulation was possible using just functions of the form $\mathbb{R}^2 \to \mathbb{R}^2$, which I will call "textures". The whole system functions by composing textures out of more primitive textures.

(Perhaps I should recast this as functions of the form $\mathbb{C} \to \mathbb{C}$? That would conveniently provide rotation and a wide variety of conformal mappings, and it would substantially reduce the number of functions needed, though it would require the redefinition of ×, ÷, and ** (and compensating additions of their elementwise variants), and probably I should be renamed Z.)

Visualization on the screen

The idea is that the inputs to a texture are coordinates representing the position on the screen, and the outputs represent in some sense a color. To fully specify an RGBA color, you need two textures (I suggest using HSVA with one texture for HS and the other for VA, or YUVA with a YA/UV split), but you can quite reasonably visualize individual textures by using a constant for the other texture.

Atomic textures

The most basic textures you start with are numeric constants, I, and T. Constants such as 0.3 are interpreted as having that value everywhere for both components: λ(x, y) → (0.3, 0.3). I, or meshgrid, is the identity function: λ(x, y) → (x, y). T, or transpose, is almost the same, but returns the coordinates in reverse order: λ(x, y) → (y, x).

The standard functions abs, ln, exp, sin, cos, tan, asin, acos, and atan are also available as atomic textures, and they also apply elementwise; e.g., sin(x, y) = (sin x, sin y).

You could imagine loading in (channels of) image files as additional atomic textures.

Binary or combining operations

Two special shunting operations are provided: composition, written backwards with tightest precedence with ".", and joining, written with loosest precedence with a comma, ",". Composition A.B is just λ(x, y) → B(A(x, y)), while joining combines values from the different functions; if A(x, y) = (ax, ay) and B(x, y) = (bx, by), then (A, B)(x, y) = (ax, by). The opposite combination can be achieved by composition with the aforementioned T texture, as (A.T, B.T) = (ay, bx). (For convenience, we could define textures X as (I, I.T) and Y as (I.T, I), so you can write A.X to get just the X output of A on both channels.)

A fairly standard list of binary arithmetic operations apply pointwise: +, -, ×, ÷, //, %, &, |, &^, ^, **. That is, given that a(x, y) = (xa, ya) and b(x, y) = (xb, yb), (a ** b)(x, y) = (xa**xb,

ya**yb), (a & b)(x, y) = (xa & xb, ya & yb), etc. In this way you can compute, for example, 3 + 4 * 5 * I % 1. % is the modulo operation, // is integer division as in Python, ** is exponentiation, ^ is XOR as in C, and &^ is the and-not, set-subtraction, or abjunction operation, as in Golang. Binary operations are applied to fractions as binary fractions.

Comparison operations are also provided: ==, >=, <=, >, <, != all apply pointwise to textures and produce answers of 0 or 1, representing, respectively, true or false, as in C. So, for example, (3, 4) < (3, 5) produces (0, 1).

Finally, four-dimensional simplex noise is provided with the # binary operator.

Interactive user interface

For playing with these textures, it occurred to me that you could probably usefully compile them into GLSL shaders for real-time display, and use an RPN-calculator approach like rpn-edit or autodiffgraph to provide instant feedback on each new atomic texture as it's created; the multitouch approach described in Interactive calculator (p. 2771) might be more usable on hand computers.

Additional atomic textures t, the current time, and M, the mouse coordinates, would facilitate simple interactive animations. Multitouch puppetry could be handled in a variety of ways, the simplest of which is something like variables t0, t1, t2, t3, t4 for the first five touches.

Topics

Some notes on the landscape of linear optimization software and applications

Kragen Javier Sitaker, 2019-08-21 (updated 2019-08-25) (35 minutes)

I started auditing an "operations research" class at the UBA this quatrimester. It's largely focused on linear programming, which is to say, optimizing linear functions subject to linear constraints, but the class also covers some other non-linear-programming topics.

I'm particularly interested in optimization algorithms, as I've said a few times before, because they potentially raise the level of abstraction in programming — rather than specifying how to compute something, you just specify how to recognize if something is the thing you wanted to compute, and then let some kind of generic solver figure out how to solve the problem. That's what earned them such a prominent place in More thoughts on powerful primitives for simplified computer systems architecture (p. 1895). And it turns out that linear-optimization solvers, as they're called, are the most developed family of optimization algorithms out there.

Other useful overviews of the space can be found in the NEOS Server Optimization Guide and the GLPK Wikibook.

I'm paying as little attention as possible to proprietary software here, but unavoidably need to mention it periodically simply because it has played such a central role in this space.

Overview of linear programming

Linear programming is a particularly popular subset of mathematical optimization because there are well-known algorithms for solving it that scale to fairly large numbers of decision variables (the variables the optimization algorithm is allowed to decide on a value for to reach the optimum), and it provides a useful approximation for many economically important problems. It's so popular that sometimes it overshadows the rest of mathematical optimization — many books purportedly about mathematical optimization consist almost entirely of material on linear optimization, with perhaps a short section at the end on nonlinear optimization.

I've tended to be a bit prejudiced against linear optimization because the real world is never perfectly linear and because I felt that linear constraints weren't very expressive. But, now that I'm taking this class, I'm starting to appreciate the astounding expressiveness of linear programming — especially "mixed integer" linear programming ("MIP" or occasionally "MILP"), an extension of pure linear programming in which some of your decision variables are constrained to integer values, which is more expressive but enormously more difficult to solve.

As the lp_solve documentation summarizes it:

The linear programming (LP) problem can be formulated as: Solve $\mathbf{A}\cdot x >= V_1$, with $V_2\cdot x$ maximal. $\mathbf{A}$ is a matrix, x is a vector of (nonnegative) variables, V_1 is a vector called the right hand side, and V_2 is a vector specifying the objective

function.

An integer linear programming (ILP) problem is an LP with the constraint that all the variables are integers. In a mixed integer linear programming (MILP) problem, some of the variables are integer and others are real.

(Some formulations additionally specify that all x_i are nonnegative, which turns out to matter very little.)

There are several algorithms known for solving linear optimization problems, of which the oldest and most-widely-used is George Dantzig's "simplex algorithm", despite its worst-case exponential complexity. Many newer solvers instead use the "interior-point method", which has worst-case polynomial complexity for continuous linear optimization. (Most mature solvers support both.) However, for mixed integer linear programming, they all must fall back on exponential-time search procedures, usually "branch-and-cut" procedures (also known as "branch-and-bound").

The software landscape

Typically, as an end-user, you solve linear optimization problems by writing your problem in an "algebraic modeling language" such as GNU MathProg or "GMPL" (a dialect of a popular proprietary modeling language called "AMPL") or ZIMPL; the model translator compiles your model (including data files, which it might pull from a database) into a standard format (such as "MPS", "LP", or `.nl` — see below), generally expanding it considerably, and submits it to a "solver".

Unfortunately, the most popular model translators and solvers are all proprietary; proprietary CPLEX, now owned by IBM, is the most popular solver, with proprietary Gurobi and proprietary SCIP second and third, and proprietary AMPL is the most popular model translator; there is an excellent AMPL book by Brian Kernighan (et al., but "Kernighan" is a sufficient recommendation) which also serves as an introduction to linear programming. Both CPLEX and SCIP are gratis for institutional academic use (we are using SCIP in the class) but not to independent researchers; I don't know about AMPL. So we can forget about these.

On the NEOS server, anyone can use most of this software in batch mode for free, for optimization jobs of up to 16 megabytes of input, 3 GB of RAM, and 8 hours of run time, thanks to the Wisconsin Institute for Discovery at UW Madison; the NEOS server has AMPL, ZIMPL, CPLEX, SCIP, CLP, CBC, Gurobi, CBC, and SYMPHONY, but not including GLPK or lp_solve, perhaps because it doesn't consider them "state of the art".

XXX I really need a diagram!

Modeler–solver interfaces (MPS, Osi, `.nl`, and LP)

Often the model translator talks to the solver through a file in some standardized file format; MPS is the most popular such format, a punched-card-based format originally designed for IBM's Mathematical Programming System/360. LP, sometimes called "CPLEX LP" because of its origins in CPLEX, is a more-readable but less-widely-supported format; and AMPL, a popular but proprietary algebraic modeling language (compatible with GLPK's MathProg), defined its own `.nl` format, mentioned earlier, that has been interfaced to a wide variety of solvers.

Linear-programming solvers have been around considerably longer

than algebraic modeling languages, so both MPS and LP are designed for humans to write, but MPS is designed badly.

(There are actually two different, incompatible MPS formats: the original fixed-field MPS format from the 1960s, and a slightly less bletcherous "free-form" 1980s version.)

Here's a small LP file that GLPK and CBC are able to accept and solve (the optimum is z = 1, y = 0.33...).

```
Maximize
 x: + 2 y + 3 z
Subject To
 a: + 2 z + 3 y <= 3
Bounds
 0 <= y <= 4
 0 <= z <= 1
End
```

You can save that in `foo.lp` and solve it with GLPK by running `glpsol --lp foo.lp -o foo.out`. You can transate it to MPS with `glpsol --check --lp foo.lp --wmps foo.mps`.

Most solvers, including GLPK's, also have bindings that allow you to call them from your program without preparing an input file for them to parse; and the COIN-OR project has written a standard API called "Osi" for doing this with many different solvers.

GLPK (modeling language (GMPL/MathProg) and solver)

GLPK, a C library, is the thousand-pound gorilla of linear optimization systems; Octave is built with it, and there are bindings to it in Java, Python, and R. The GLPK package includes the model translator for the popular algebraic modeling language GNU MathProg (*aka* GMPL), and also solvers for the revised simplex method, the primal-dual interior point method, and the branch-and-bound method.

MathProg is a subset of the AMPL language mentioned above, including nearly all the facilities of AMPL for defining optimization problems, but without statements like `include`, `model`, and `data`.

It supports MPS and LP format files for both input and output, and can even translate between them, as well as generating them with the MathProg model translator.

GLPK was originally released in 2000, but has been under active development ever since; there is a Wikibook about it, linked above in the introduction.

Here's a MathProg version of the example LP file above, with additional directives to solve it and display the results:

```
var y >= 0, <= 4;
var z >= 0, <= 1;
maximize x: 2*y + 3*z;
subject to a: 2*z <= 3 - 3*y;
solve;
display x, y, z;
end;
```

If you save this as `min.mod` you can run `glpsol -m min.mod` to get the

solution, or `glpsol -m min.mod --wlp min.lp` to get a slightly expanded version of the LP file above. Pitfall: if it's gzipped, like most of the examples that come with GLPK, `glpsol` will uncompress it to run it, but at least the version I have dies with a spurious "read error".

This example is somewhat freer-form than the LP file (there's an inequality with decision variables on both sides, for example), but to really show the advantages of an algebraic modeling language like MathProg, we need a bigger model. For example, here's an integer-linear-programming sudoku solver I wrote this afternoon:

```
param N default 9; set S := 1..N; param SW 'square width' default 3;
set G 'givens' default {};
param givenrow {G}; param givencol {G}; param givenchoice {G};

var X { S, S, S } binary;

maximize pleasure: 69;

rows {col in S, choice in S}: sum {row in S} X[row, col, choice] = 1;
mutex {row in S, col in S}: sum {choice in S} X[row, col, choice] = 1;
cols {row in S, choice in S}: sum {col in S} X[row, col, choice] = 1;
squares {bigrow in 0..SW-1, bigcol in 0..SW-1, choice in S}:
    sum {row in 1..SW, col in 1..SW}
        X[bigrow * SW + row, bigcol * 3 + col, choice] = 1;
givens {j in G}: X[givenrow[j], givencol[j], givenchoice[j]] = 1;
```

`glpsol --check --wlp sudoku.lp -m sudoku.mod` translates this 12-line MathProg model into a CPLEX LP file of 2000+ lines, beginning as follows:

```
\* Problem: sudoku *\

Maximize
 pleasure: 0 X(1,1,1)
\* constant term = 69 *\

Subject To
 rows(1,1): + X(1,1,1) + X(2,1,1) + X(3,1,1) + X(4,1,1) + X(5,1,1)
 + X(6,1,1) + X(7,1,1) + X(8,1,1) + X(9,1,1) = 1
 rows(1,2): + X(1,1,2) + X(2,1,2) + X(3,1,2) + X(4,1,2) + X(5,1,2)
 + X(6,1,2) + X(7,1,2) + X(8,1,2) + X(9,1,2) = 1
 rows(1,3): + X(1,1,3) + X(2,1,3) + X(3,1,3) + X(4,1,3) + X(5,1,3)
 + X(6,1,3) + X(7,1,3) + X(8,1,3) + X(9,1,3) = 1
```

There's a SciTe-based IDE for GLPK called Gusek I haven't tried. I've often had a lot of trouble debugging my MathProg models, and I wonder if it might help.

ZIMPL (modeling language)

ZIMPL, the Zuse Institute Mathematical Programming Language, is a modeling language from the same academic research institute, the Zuse Institute Berlin, that wrote the solver SCIP, but ZIMPL is free software, while SCIP is not. It's mostly compatible with GNU MathProg. It started out as Thorsten Koch's Ph.D. thesis in 2004, ZIB-Report 04-58, "Rapid Mathematical Programming".

("Mathematical programming" is a synonym for "mathematical optimization".)

The ZIMPL language appears to be more or less at the same level of abstraction as MathProg, but with arguably nicer syntax. So for example in ZIMPL you might say

```
minimize cost: 12 * x1
    + sum <a> in A : u[a] * y[a];
subto oneness: sum <a> in A : u[a] == 1;
subto nonnegative: forall <a> in A : y[a] >= 0;
```

while in AMPL/MathProg you would say

```
minimize cost: 12 * x1
    + sum {a in A} u[a] * y[a];
subject to Oneness: sum {a in A} u[a] = 1;
subject to Nonnegative {a in A}: y[a] >= 0;
```

It can produce MPS and LP files, as well as something called "Polynomial IP". Beware, it generates the name for its output file from the name of the input file, and so it will overwrite any existing file of the corresponding name.

Here's a translation of the tiny model used as examples earlier into ZIMPL:

```
var y >= 0;
var z >= 0;
maximize x: 2*y + 3*z;
subto a: 2*z <= 3 - 3*y;
subto ym: y <= 4;
subto zm: z <= 1;
```

If saved as `foo.zpl`, you can convert it to LP format with `zimpl foo.zpl`, writing the output to `foo.lp`.

Since ZIMPL and MathProg are so similar in their capabilities, but MathProg is bundled with the popular GLPK solver, I don't know what the advantage of using ZIMPL would be — except that the proprietary solver SCIP has ZIMPL built in, and so it's more convenient to use SCIP with ZIMPL than with MathProg.

The biggest difference I've found so far between ZIMPL and MathProg, actually, is that MathProg lets you specify things to do after the solution is found, at least when you're using the GLPK solver. This way you can present the solution in a more readable form. My Sudoku solver above, for example, has half a page of code tagged onto the end to display an ASCII-art Sudoku board. To do the same thing with SCIP and ZIMPL, I ended up writing a Python script to match regexps against the SCIP output.

CMPL (modeling language)

CMPL is another algebraic modeling language from the COIN-OR project, GPLv3 licensed, bundled with an IDE called Coliop; it can use the CBC or GLPK solvers, as well as proprietary solvers. It also includes Java and Python interfaces, presumably as a sort of embedded DSL. The project mailing list seems consistently very low traffic since 2011.

I haven't tried it yet, although it looks like it might address some of my annoyances.

MiniZinc (modeling language)

MiniZinc is a constraint modeling language that is not limited to linear constraints; it can use CBC as a solver, as well as MinisatID, proprietary SCIP, and over a dozen other solvers. I haven't tried it yet.

libisl (ILP solver)

The documentation says:

> isl is a library for manipulating sets and relations of integer points bounded by linear constraints. Supported operations on sets include intersection, union, set difference, emptiness check, convex hull, (integer) affine hull, integer projection, and computing the lexicographic minimum using parametric integer programming. It also includes an ILP solver based on generalized basis reduction.

I have it on my laptop because GCC depends on it, apparently for a newish loop optimization framework internal to GCC called Graphite. It sounds scarily powerful. Unlike the others, it's specifically intended for *integer* optimization, with no support for continuous-domain optimization.

Needless to say, it doesn't interoperate with the others.

COIN CLP/CBC/SYMPHONY (solvers)

The COIN project is an Apache project for making operations-research results reproducible by basing them on open-source software. It includes the solvers CLP, CBC, and SYMPHONY; CBC and SYMPHONY require an underlying linear-programming solver like CLP; it also defined an API called "Osi", the Open Solver Interface, as an alternative to the file-format interfaces described earlier. I haven't yet figured out how to use SYMPHONY yet.

These three solvers are the only free solvers I've found on the NEOS server.

I've been able to solve minimization MPS problems with CLP using `clp foo.mps` but solving maximization problems or seeing the actual results requires a slightly more elaborate command:

```
$ clp solar.mps -maximize -dualsimplex -solution solar.solution
```

If you apply this to an MPS file containing an integer-programming problem like the Sudoku solver given above, you get garbage, because CLP finds a continuous solution instead of an integer solution. There is a similar `cbc` executable which has an `-branch` option to avoid this:

```
$ cbc sudoku.mps -dualsimplex -branch -solution sudoku.solution
```

This finds an answer which appears to be correct, using binary variables as Garns intended, in about half a second. By contrast, with GLPK, `glpsol --mps sudoku.mps --output sudoku.sol` finds a solution in 4 seconds. The CBC user's guide explains in detail how CBC supports integer variables but does not explain how to use the `cbc` executable; Prof. Haroldo Gambini Santos wrote a short separate CBC command-line guide in 2011; it recommends the simpler

```
$ cbc sudoku.mps solve solu sudoku.solution
```

which seems to be equivalent. Without the "solve" command (or the lower-level `dualsimplex` and `branch` commands in the version above), `cbc` will still be happy to write `sudoku.solution`, but it may contain no solution, or only a solution of the linear relaxation of the problem.

On a more complex problem, unfortunately, I got CBC to dump core with this command. The other command worked better but I think it may not have actually given the right answer.

A nice feature of CBC is that if you kill it with ^C it responds by displaying information about the process it's made so far on the problem (and then, as you would expect, exiting).

CBC also accepts input in CPLEX LP format.

It seems that CLP, CBC, and COIN-OR in general have stagnated somewhat since John J. Forrest, the primary author of CLP and CBC, has retired from IBM Research around 2005; there's mojibake in the CLP User Guide and the CBC User Guide, for example. CLP and CBC are primarily intended to be invoked via an API, but they support MPS input.

CLP, and perhaps COIN-OR as a whole, descends from a proprietary IBM product called OSL, "Optimization Solutions and Library", which had its last release in 2000 and was withdrawn in 2003 with a recommendation that users switch to COIN-OR:

> For those customers considering options in the area of Optimization, IBM is strongly recommending the use of the open-source COIN solution. Much of the IBM Research Division's efforts in Optimization for the past few years (along with effort from many others in the open-source community) have gone into developing and enhancing COIN. It has progressed to the point where IBM Research feels that COIN is equal or superior to OSL in most respects....and COIN is very competitive functionally and from a performance viewpoint with other commercially available Optimization offerings.

According to the user guide, running `make unitTest` builds an executable called `clp` which can be used as a standalone solver with MPS input. Presumably the `cbc` executable is similar.

(I suspect IBM acquired ILOG and thus CPLEX around 2003; the announcement above suggests that it was maybe a bit after 2003, since it doesn't mention CPLEX.)

DSDP (solver)

The abstract of the DSDP tech report (ANL/MCS-TM-277) says:

> DSDP implements the dual-scaling algorithm for semidefinite programming. The source code if[sic] this interior-point solver, written entirely in ANSI C, is freely available. The solver can be used as a subroutine library, as a function within the MATLAB environment, or as an executable that reads and writes to files. Initiated in 1997, DSDP has developed into an efficient and robust general purpose solver for semidefinite programming. Although the solver is written with semidefinite programming in mind, it can also be used for linear programming and other constraint cones.

Apparently semidefinite programming is some kind of generalization of linear programming:

> All linear programs can be expressed as SDPs, and via hierarchies of SDPs the solutions of polynomial optimization problems can be approximated. ... A linear programming problem is one in which we wish to maximize or minimize a linear objective function of real variables over a polytope. In semidefinite programming, we instead use real-valued vectors and are allowed to take the dot product of vectors; nonnegativity constraints on real variables in LP (linear programming) are replaced by semidefiniteness constraints on matrix variables in SDP (semidefinite

programming).

The paper that introduced semidefinite programming gives further context.

> Semidefinite programming unifies several standard problems (eg, linear and quadratic programming) and finds many applications in engineering. Although semidefinite programs are much more general than linear programs, they are just as easy to solve. Most interior-point methods for linear programming have been generalized to semidefinite programs.

This all sounds great, but, to me at least, it is not at all apparent how to use the `dsdp5` command to solve a linear program. The documentation is all for the library's C API, and so are the examples.

NLopt (solver)

NLopt is a *non*linear optimization library, so it probably doesn't really belong in this list, but in theory it should be able to solve linear optimization problems too, as a special case; it will be interesting to compare it. It has bindings in C, C++, Fortran, Octave, Python, Guile, and R.

Octave (programming environment)

Although Octave is mostly a matrix computation system, it includes support for linear programming, as well as quadratic programming, general nonlinear programming, and linear least-squares solution of matrices.

Sagemath (programming environment with modeling embedded DSL)

The Sage math programming environment has a mixed integer linear programming module which provides an embedded DSL in the Sage environment for constructing MILP optimization problems. It looks a little clumsier than MathProg. It can use GLPK, CBC, CVXOPT, PPL, or some proprietary solvers.

It *also* has a semidefinite programming module, using CVXOPT as the default (and by default only supported) solver.

I haven't tried it.

Parma Polyhedra Library "PPL" (solver)

The GPL C++ Parma Polyhedra Library provides a kind of generalization of mixed integer linear programming that I don't fully understand, especially suited for static analysis of hardware systems, with APIs in C, Java, OCaml, and Prolog. I haven't tried it. The comprehensive user guide says a lot of things like this:

> Note: The affine dimension *k* <= *n* of an NNC polyhedron P in Pn must not be confused with the space dimension *n* of P, which is the dimension of the enclosing vector space R^n.

CVXOPT (modeling embedded DSL and solver)

CVXOPT is a GPL3+ library for Python under active development since 2004; it is intended for convex optimization but provides a lot of general numerical functionality, including interfaces to GLPK, plus its own solvers for linear and quadratic optimization (under the rubric "cone programming"), as well as nonlinear optimization and semidefinite programming. I haven't tried it but I assume it is suitable for writing your models in too.

Pyomo (modeling embedded DSL)

Pyomo, formerly "Coopr", is an algebraic modeling language like ZIMPL or MathProg, but realized as an embedded DSL in Python, or perhaps more accurately, a Python API, with the concepts closely matching those of ZIMPL or MathProg. It supports linear optimization and also quadratic optimization, general nonlinear optimization, etc. It's able to read model parameters from data files in MathProg format, but not to read MathProg models.

Pyomo can use GLPK, CBC, or some undocumented set of other solvers.

FLOPC++ (modeling embedded DSL)

FLOPC++ — another part of the COIN-OR project — is, similarly, an algebraic modeling language realized as an embedded DSL in C++. I haven't tried it but it looks like roughly the same level of pain as Pyomo. It can at least use CBC.

Google OR-Tools (solvers and embedded DSLs in multiple languages)

Google has a set of free-software "operations research tools" called OR-Tools, including a linear optimizer called Glop, licensed under the Apache license 2.0, with bindings in Python, C++, Java, and C#. Glop doesn't seem to support MIP, but the bundle also includes a constraint solver and a SAT solver, as well as interfaces to external MIP solvers (they recommend SCIP). I haven't tried any of them yet.

lp_solve (solver)

lp_solve is a simplex-method solver, using branch-and-bound for mixed integer programming, which supports the MPS file format. Early versions were proprietary, but recent versions are free software.

I had some trouble getting it to interoperate with GLPK or ZIMPL. I *did* eventually get it to read an MPS file generated by ZIMPL, and I think I know why the GLPK-generated version was failing; my example problem (given in MathProg and LP format above) is a maximization problem, not a minimization problem. Upon translating it to MPS with `zimpl -t mps min.zpl` (see below), ZIMPL issued the following warning:

```
--- Warning: Objective function inverted to make
             minimization problem for MPS output
```

And then lp_solve was able to solve it with `lp_solve -mps min.mps`, and indeed the value of the objective function was -3.66...7 rather than 3.66...7. So I guess that the problem was that MPS doesn't have a way to say you're trying to maximize the objective function, not minimize it.

With this in mind, I was then able to solve the free-form MPS file I had generated with `glpsol -m min.mod --wfreemps min.fmps` by using `lp_solve -max -fmps min.fmps`, `-max` being an `lp_solve` flag to override this. GLPK can also generate fixed MPS files lp_solve can read with its `-mps` flag.

lp_solve has its own "LP format" which is not the same as the CPLEX LP format used by GLPK, ZIMPL, CLP, and CBC, leading to much confusion on my part. The documentation does explain this:

The lp-format is lpsolves [sic] native format to read and write lp models. Note that this format is not the same as the CPLEX LP format (see CPLEX lp files) or the Xpress LP format (see Xpress lp files)

I was able to translate the file from MPS to this incompatible LP format with `lp_solve -max -fmps min.fmps -wlp min.lps.lp`:

```
/* min */

/* Objective function */
max: +2 y +3 z;

/* Constraints */
a: +3 y +2 z <= 3;

/* Variable bounds */
y <= 4;
z <= 1;
```

This can be reduced to the following, a format which is quite reasonable for writing by hand, and still work as input to lp_solve:

```
max: +2 y +3 z;
a: +3 y +2 z <= 3;
y <= 4;
z <= 1;
```

Unfortunately nothing but lp_solve supports this format. (But I think it can translate it to MPS format.)

In theory lp_solve supports CPLEX LP format with the flags `-rxli xli_CPLEX` to read or `-wxli xli_CPLEX` to write, but I think the Debian package failed to include the relevant module, so this doesn't work. (If you really needed to do this, you could use GLPK to translate from LP to MPS so lp_solve can handle it.)

lp_solve even supposedly has an "xli" to read MathProg, presumably linked with GLPK.

lp_solve seems to be much weaker than GLPK's solvers; I have an underconstrained 9x9 Sudoku puzzle, using the above solver, here that GLPK can solve in 4.1 seconds, but lp_solve chewed on it for 22 hours without solving it before I killed it. The two projects started about the same time, but GLPK has seen continued development. I suspect that any current interest in lp_solve is largely because GLPK is GPL, so you can't build proprietary software with it, while lp_solve is LGPL, so you can.

Like GLPK, lp_solve also has an API as well as file formats — in C, Java, and Free Pascal.

Hacks to expand what you can do with a linear optimizer

The description from lp_solve makes linear optimization sound really weak and limiting:

Solve $\mathbf{A}\vec{x} \geq \vec{\mathrm{V}}_1$, with $\vec{\mathrm{V}}_2 \cdot \vec{x}$ maximal.

And, in some ways, it is; but as I'm learning in this class, there are standard hacks for turning a much larger variety of problems into linear problems. Some of these are done for you automatically by

model translators for algebraic modeling languages, while others aren't.

Minimizing vs. maximizing

The simplest hack is switching between minimizing and maximizing. The lp_solve definition above might suggest that you can't minimize things, but of course when $\vec{x}$ is chosen to make $\vec{V}_2 \cdot \vec{x}$ minimal, then $(-1\ \vec{V}_2) \cdot \vec{x}$ is maximized. Moreover, its maximal value is precisely the opposite of the minimal value for $\vec{V}_2 \cdot \vec{x}$; so if your solver wants to maximize, you can just flip the sign on its objective vector and the objective result, and you can minimize.

As mentioned earlier, ZIMPL will do this when generating an MPS file, since apparently MPS can't express whether you want to maximize or minimize your objective function, and so given just an MPS file, GLPK, lp_solve, CLP, and CBC assume you want to minimize.

Mixing ≥ with ≤

Similarly, $\mathbf{A}\vec{x} \geq \vec{V}_1$ suggests that all your constraints must have decision variables on the left side of a "≥". But that's silly; the constraint $2y \geq 4$, for example, is equivalent to $-2y \leq -4$. So by flipping the signs on a row of the matrix and the corresponding limit, you can effectively get both ≤ and ≥ constraints in the same matrix. And that's how you can get criteria like $1 \leq 2y + 3z \leq 2$.

Indeed, even the CPLEX LP format supports this directly for decision variables:

```
Bounds
 0 <= y <= 4
 0 <= z <= 1
```

Decision variables on both sides

Similarly, if you have a criterion like $2y \leq 3z + 1$, you might think that doesn't fit into the $\mathbf{A}\vec{x} \geq \vec{V}_1$ mold. But all you have to do is subtract $3z$ from both sides — $2y - 3z \leq 1$ — and Bob's your auntie!

Equality constraints

Similarly, if you have an *equality* criterion like $3p = 4q - 2r$, you can convert it into two *inequalities* — $3p \leq 4q - 2r$ and $3p \geq 4q - 2r$ — which can only be simultaneously true when strict equality is satisfied. This reduces the dimensionality of your feasible region below the full dimensionality of the decision-variable space, though that's not a problem for the simplex method.

Binary gates with big *M*

A common nonlinearity that can be handled fairly easily by integer linear programming is a sort of "something-or-nothing" criterion; for example, with solar panels, the amount of power you can produce at a site (and thus money you can make) is proportional to the amount of panel area installed at the site, but to produce any power at all at the site, you need to run power lines to the site, which has some fixed cost.

If you have some upper limit M on the power you can produce, you can multiply that upper limit by a binary variable W that encodes

whether or not a site is active; for example, in MathProg notation:

```
param M {s in Sites} := available_area[s] * power_per_area;
var W {s in Sites}, binary;
subject to power_installation {s in Sites}:
    0 <= panels_installed[s] <= W[s] * M[s];
```

When W_s is 0, this forces `panels_installed[s]` to also be 0; when W_s is 1, `panels_installed[s]` can have any value at all, as long as it's less than M, which was chosen to be large enough not to be a restriction in practice. (Solvers can suffer precision problems if M is too many orders of magnitude larger than the actual values of the variable, though.)

Then, you can use this variable W in your objective function to take into account the installation cost.

(I mean "gate" in the sense of something that can block something else from happening, not in the sense of a NAND gate.)

Absolute values

A much cooler trick that doesn't even require dropping back to integer linear programming is encoding absolute values. Suppose one of the terms of an objective function you're trying to *minimize* is the absolute value of some linear function f of your decision variables, $|f(\vec{x})|$. That's not linear, or even differentiable, because it changes direction abruptly when $f(\vec{x}) = 0$, so you can't solve it directly with a linear solver. What you can do is replace $|f(\vec{x})|$ with a new variable t which is constrained to be greater than or equal to $|f(\vec{x})|$ using two new constraints: $t \geq f(\vec{x})$ and $t \geq -f(\vec{x})$. This ensures that $t \geq |f(\vec{x})|$, which, in conjunction with the minimization criterion, forces $t = |f(\vec{x})|$, and again, Bob's your auntie!

When you're trying to *maximize* a sum including $|f(\vec{x})|$, which is a more unusual thing to do, this becomes trickier; you must drop back to mixed integer programming and introduce a binary variable W and maximum M bound and use a variant of the binary-gate trick to prevent t from ever being strictly greater than $f(\vec{x})$, with $t - f(\vec{x}) \leq WM$ and $t - (-f(\vec{x})) \leq (1-W)M$.

Mutual exclusion

As demonstrated in the Sudoku example above, you can use a simple sum condition to require that precisely one of a set of options be chosen: $W_1 + W_2 + W_3 = 1$.

Piecewise-linear functions

Mutual exclusion allows you to use piecewise-linear functions in your linear constraints and objectives. You can't do this in the obvious way — $W_1 f_1(x) + W_2 f_2(x) + W_3 f_3(x)$, with the f_i being the linear pieces — because it wouldn't be linear. But you can introduce new "abscissa" variables z_1, z_2, z_3 that are gated by the W_i and restricted to their own regions: $3W_1 \leq z_1 \leq 6W_1$, for example, restricts z_1 to $[3, 6]$ when $W_1 = 1$, and 0 otherwise. Then $f_1(z_1) + f_2(z_2) + f_3(z_3)$ gives you the piecewise-linear function you wanted, and $z_1 + z_2 + z_3$ gives you its abscissa, which perhaps you want to constrain to be equal to x.

This is a variant of the binary-gate trick that requires no big M.

(I've written it here with single scalars z_i, but you can extend it to

piecewise-linear functions of multiple variables in a few different ways.)

This has the disadvantage that if your piecewise-linear function is discontinuous, it ceases to be a *function* at the discontinuities; it can take two different values.

(I think there's also a way to make a *convex downwards, continuous* piecewise-linear function out of absolute values without involving integer linear programming, in the case where you're minimizing, but I haven't seen it worked out yet.)

Ratios between linear forms where the denominator has known sign

Topics

¿Qué necesito para relación de pareja?

Kragen Javier Sitaker, 2016-03-09 (6 minutes)

¿Qué necesito en una relación de pareja?

Supongo que lo más básico es el amor: la aceptación incondiconal de otra persona, el reconocimiento psicológico de la naturaleza ilusoria de la separación entre las personas: que la diferencia entre mi bienestar y tu bienestar es una ilusión.

Pero el amor sólo, como emoción o estado psicológico, no es suficiente. Es igual o más importante el respeto para la autonomía de la otra persona: si bien no hay una diferencia real entre los estados del mundo que son buenos para mí y los que son buenos para vos, hay otras cuestiones. Cada une tiene su propia información acerca de sus propias necesidades — sé cuando tengo hambre y que tan importante es para mí ver a mi mamá, lo sé más que sabés vos, mientras que vos sabés más cuando vos tenés hambre. También, cada une tiene su propio sendero de desarrollo y crecimiento.

Más que nada, para mí, el rol de la relación de pareja es apoyar y habilitar este crecimiento de cada une. Para eso se necesita un respeto profundo y una alta priorización de ese bienestar y también tal crecimiento: a veces esto implica interrumpir otras cosas (reuniones, trabajo, viajes, sueño) para cuidar a la pareja, y a veces (más difícil) resulta necesario soltar algún mambo de cada une para lograr la capacidad de estar presente para le otre. Por ejemplo, si estoy muy apegado a mi concepto de que soy súper empático, me puede costar mucho reconocer los momentos que no logré empatizar, y eso me puede obstaculizar en prestar más atención al tema y tener la humildad para escuchar que había equivocado.

Con el traspaso de los años, cada une puede hacer a le otre más fuerte y capaz, a través de enseñarle cosas, pero también a través de hacer cosas que le otre no logró. Si sé hacer un cappuccino y vos no, te lo puedo enseñar, o te puedo hacer un cappuccino cuando lo querés. Así volveremos más fuertes en equipo que solos.

Otro aspecto de trabajar en equipo (cosa que veo como fundamental de la pareja) es compartir perspectivas. Ya que cada une tiene su propia perspectiva acerca de cualquier cosa que experimentamos, al consultar a le otre, tendremos una perspectiva más completa, y podremos actuar más inteligentemente juntes que separades.

Al unir recursos y negociar soluciones mutualmente agradables para las situaciones que enfrentamos, podremos lograr mucho más juntes que soles. Si vos querés pasar unos años en la universidad, poder vivir conmigo mientras estoy ganando plata será mucho más conveniente que intentar ganar plata mientras estudiás; al cocinar juntes, siempre que nos podemos poner de acuerdo en qué comer, tendremos mucho mejor y más comida por menos esfuerzo.

En otros casos, hay muchas ventajas de que estamos en dos cuerpos distintos, no el mismo cuerpo; más allá que los placeres del sexo (cosa que nos nutre emocionalmente de una forma profunda) poder estar en dos lugares distintos a la vez nos permite hacer cosas que no podríamos

hacer soles: mirar el doble de las cosas, llevar cosas acá y allá, etc. Eso solo resulta ventajosa para les dos mientras tenemos una relación igualitaria; si no, vuelve un suerte de explotación, en lo cual la pareja más débil está sirviendo a le otre, y así tiene mucho menos energía y tiempo para sus propias cosas.

Eso vuelve a la cuestión del reconocimiento de la unidad de intereses: nuestra habilidad de colaborar así siempre está sujeta a las limitaciones impuestas por nuestro egoismo. Mientras siento que lo que te hace bien también por eso me hace bien, puedo pesar los dos bienes dentro de mí; mientras que no, llegamos a las negociaciones, donde te ofrezco hacer algo que me cuesta si vos hacés algo que a mí me importa pero te cuesta a vos. Para mí, los dos aspectos, negociaciones y amor, siempre existen en cualquier relación real. La negociacion es necesaria para convivir con otro ser que no es perfectamente altruísta o no confía que sos perfectamente altruísta.

Pero yo. ¿Qué necesito yo?

Pero todo lo anterior tiene que ver con que pienso que cualquiera necesita en una relación de pareja. No hablé mucho de lo que necesito yo en particular, que tal vez otras personas no necesitan.

Necesito mucho cuidado y suavidad en cuanto navegamos los conflictos. Hay personas que pueden bancar palabras duras, criticándolas agudamente con ataques contra su carácter y naturaleza, y después perdonar con el excusa de que esas palabras eran motivadas por enojo y no representaban una manifestación verdadera de lo que veía le otre. Yo no. Una vez que me decís que soy insincero o estúpido o lo que sea en un momento de enojo, por más que negás creerlo mil veces, siempre después me preguntaré cuál es la verdad y cuál es la mentira de tu creencia: me mentís ahora que piensás que no soy estúpido porque tenés miedo que la verdad me lastimaría, o me mentiste cuando estabas enojade para lastimarme?

Necesito mucha honestidad, en general. No banco muchas mentiras.

Necesito inteligencia. Estuve en relaciones antes con personas de baja inteligencia, y por más que eran personas muy lindas y cuidadosas y honestas, no podía conectarme con ellas a un nivel para formar un buen equipo, mucho menos un buen equipo igualitario.

Necesito la libertad de estar en relaciones íntimas con otras personas.

Necesito, creo, una conexión sexual activa y fuerte.

Topics

- Psychology (p. 3669) (18 notes)
- Journal (p. 3532) (11 notes)
- Español (6 notes)

Energy storage efficiency

Kragen Javier Sitaker, 2019-07-30 (4 minutes)

From https://news.ycombinator.com/reply?id=20561792&goto=threads%3Fid%3Dkragen%2320561792

Indeed, I think the finite amounts of existing dams are the reason people are looking to batteries.

Your point about the efficiency is interesting, although I didn't understand it at first. I think you're saying that the capital cost of the lithium-ion battery storage is partly defrayed by the higher efficiency of the storage system? Like, for each kilowatt-hour of Li-ion storage (with, let's say, a round-trip efficiency of 95%, although I think that's too high), you "get back", say, 0.25 kWh every time you use it, that you would have lost if you'd stored that energy in pumped storage instead? So over, say, 15 years, you "get back" US$82 or so, at US$5.48 per year?

https://electrek.co/2018/11/20/tesla-gigafactory-battery-cells-made-cost-advantage-panasonic-lg-report/ claims that the battery cell cost is US$111 per kWh at the moment (though other manufacturers are still stuck around US$140), so that would work out to about a 5% annual IRR if the cells were the only cost; I think that in fact they are on the order of half the cost (though Tesla's blog post here doesn't actually list prices!) and so that would be a 2.5% or so IRR. Not enough to justify the battery investment on its own, but it would definitely be a significant boost to the project's ROI.

I have a couple of objections to that line of reasoning, one trivial and one serious.

The trivial objection is that the wholesale cost of electrical power, although it varies a lot, averages about half of the 6¢/kWh you're imputing. https://www.zmescience.com/ecology/climate/cheapest-solar-power/ talks about the just-signed Atacama project at 2.9¢/kWh, which I think includes the cost of some storage. So the numbers are more like 1.25% IRR rather than the 2.5% I suggested above or the 5% you suggest.

The more serious objection is that, when you're filling up your utility-scale storage during hours of excess power production, you're not paying 6¢/kWh or 2.9¢/kWh. In fact, due to non-dispatchable "baseload" plants like coal and nuclear, it's common right now for *the power plant to pay you* to take the power, with the price typically around -4¢/kWh, which is the cost of burning it up in giant resistors. When instantly-dispatchable solar plants come to dominate power production, we can expect to see a price floor of 0¢/kWh. Maybe if a storage-plant operator is paying a solar-plant operator to leave their PV plants running, they'll have to pay 0.01¢/kWh or 0.1¢/kWh. But they won't be paying anywhere close to the average price of electrical energy. They'll be paying the *marginal* price of generating electrical energy *when it is cheapest*.

So that means that the amount of money you make from a utility-scale energy storage plant isn't going to be determined by how

much energy you need to charge it up. Your round-trip energy efficiency could be 10% or 5% and you still wouldn't pay a significant percentage of your revenues to obtain that energy. What determines your revenues is *how much energy you can release* once you are selling energy rather than buying it. (And the quality of your trading strategy, of course; if you decide to wait to sell your energy until your LMPs go above US$45/MWh, and they sit at US$42/MWh all night long, you don't make any money.)

Round-trip energy efficiency only matters *at all* in the sense that it diminishes your effective storage capacity — if theoretically you have "1 MWh" stored, but when you turn it on, only 0.9 MWh flows to the grid, you only get paid for that 0.9 MWh, and that's what you need to pay your capex and opex with. But it only matters very marginally whether you had to buy 1.1 MWh or 2 MWh or 5 MWh or 10 MWh to charge up your storage facility.

Topics

Service-oriented email

Kragen Javier Sitaker, 2017-06-20 (updated 2017-06-21) (15 minutes)

What would a REST or REST-plus-invalidations system for my mail look like?

You could plausibly argue that this is the wrong question to ask, because really the right architectural style is something else — something more like Kafka, which gives you a way to safely resend operations that may have side effects, and to incrementally replicate large databases. But let's leave that aside for the moment.

At the bottom of the stack, we have three kinds of resources: a mbox file (which we probably need invalidation notifications, quick ETags, and byte-range fetches for), some kind of metadata store for tags and marks, and an index listing all the mboxes. Everything else is built on top of that.

The mbox-file resource can be provided by a service that keeps open an ssh connection and sends commands over it, or several ssh connections, or with rsync, or local file access, or whatever. Probably the most sensible way to do it would be to tunnel REST requests over an ssh connection to a remote REST service accessing a local file.

The metadata store is probably local and probably can be a simple key/value store.

A layer up, we convert blocks of the mboxes into resources of their own. The interface here is something like

```
GET /size?url=foo/bar/baz
GET /block?url=foo/bar/baz
          &start=180052992
          &bytes=1048576
```

/size and /block are cacheable stateless microservices that merely transform ordinary GET requests into GET-with-byte-range and HEAD requests.

Ideally the mbox blocks would be large enough to be large compared to the bandwidth-delay product, but small enough to amount to a tolerable delay when they are fetched. This is infeasible when my latency is 200+ ms, so I probably have to hide the latency as much as possible with parallelism and prefetching. Specifically, my ping time to canonical.org is currently 175–177 ms, my bandwidth at the office is about 1.5 megabytes per second, and a tolerable delay is 100 ms, which means that we want to ensure that 95%+ of our requests can be served from a local cache, which implies very aggressive cache prefetching. Or sucking it up, I guess.

The bandwidth-delay product is about 256 kilobytes, coincidentally almost exactly the bandwidth-delay product of a spinning-rust disk (though about 64 times that of an SSD being read). So, with chunks of 256 kilobytes, we could maintain full bandwidth usage with a parallelism factor of 2; with 128 kilobytes, a parallelism factor of 4; or with 64 kilobytes, a parallelism factor of 8. Probably the best overall compromise is a parallelism factor of 8 and 128-kilobyte chunks. Then, on the rare occasion that we have to suck up a cache miss, the size of the 128-kilobyte chunk will add 85 ms of

latency on top of the 175 ms already inherent in the link — not insignificant, but not the main bottleneck.

My current mailbox is 4.3 gigabytes, which will take about 45 minutes to initially download at 1.5 megabytes per second.

Given the blocks, we need to find the messages within, and the message boundaries ("\nFrom " or "\AFrom ") might cross block boundaries. For this we use a stateless boundary-parsing service:

```
GET /bloxparse?url=/block?url=foo/bar/baz%26start=180052992%26bytes=1048576
```

This returns three items: an array of (potentially many) byte offsets where definite message boundaries within the block are found, an array of byte offsets before the beginning of the block which could potentially be message boundaries (containing zero or one item; this is if the block begins with something like "rom "), and an array of byte offsets just before the end of the block which could potentially be message boundaries (similarly, containing zero or one item, for cases where the block ends with "\nF" or something.)

If I run this locally, it should be cached pretty aggressively, because I have 181490 message boundaries in those 4.3 gigabytes, so a single 11-byte decimal number or 8-byte binary number stands in for 23 kilobytes of data. This very minimal summary, which can be generated in about 10 CPU seconds (with the bsdmainutils from command, for example), occupies about 2 megabytes and can avoid transferring 4.3 gigabytes.

This /bloxparse service is invoked by an /mboxparse service which manages some amount of concurrency and reconciles the possible offsets near the ends of blocks.

(In a more generalized sense, they aren't so much possible offsets as sets of candidate finite state machine states: if the finite state machine is in state 3 or 5 at the beginning of this block, then it has found a message boundary; at the end of the block it is in state 2. More generally it is a mapping from input states to output states. But maybe that level of complexity isn't needed here, since the message-boundary-parsing service is very simple.)

The /mboxparse service takes the same parameters and provides the same result format as /bloxparse, but also takes optional blocksize and nprocs parameters.

However, it isn't necessary to parse the whole mailbox in order to start parsing message headers and contents from it and processing queries. We can chain two calls to the /block service to fetch a message, the second-invoked one fetching a 128k-aligned block and thus more neatly cacheable, and the other one extracting a message from it according to /mboxparse.

Somewhere nearby we should have a service that reformats /mboxparse results into lists of links to messages, and perhaps another that maps somewhat simple links to messages into redirects to the double-chained /block resource mentioned above:

```
/msg?mbox=foo/bar/baz
    &start=180053082
redirect to
    /block?url=/block?url=foo/bar/baz
        %26start=180052992
```

```
    %26bytes=1048576
&start=180053082
&bytes=20330
```

It would be nice to be able to somehow incrementally list the messages that have been parsed so far, like with some kind of re-rereadable message queue, like Kafka. Lacking such a thing, we can prefetch stuff in the background, I guess.

Once we have some message URLs, we can start parsing those messages so we can index them, both with full-text indexing and with traditional database-column indexing. Perhaps we maintain cached index segments corresponding to segments of the mailbox, and cache merge results for those index segments, re-merging index segments on demand.

A parse request might look like

```
GET /parse822?url=foo/bar/baz
```

where foo/bar/baz is something like the /msg? URL mentioned above. It returns some kind of parsed representation of the message headers, perhaps encoded using FlatBuffers; this parsing can be cached. (Maybe /hdr is better?)

A query process might fetch index segments (computed on the fly if necessary) for ever-larger sections of the mailbox, starting from the end, running sort-merge joins on them, perhaps ceasing once it has enough results to fill the screen (or a bit more), and perhaps maintaining a couple of different query frontiers open at once — full-text index segments and header index segments, for example.

The desire to be able to move various kinds of processing to the storage server suggests that it would be best for the service identifiers like /parse822 to be mapped not to processes on machines but to pieces of code, perhaps bundled up with some kind of handle to a persistent state for the services that aren't purely stateless. Then a sort of query optimizer can try running the service locally, and if that's taking a long time (e.g. >10ms), try running it remotely as well and use whichever one gets faster results. Some kind of standardized bytecode would be one way to handle such mobile-code services. (Also you could hide this behind a process-on-machine resource.)

All of this somewhat abstracts away the question of bandwidth usage, treating it as an implementation detail of the services, when it kind of isn't really — if I have a bunch of useless local jobs sucking up all my uplink bandwidth, it's going to make anything else I try to do slow. You could conceivably pass in a handle to an accounting resource to the service when you invoke it for it to charge its bandwidth and even CPU usage to, which would have the ability to cut it off and cause it to fail fast if it went over its resource limits or if you just no longer cared about the results.

All of this is about reading mail. Sending mail is a different matter; we want to ensure that mail gets sent exactly once. If the client has a private namespace on the server (/outmail/myhostname) it can peremptorily allocate message-IDs within there, assuming it hasn't suffered an attack of amnesia due to some kind of virtual machine checkpoint and restart (which could cause it to reuse the same

message ID), and PUT messages there. As long as PUT is atomic then it's all good; retries are safe. In HTTP, PUT also has an If-None-Match: * option which can be used to ensure that you never overwrite an existing message, but what do you do in that case? (I guess you can GET the message to see if you previously PUT the same message and forgot about it, or whether it's a collision.) If there's some kind of collection indexing, then a mail-sending process on the server can watch the collection index for changes, passing the mail to Postfix or whatever when it sees a new message.

That is, Kafka-like topics aren't necessary for reliable exactly-once mail sending. HTTP semantics are more than adequate.

Other useful stateless services:

```
/mime?url=foo/bar/baz
```

Gives you an index of the MIME structure of the RFC-822 message; the links add more parameters and contain enough data to do the parsing efficiently.

```
/col?parser=/parse822
    &col=subject
    &items=/msglist?foo
```

Passes the message URLs in /msglist?foo through /parse822?url=$url and extracts URL-subject pairs (the "subject" "column") from them.

```
/invert?from=bar
    &to=baz
    &data=/col?foo
```

Invokes /col?foo and inverts it, converting values to keys and keys (such as message URLs) to values, and sorting by the new key. Excludes rows whose keys are outside the range from bar to baz. (Maybe the exclusion should be elsewhere?)

```
/union?data=/a
    &data=/b
    &data=/c
```

Merges some sorted sequences of key-value pairs. Note that applying /invert with a range to /union gives you a single-field query over a merged index.

```
/join?data=/a
    &data=/b
    &data=/c
```

Given some sequences of key-value pairs, this returns one row for each key that occurs in all the datasets, with the values for each dataset concatenated.

By applying /join to some /inverts applied to /inverts with ranges applied to /unions of the appropriate /cols, you can get a multi-field query. If you leave out the /inverts, you get a message index.

Ideally every service should be self-describing in that if you invoke

it without all the mandatory parameters, it gives you IDL for how to call it, an HTML form or equivalent. Also, we need some way of assigning media types to URLs and URL input fields to make it easy to plug things together, and a better invocation syntax with less noisy parameters, and supporting URL nesting. Something like Clojure keyword syntax.

Also, both requests and responses should be able to contain capabilities, in order to avoid confused-deputy attacks.

An interactive prompt should promiscuously prefetch stuff as you are composing your request so you can see what you're doing, since GET is safe. If you do some interactive exploration, you should then be able to go back and factor out some parameters from your script, then turn it into a new microservice.

Minimal implementation

Of course I don't need all this shit to be able to try it out, and I do need my mail pretty soon. A mini HTTP server with a small number of simple scripts (parse message starts, provide mailbox size and mailbox blocks) that I can tunnel over ssh should be pretty doable. I'd need some special-purpose caching logic but nothing really special. Also mapping some URLs to ssh-tunneled URLs and others to local scripts seems like it should be pretty doable.

In terms of data formats, I can probably get by with space-separated URL-encoded fields with newline record terminators for now. Or Excel CSV, which is probably just as easy, but doesn't sort properly.

Sorting and caching is probably pretty easily done with LevelDB or maybe Redis. Or both. LevelDB on my laptop can handle about 300k record insertions per second, so about half a second to sort the subjects of all my messages. /bin/sort sorts my 100k-line ~/netbook-misc-devel/bible-pg10.txt in 770 ms, or 420 ms with `LANG=C`, which is the same speed (even with `-S 1G`). It seems like it should be possible to go faster since that file is only 4.4 megabytes; `LANG=C wc` takes 100ms.

As another sample computation, `time grep -a '^Subject: ' adjuvant-mbox.1g | wc` finds 43893 subject lines in 1073741824 bytes (1 GiB) containing 41630 message starts in 11.5 seconds. Oh wait, it takes 740ms if it's already in memory, then another 740ms if I pipe it through sort. The total time for `grep | sort | wc` drops to 1000ms with `LANG=C`, which mostly affects sort.

HTTP/2 supports out-of-order responses over a single connection, so using HTTP/2 would avoid the need to use multiple HTTP connections. The quasi-required encryption would probably hurt performance, but it isn't really required in the standard.

I should check out Hyper the Terminal and see if it has anything interesting (no, not for this). And maybe Spark and Samza.

Topics

• Systems architecture (p. 3691) (48 notes)
• Protocols (p. 3668) (21 notes)
• REpresentational State Transfer (p. 3684) (8 notes)

Literate programs should include example output, like Jupyter, but Jupyter is imperfect

Kragen Javier Sitaker, 2018-04-27 (3 minutes)

For decades, I've appreciated the aspiration of literate programming, to produce software that can be pleasantly and easily read as literature. However, notably, for decades, literate programming has remained unpopular. I think the biggest reason for this is that literate programming as we have practiced it so far omits one of our most powerful tools for understanding programs in practice: running them.

That is, standard literate programming tools like CWEB or noweb include nothing in the readable text that is contingent on what the program actually does when you run it. Typical literate programs don't even include unit tests, which implicitly make claims about what *would* happen if the program were to be run — if the author tells you the unit tests succeed, then you can suppose that that is what actually does happen. In cases like tests written with Python `doctest`, the test case shows the actual output from the program.

The Mathematica-derived "notebook" interface of IPython/Jupyter seems like a promising step forward, because it does show example output. Peter Norvig is working on a set of such notebooks called pytudes to demonstrate some challenging problems.

Jupyter notebooks are pretty awesome, but they have some limitations. To wit:

Because Python is an imperative language, the interpreter state (and thus the results of cell evaluation) depends not only on the contents of the notebook, but also the order in which the cells have been evaluated — and, potentially, even the number of times each cell has been evaluated, and previous cell contents that were evaluated, even if they no longer exist.

Perhaps worse, there's no way to "export" a class, function, or value defined in a notebook so that it can be used from elsewhere. Each notebook is relatively self-contained — for better or worse. You can't use a notebook as a way to present and document one module of a larger system, although you can use them as a way to document how to *use* such a module. (There is the somewhat primitive `%run` mechanism, which I think is analogous to C's `#include`.)

Jupyter notebooks are also less than ideal for multiuser use. There are a variety of ways you could imagine a multiuser notebook working — for example, all the users could all be editing a single shared document, or each one could have their own document that imports certain cells from other documents, or some combination.

The flatness of Jupyter notebooks makes them hard to navigate once they're over a certain size. Hyperlinks and folding could help.

However, it's important to point out all the ways the Jupyter notebook interface is vastly superior to traditional command prompts, REPLs, and source code editing environments and even, in most cases, traditional literate programming tools:

• Typography: headers, equations, bold and italics;
• Collapsible output (i.e. you can hide the output of a cell, and actually the whole cell, if you like);
• Multimedia output — plots, raw HTML, tables, equations.
• Pywidgets provides interactive experimentation with variable values.

Topics

Heating my apartment with a plastic tub of hot water

Kragen Javier Sitaker, 2018-06-17 (3 minutes)

My apartment receives hot water from a shared hot-water heater, and taking a hot shower warms up the 100 m³ apartment noticeably. So I filled up a large plastic tub and dragged it into the living room.

This plastic tub is 330 × 300 × 600 mm, more or less, and it's full of 40° water while it's 15° outside. This is about 60 kg of water, which has the thermal mass of about 120 kg of air, and about 1500 kilocalories or 6 megajoules. If this tub takes 6 hours to cool off, which seems plausible, it's heating the apartment at almost 300 watts on average during that time.

To heat the apartment with the hot water supply at the 2000 W or so it really needs, I would need about 19 milliliters per second of water, or about 70 liters per hour, considerably below the shower's output of some 300 mℓ/s. The hot water flow would need to be intermittent, because the first liter that comes out of the faucet is cold, and the first ten or twenty liters are noticeably below maximum temperature — drawing water continuously at 19mℓ/s would just warm up the pipes in the wall, not the interior of the apartment.

Refilling one of two 35-ℓ hot water tanks every half hour would take about 2' out of every 30', with the final result being equivalent to a 90-minute shower every day.

So my shower is heating my house at 300 mℓ/s · 25° · 1 kcal/kg/° · 1 g/cc = 31 kW. That's pretty respectable, especially given that the electrical service is only 66 amps, 16 kW. Too bad that, in its current form, the shower also raises the humidity uncomfortably high.

The hot-water tanks could take the form of tall cylinders with hot water at the top, cold water at the bottom, and a heat exchanger in between, driven either by thermosiphon action or by an actual pump. At 140 mm diameter and 2.5 m tall, they would hold 38.5 liters and thus 4 MJ of heat at $\Delta T = 25°$. By separating the cold water from the hot water in the same tank, they would avoid the need to have double the tank capacity to separate hot from cold, or to put air into the tanks between emptying and refilling, which can promote rusting.

However, if you just left them uninsulated and didn't use a heat exchanger, the result could be adequate with a bit more work. Each tank as described would have a surface area of 1.1 m², and at 40°, a black body radiates 545 W/m², so this would be about 600 W of heat emission per tank — counterbalanced by absorbing 430 W from the environment, so only 170 W net. This is about an order of magnitude too low, suggesting that you could get the right result by flattening the tank out to be an order of magnitude thinner and thus wider.

Topics

• Physics (p. 3632) (119 notes)
• Thermodynamics (p. 3747) (49 notes)

Cobstrings

Kragen Javier Sitaker, 2015-08-21 (updated 2015-08-31) (5 minutes)

COBS ("consistent overhead byte stuffing") is a byte-stuffing method for avoiding NUL bytes in your packets so that you can use them for framing which has some interesting virtues. You take your packet

```
f o o\0\0\1\0 x y z\0\7
```

append a NUL to it so that it's a concatenation of NUL-terminated strings, then compute the lengths of all the NUL-terminated strings:

```
f o o\0\0\1\0 x y z\0\7\0
3       0 1   3       1
```

Now you convert the NUL-terminated strings to counted strings with the counts starting at 1.

```
f o o\0\0\1\0 x y z\0\7\0
3       0 1   3       1
\4f o o\1\2\1\4 x y z\2\7
```

So now your packet is the same length as before but contains no NULs, if we count the terminating NUL we added, through a reversible transformation that eliminates the NULs.

But that's impossible, by the same pigeonhole principle that shows that file compression has its limits. And indeed we run into the standard problem with counted strings: you run out of counts. What do you do if you have more than 254 bytes between NULs?

COBS solves this by reserving \xff for a 254-payload-byte prefix block, which does not have a terminating NUL; so a COBS string in general consists of a sequence of zero or more \xff-prefixed blocks followed by an excess-1-count-prefixed block, which may be empty.

It occurred to me that this representation for in-memory strings has some advantages:

• Like C strings, it only requires a single extra byte of overhead, at least in the cases where C strings are a good idea.
• Like C strings, you can cut a string into tokens in place by overwriting separators in it with metadata. (This requires moving around later parts of the string if it's long, though.)
• Unlike C strings, it's 8-bit clean; it can store any byte, including NUL bytes, avoiding security holes, among other problems. This means COBS strings can be safely nested!
• Unlike C strings, appending to a long string (of a few kilobytes) is reasonably efficient, as is taking its length; so `stpcpy` is unnecessary. And Boyer-Moore search is plausible.
• Unlike C strings, you can copy it several bytes at a time.

It has a couple of disadvantages compared to C strings: code implementing fundamental string operations is a bit more complicated; you can't compute a suffix of a string without mutating it, which means that a number of standard library functions need an

extra "start index" argument; and copying data into and out of the string requires special consideration for possible chunk boundaries.

Here, I'm eliminating as unhelpful in this context the constraint to not use NUL bytes, so a \0 length byte is an empty string, a \1 length byte is a single-byte string, a \xfe byte is a 254-byte string, and the \xff byte prefixes 255 bytes of payload data.

So a long string might look like this in memory:

```
FF C O B S ( " c ... s oSPFF t h a t ... s t a rFF t i
 n g...SPSPBE / * c o bSP c h u n k ... \nSPSPSPSP}\n
```

with the \xBE at the beginning of the final block both signaling its length and

I think these functions may be correct, but I have not tested them.

```
void cobcpy(unsigned char *dest, unsigned char *src)
{
   while (src) {
      memcpy(dest, src, cobclen(src) + 1);
      dest += 256;
      src = cobcnext(src);
   }
}

void coblen(unsigned char *s)
{
   size_t n = 0;
   for (; s; s = cobcnext(s)) n += cobclen(s);
   return n;
}

void cobcat(unsigned char *dest, unsigned char *src)
{
   while (src) {
      while (cobcnext(dest)) dest = cobcnext(dest);
      cobaddbytes(dest, cobcbody(src), cobclen(src));
      src = cobcnext(src);
   }
}

/* returns index of first c in cob, or coblen if not found */
size_t cobchr(unsigned char *cob, char c) {
   size_t n = 0;
   while (cob) {
      unsigned char *m = memchr(cobcbody(cob), c, cobclen(cob));
      if (m) return n + m - cobcbody(cob);
      n += cobclen(cob);
      cob = cobcnext(cob);
   }
}

int cobcmp(unsigned char *a, unsigned char *b)
{
   for (;;) {
      int n = cobclen(a), nd = cobclen(b) - n;
```

```
      if (nd < 0) n += nd;
      int d = memcmp(cobcbody(b), cobcbody(a), n);
      if (d) return d;
      if (nd) return nd;
      if (n != 255) return 0;
      a += 256;
      b += 256;
   }
}

/* append n bytes starting at src to cob */
void cobaddbytes(unsigned char *cob, unsigned char *src, size_t n)
{
   while (cobcnext(cob)) cob = cobcnext(cob);
   int available = 255 - cobclen(cob);
   int to_copy = (n < available ? n : available);
   *cob += to_copy;
   memcpy(cobcbody(cob) + cobclen(cob), src, to_copy);

   /* copy any remaining bytes into a new chunk */
   cob = cobcnext(cob);
   if (cob) {
      int tail = n - to_copy;
      *cob = tail;
      memcpy(cobcbody(cob), src + to_copy, tail);
   }
}

/* cob chunk next; returns NULL if there is no next chunk */
unsigned char *cobcnext(unsigned char *s)
{
   return (cobclen(s) == 255 ? s + 256 : 0);
}

/* cob chunk length */
int cobclen(unsigned char *s)
{
   return *s;
}

/* cob chunk body */
unsigned char *cobcbody(unsigned char *s)
{
   return s+1;
}
```

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- C (p. 3359) (28 notes)

Steampunk spintronics: magnetoresistive relay logic?

Kragen Javier Sitaker, 2013-05-17 (15 minutes)

I thought it was possible to do high-speed digital electronic computation with 19th-century materials and fabrication techniques, using the (ordinary) magnetoresistive effect and differential circuits to obtain nonlinear amplification. This document explains why I thought it was possible.

However, now I think it's *not* possible, because it involves using iron or otherwise ferromagnetic wires to carry the signals that need to be switched, and those wires, in addition to being magnetoresistive, are extremely self-inductive; the consequence is that they can only practically carry very slowly varying signals (1Hz or worse). You could try to balance the heavy self-inductance with a coaxial sheath to add capacitance, but I'm not convinced that will work adequately.

The rest of this document was written before I knew about this problem.

Electromechanical relays

Electromechanical relays are a relatively simple invention, once the idea is pointed out: an electromagnet pulling a springy bit of metal into or out of electrical contact, opening or closing an electrical circuit. You can make one with a little fuss if you have some wire and ferromagnetic metal, and one version was invented in 1840 by Samuel F.B. Morse for his Telegraph. They are sufficient for building Boolean logic devices to perform arbitrary digital computations, as shown by Claude Shannon in his thesis and extensively exploited for telephone switching in the early 20th century. Indeed, since they're fairly close to ideal switches (very unlike bipolar transistors), logic circuits built using them are quite simple, using only one or two relays per gate or flip-flop, even without any elaborate mechanical tricks.

However, for digital computation, relays have some serious drawbacks. Modern silicon semiconductor devices can operate comfortably in the 4GHz range (i.e. 250ps per cycle) at room temperature; at higher cost, germanium, silicon germanium, gallium arsenide, ECL, and/or further cooling can push speeds higher, past 10GHz (100ps).

By contrast, electromechanical relays typically can't operate faster than about 100kHz (10 μs, 10 000 000 ps). This is a pretty big disadvantage, and it's the reason relays have been out of favor for digital logic since the 1940s. The basic problem is that the metal contacts are macroscopic objects full of baryons, so you have to move an awful lot of electrons through one set of contacts to produce enough energy to move another set of contacts. The vacuum tubes that replaced relays in the 1940s move only electrons, which have three orders of magnitude less mass per unit charge, and semiconductors share this advantage.

(Relay coils also have substantial inductance which limits the speed at which current in them can rise and fall.)

(Semiconductor devices also have the advantage of being very small, speeding up the timescale of anything, but this advantage could in theory be reproduced by other kinds of devices as well.)

A digression: relay logic circuits

A normal single-pole double-throw relay has five terminals:

- `C1` and `C2`: two terminals on the coil;
- `A`, an armature terminal;
- `NO`, a normally open contact which gets connected to `A` after a delay when there's current flowing through the coil; and
- `NC`, a normally closed contact which is connected to `A` except (after a delay) when there's *no* current flowing through the coil.

There are any number of possible ways to do digital logic using relays. This one is among the simplest.

Let's us a kind of "open-collector" convention: we represent logic 0 as a low-resistance path to ground, and logic 1 as the lack of a low-resistance path to ground. In that case, you can construct a basic inverter as follows, with V+ being the power supply rail that doesn't represent logic 0:

```
X' = Not(X): Relay(C1=V+, C2=X, A=gnd, NC=X')
```

This inverter is more than it appears: the normally-open contact `NO` is actually available as a buffered copy of the input `X`, and any number of outputs can be connected to the inverter's input and will be implicitly ANDed, providing fan-in, the ability to combine different pieces of information. That is, this isn't just an inverter; it's a single-relay combination N-ary AND-NAND gate, or a buffer.

(You can safely stick a resistor in series with the coil of each relay operated this way, since the coils are never placed in series with each other.)

You can put together an RS latch in the usual way:

```
Q, Q' = RS(R', S'): Q' = S' = Not(R'); Q = R' = Not(S')
```

That is, you hook up the two inverters in a loop, each inverting the other's output — wire-ANDed with an active-low asynchronous reset or set input. This gives you a bistable circuit whose state you can change, which gives you memory, making sequential circuits possible.

The fanout of this kind of relay logic will be limited only by the number of coils whose current can be run through a single set of relay contacts before the resistance of the point of contact creates a substantial voltage drop. Typically the fanout provided in this way is very large. As long as the fanout is greater than 1, you can build circuits that do universal computation.

There are two more features crucial to digital logic: level restoration and inversion.

Level restoration means that, if the input to an inverter only barely qualifies as a logic 1, the output needs to be a logic 0, but not "only barely" — it needs to be a better 0 than the input was a 1, or if the input was barely a 0, the output needs to be more-than-barely a 1. That is, you need a kind of "range compression" between the input and output. Relays make this really easy: when the contacts are open,

the resistance across them is some 20 orders of magnitude higher than when they are closed.

Inversion is the ability to get a zero from an all-ones input, or a one from an all-zeroes input. Some kinds of digital logic used in the past were capable of producing various combinations of AND and OR using only diodes and resistors, but not inversion, which requires some kind of amplifying element.

So whatever kind of thing you use for digital logic will need memory, fan-in, fanout, level restoration, and inversion.

The magnetoresistive effect

Kelvin (alias William Thomson) discovered what is now known as the "magnetoresistive effect" in the 1850s:

W. Thomson, "On the Electro-Dynamic Qualities of Metals: Effects of Magnetization on the Electric Conductivity of Nickel and of Iron", Proceedings of the Royal Society of London 8, 546–550 (1856–1857). http://archive.org/details/philtrans05965210 http://zapatopi.net/kelvin/papers/on_the_electro-dynamic_qualities_of_metals.html http://archive.org/stream/philtrans09842257/09842257#page/no/mode/2up

He found about an 0.5% change in the resistivities of ferromagnetic metals when a strong magnetic field was applied, depending on whether the field was parallel to the current or at right angles to it. Importantly (and necessarily for the P-symmetry of the electromagnetic force), the resistivity change doesn't depend on whether the magnetic lines of force run in the same direction as the electrical current or the opposite direction.

0.5% is a pretty small change, and Thomson had difficulty measuring it accurately. (Wikipedia claims he got up to 5%, but I'm not seeing that in the papers he published. Other sources claim you get 5% with a 90% nickel, 10% iron alloy, but I'm not clear on how sensitive that mixture is to impurities.)

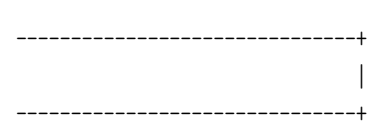
Speculation: ordinary magnetoresistance provides amplification

But I speculate that you can *almost certainly* use this "ordinary magnetoresistance" effect to produce a solid-state "relay" that provides amplification, using 1850s technology! Better yet, you can produce a kind of low-quality radio detector, which was the crucial invention that made radio transmission of audio possible, replacing the electromechanical coherer invented in 1890.

This speculation depends on the crucial assumption that the current running through the magnetoresistive element does not produce a magnetic field that tends to diminish the resistance — that is, that the magnetoresistive effect does not diminish at higher current densities. If this assumption is wrong — and I'm still not sure, because I don't understand the physics — then it may not be possible to use a magnetoresistive element to control a larger amount of power using a smaller amount of power.

I argue that this assumption is reasonable for two reasons.

First, consider a long, thin, flat, wide insulated metal tape, folded in half the middle:



An electrical current through this tape will produce magnetic fields that very nearly cancel, but the magnetoresistance (produced by an externally applied magnetic field) of each half of the tape should still be nearly the same as before you folded it in half.

Second, resistivity is typically greatest when the current flows *parallel* to the magnetic field. But in that case the magnetic field from the current is at right angles to the applied magnetic field, and so will not affect it in any way.

These are not ironclad arguments: the "cyclotron orbits" pursued by the electrons inside the metal might still produce magnetic fields in unexpected orientations.

Balance: Wheatstone bridges to amplify output

There's still the issue of the 0.5%, though. If the difference between zero and strongest magnetic field you can provide changes the resistance of your magnetoresistive element by only 0.5%, then your current will vary by only 0.5%, or less still, which means that if you just use that current directly to create another magnetic field, that other magnetic field will vary by only 0.5%. Which is to say, you completely fail at amplification.

But that's not the only thing you can do with a variable resistance. If you balance the resistance of your magnetoresistor with a similar-sized resistor in a Wheatstone bridge configuration, the voltage across the middle of the bridge will be zero when the magnetoresistor is at its reference resistance, but nonzero when its resistance is perturbed by a magnetic field. To be specific, if its resistance varies by 0.5%, the voltage across the middle of the bridge can be around 0.25% of the total voltage applied to the network.

That doesn't sound like much, but that low voltage can be (by my assumption above) providing an arbitrarily large amount of current — current which can be used to energize another coil to an arbitrarily strong field, a field which will be zero when the bridge is balanced.

This means that you can get amplification, which translates into the necessary overunity fanout. Your amplification is limited only by the precision to which you can balance the Wheatstone bridge.

Magnetoresistive radio detectors

As mentioned earlier, the perturbation in resistance depends on the absolute value and orientation of the magnetic field, not its direction; that means that the perturbation induced by a coil is symmetric around zero in the current applied to the coil. You have some kind of slight resistance bump, up or down, centered on zero.

Typically, this takes the form of "Kohler's rule", which says the extra resistance is proportional to the square of the applied magnetic field.

That's a significant nonlinearity! That means that the average resistance in the magnetoresistor is not the resistance in the magnetoresistor at the average current in the coil. If you have a pure-AC current in the coil, that can still produce a significant magnetoresistive effect, which means you can demodulate AM radio.

Level restoration

How can you do level restoration with magnetoresistive logic? One way is to exploit the symmetry around zero (and indeed the zero derivative at zero implied by Kohler's rule) mentioned in the previous section. You can improve the zero level a bit further by biasing the minimum a bit with a permanent magnet, as described.

You could use a cascade of two inverting magnetoresistive "relays" to restore both the 0 level and the 1 level, but there may be easier approaches; for example, magnetoresistance in many materials saturates at some flux density, and flux density in ferromagnetic materials saturates at some applied magnetomotive force. (Every material tends to relative permeability of unity, in the limit of sufficiently large field strength.)

Inversion

If the bridge is balanced so as to have zero potential difference and so zero current across the middle at zero applied field, you don't have inversion. But if you balance it a little differently, you can have a potential difference at zero applied field, which shrinks to zero at the applied field corresponding to logic 1, then starts to grow again. Alternatively, you can bias the field using a permanent magnet rather than the resistors, giving you a nice zero derivative in the neighborhood of logic 1. Either way, you get inversion.

Fan-in

How can you connect multiple inputs to a single output? One fairly inconvenient way — analogous to what we do in TTL and the like — is to physically colocate multiple coils, driven by different inputs, to produce magnetoresistance in the same magnetoresistor.

Memory

Aside from the usual feedback approach with a couple of inverters eating each other's tails, you might be able to use high-hysteresis, low-coercivity materials for a sort of ferrite core memory, with a nondestructive readout. I'm not entirely sure how you'd erase it, though; perhaps you could saturate your magnetic storage core in either direction, but take the reading at a point where a permanent magnet interferes with the field, either constructively or destructively?

Topics

Mayonnaise

Kragen Javier Sitaker, 2019-03-19 (updated 2019-06-10) (10 minutes)

Basically, homemade minipimer mayonnaise is a fucking miracle. From 200 mℓ of oil (US$0.18), an egg (AR$35 for half a dozen, US$0.15 per egg), 20 mℓ of lemon juice (AR$100 for 500 mℓ, I think, so US$0.10) and a pinch of salt (say US$0.01), you get about 300 g of mayonnaise customized to taste. Not counting depreciation on the minipimer and the 40 kJ of electrical energy, that's 44¢, and the whole process takes under 4 minutes, including getting the ingredients out and washing the dishes. And the result keeps even without refrigeration, though beware of circumstances where it can get diluted with water, lowering the acidity and salinity to levels where bacteria and fungi can grow.

The recipe is, as explained in that Twisted Sister song, huevos con aceite y limón. In more detail:

Mayonnaise recipes

https://www.fifteenspatulas.com/homemade-mayonnaise/

- one egg
- .25 cups oil
- .5 tsp ground mustard
- .5 tsp salt 30 seconds add another cup of oil over 60-90 seconds
- 2 tbsp lemon juice

https://www.inspiredtaste.net/25943/homemade-mayonnaise-recipe00/#itr-recipe-25943

- one large egg
- 1 tbsp dijon mustard
- 1 tbsp red or white wine vinegar
- .25 tsp kosher salt
- 240 mℓ (1 cup) oil (grapeseed, safflower, or canola)
- 1 tsp lemon juice

add oil incrementally while blending

https://www.epicurious.com/recipes/food/views/mayonnaise-24108003

- one egg yolk
- ½ tsp dijon mustard
- ¾ cup oil
- 1 tsp wine or cider vinegar
- 1½ tsp lemon juice
- ¼ tsp white pepper

add oil incrementally while whisking; add the salt and pepper at the end

use room temperature eggs

http://dish.allrecipes.com/making-mayonnaise/

- two yolks from large eggs

- 2 tsp lemon juice or vinegar
- 1 cup oil (light olive, grapeseed, or canola)
- a pinch of salt

  add oil incrementally while whisking
  add the salt at the end
  use room temperature eggs
  don't use bottled lemon juice
  add more yolk on emulsification failure
  for aioli add a very finely minced clove of garlic
  maybe try tarragon (estragón)

http://www.cookingforengineers.com/recipe/43/Homemade-Mayoonnaise

- 2 large egg yolks
- 3 tbsp lemon juice
- ¼ tsp salt
- pinch of white pepper
- 1 cup oil

  whisk in oil drop by drop after whisking other components
  freeze unused egg whites in ice cube trays
  https://aseasyasapplepie.com/homemade-mayonnaise-30-seconds/

- 1 egg (cold)
- 200 g sunflower oil (cold)
- 2 tbsp vinegar or lemon juice
- ¼ tsp salt
- 1 tsp dijon mustard

  warns not to break the yolk (!)
  use an "immersion blender" and it'll be easy; hold it still 10 seconds to get the emulsification started
  https://cooking.nytimes.com/recipes/12459-mayonnaise

- 1 large egg yolk
- 2 tsp lemon juice
- 1 tsp dijon mustard
- ¼ tsp kosher salt
- 1 tsp cold water
- ¾ cup oil (safflower or canola)

  whisk slowly
  slowly dribble in the oil

https://www.jamieoliver.com/recipes/eggs-recipes/my-beautiful-mayo/

- 2 egg yolks
- 1 heaped tsp dijon mustard
- 500 mℓ mixed oils
- 1–2 tbsp white wine vinegar
- ½ lemon
- sea salt

  add oil slowly
  add vinegar after most of the oil

Notes from my experience

Since noting the above, I've made mayonnaise about 20 times. My experiences:

• With a 600-watt minipimer ("immersion blender") there is very little you can do to keep the mayonnaise from emulsifying properly in less than a minute; I managed to do it once by not having enough egg in there, so I added another egg.
• This is true regardless of how much or how little oil you are including or how warm or cold anything is.
• Alioli — aioli in French — is fucking amazing. Including a clove of raw garlic — without bothering to mince it, because the minipimer takes care of that — makes a magical gel of deliciousness that can be applied to any food.
• Lower-power minipimers are less reliable at emulsification and much slower, so if you're using one of those you may actually need to wait to add some of the oil. You still don't need to do the thing where you carefully add the oil in a tiny trickle to keep it from separating out, which apparently you do have to do if you're mixing it with a fork or a whisk. The longer time probably produces more wear and tear on the minipimer.
• A bit of raw onion in the mayonnaise instead of garlic produces a result very much like sour cream with onion.
• Both vinegar and bottled lemon juice are fine, but I like the flavor of the lemon juice a bit better.
• The mayonnaise thus made separates a bit over the course of a few days if not eaten first, forming a sort of hardened grease on the top and up the sides of the mixing bowl.
• I've been using bargain-basement sunflower/soy oil mix (AR$33 per 900 mℓ last time I bought it, although then US$1 was AR$41, and now US$1 is AR$45) and can't tell the difference from pure sunflower oil.

I'm using cut-off bottoms of Coke bottles as mixing bowls. These are somewhat annoying to actually get the mayonnaise out of, because of the five smallish projections around the bottom, but I suspect those help a bit with the mixing part, by virtue of allowing the minipimer to spin very close to the bottom without actually touching it. Miraculously, neither of the minipimers I've used for this so far have the geometry to cut up the bottle bottom.

Minipimers are also much easier to wash than whisks are; you stick the end of the minipimer into a bowl of water and turn it on. The emulsion nature of mayonnaise helps here, dispersing the oil in small micelles. The PET of the Coke bottle is somewhat less accommodating; although the oil doesn't soak into it the way it does with polyethylene and polypropylene, really getting it off the plastic requires copious detergent and hard work with a sponge. Fortunately, this is unnecessary if you just rinse the mixing bowl with water and then make more mayonnaise in it as soon as you run out.

So far I haven't gotten salmonella from my homemade mayonnaise. Wikipedia tells me that large salmonella outbreaks in mayonnaise are invariably associated with inadequate acidity, with pH of 5 or even 6, while pH in the 3.6 to 4.1 range prevents spoilage. Unfortunately I don't have litmus paper handy, so I don't know what the pH of my mayonnaise is, and I don't carefully measure the acid as I add it.

You can turn the mayonnaise into a super lazy egg salad by adding a couple of boiled eggs, and optionally black or white pepper and a slice of raw onion, and chopping for just two to ten seconds with the same minipimer before washing it. If you have the eggs boiled beforehand, this is an excellent antidote to the urge for expensive convenience food.

Failed mayonnaise rescue by defecation

A couple of times using an underpowered minipimer I've failed to get the mayonnaise to mayon at first. In each case, after failing to solve the problem by adding more egg yolks, I let it sit covered in the fridge for an hour or two, and a substantial amount of bright yellow oil separated at the top; after pouring this off, it was relatively easy to get the less-oily remainder to gel, and it could then assimilate the oil that had been poured off. I suspect that what's happening is that the egg is being reduced to separated droplets floating in the oil, and further stirring is not effective at getting them to join up, though perhaps it makes them smaller, and certainly it keeps them from settling.

This process of separating immiscible liquids, or a liquid and a solid, by allowing the heavier one to settle out, is known as "decantation" or "defecation". Do not attempt to instead separate the mayonnaise after digesting it.

Presumably the aqueous phase of a successful mayonnaise is continuous, and that is the difference; I don't know if its oil phase is discontinuous, separated into droplets, or if the two phases form interpenetrating open-cell foams. The aqueous phase of the failed mayonnaises is definitely discontinuous, though, because tiny droplets of it remain suspended in the oil after it separates.

Is it healthy?

Aside from the concerns about salmonella, mayonnaise has a lot of calories; it's more than 70% pure fat. Eating fat is probably good for you (the late-20th-century conventional wisdom to the contrary has been shown to be false) but eating calorically dense foods is not. So it should be used as an ingredient in a more diverse food, such as a salad, not eaten by itself. Commercial mayonnaise is unappetizing enough that you have to be Michael Hart (peace be upon his blessed soul) to eat it by itself, but this stuff is tasty enough that I'm at risk of just eating it with a spoon.

If you're eating a low-carbohydrate diet, perhaps a ketogenic diet, maybe you shouldn't eat commercial mayonnaise, because it invariably contains modified food starch as a thickener. But real mayonnaise, made from egg, oil, and either lemon juice or vinegar, contains only traces of carbohydrates; lemon juice is about 6% carbohydrates (according to the bottle I have here) and the mayonnaise is about 5% lemon juice, so each 100 g of mayonnaise might contain 300 mg of carbohydrates from the lemon juice.

Topics

Surrealist code

Kragen Javier Sitaker, 2016-10-11 (3 minutes)

It occurred to me that you could maybe produce random deep-sounding or epic-sounding surrealist things with a dictionary substitution, which is to say a code, applied to ordinary English sentences, respecting the same parts of speech. For example, take this ordinary sentence by Thomas Ptacek:

If I were in charge of Russia, I would not want to get caught hacking the US.

Tag it by parts of speech and inflection:

If(conj) I(pron) were(v, 1st person singular subjunctive) in(conj) charge(n, plural or mass) of(prep) Russia(proper noun), I(pron) would(modal v taking infinitive) not(adv) want(v, inf) to(prep) get(aux v taking past participle) caught(v, past participle) hacking(v, present participle) the(article) US(noun).

Now, likely, you can make a substitution of other words that are the same part of speech while leaving it nearly grammatical. Some of the words — the ones in closed classes like conjunctions, prepositions, and modal verbs, and the ones that determine the inflection of other verbs — might be best to leave unchanged. For the ones you substitute, you could have a boring choice:

If I occurred in things of English, I could maybe take to have tagged making a sentence.

But maybe if you pick words with great emotional resonance instead, you could end up with something that sounds poetic, surrealist, or deep instead:

If I exalted in stone of suffering, I should deeply have danced flying the blood.

There are lists of "emotionally powerful words", "trigger words", "power words", or "high emotion words" going around for two-bit hustlers to spice up their sales pitches with. If I pick some words at random from one of these lists, I get this instead:

If I banned in insiders of Eva, I would stoically get satisfied controlling the miracle.

Another one quotes Churchill approvingly, "We have before us an ordeal of a most grievous kind." If I use the uncommon words from the Churchill quote, I get this:

If I struggled in ordeals of God, I would not wage to get surpassed suffering the tyranny.

The list of "317 power words" underneath gives results that are not as evocative, perhaps because the author is a two-bit clickbait hustler and not Winston Churchill, producing this instead:

If I collapsed in meltdowns of IRS, I would not worry to get cautioned smashing the mistake.

If I instead pick an arbitrary page of the Silmarillion for my words, I get this:

If I lay in lands of Beleg, I would not stay to get spoken asking the tree.

Topics

Notes on 3-D printing a mechanical LUT

Kragen Javier Sitaker, 2014-04-24 (3 minutes)

I tried 3D-printing a mechanical LUT on Julian Cerruti's Prusa Mendel the other day. The idea is that you position a probe in X and Y to set the input, then drop it down to see how far down it can fall before hitting the LUT to get the output.

It didn't go well, for several reasons.

First, I was trying to print at high resolution, both horizontally and vertically.

Vertically: my model had a 1/32 solid part at the bottom, and rose to a max height of 15/64 above that; these units were scaled to √2 centimeters, so the minimum thickness was √2 cm/32 = 0.044 cm = 0.44 mm, which worked okay (the first layer thickness was set to 0.35 mm, and subsequent layers 0.2 mm); the total thickness then would have been 17 √2 cm/32 = 3.76 mm. But note that 3.76mm is only 18 layers of plastic, which is close enough to the nominal LUT's desired resolution of 16 thicknesses that there could be aliasing problems. It would be better to double the thickness and align it precisely with the layers: two layers of solid, plus two layers per LUT level, for a total of 34 layers: (+ .15 (* 34 .2)) = 6.95 mm.

Horizontally: across the √2 cm width of the model, I had 15 intervals between grid lines, or 0.94 mm per interval. While the Mendel is perfectly capable of positioning the print head with precision well below a millimeter, the extruded plastic is substantially wider. Measurement of the infill lines on the bottom surface shows them to be spaced about 1mm to 1.5mm apart. Corner radius seems to be about 0.5mm. This means you can't practically make a positive feature with a total width below 1mm, with the software stack Julian's using, anyway (slic3r and pronterface, I think).

I think it's possible to achieve this with 1mm to 1.5mm horizontal spacing. It should be possible to make *holes* much narrower than 1mm, as long as they don't have to be too close together.

This brings me to the second problem: discontinuous layers. FDM machines typically print in horizontal slices to avoid crashing the print head into already-printed stuff, but they have various kinds of trouble when those horizontal slices contain discontinuities. At a minimum, they pull threads of plastic between the different places they visit; but also, the process of switching the pinch wheel between forward and reverse seems to be fairly undependable. So it might be desirable to add material *in between* grid points so that each layer is a single continuous piece. If the object is still a heightfield, this continuous-piece constraint simplifies to "no local maxima".

A third potential problem — which didn't show up in this case, but did in previous prints — is horizontal surfaces over infill. This requires the infill to be nice and solid to support the surface well; otherwise it can end up with holes in it.

Topics

Editor buffers

Kragen Javier Sitaker, 2015-07-15 (updated 2015-09-03) (16 minutes)

A text editor buffer is a potentially very large mutable string which you may want to make persistent, in the sense of storing to disk. One engineering problem, a less serious one than it once was, is how to implement this data structure efficiently, given that it needs to efficiently support arbitrary insertion and deletion, string search near the cursor, reading the region near the cursor, and moving the cursor to previous (or arbitrary) positions.

Text editor buffers also often contain associated metadata, including "markers" that you can efficiently jump the cursor to, which stay with the text they are attached to even when there are insertions and deletions elsewhere in the buffer. This means it is not sufficient to merely store an integer offset into the buffer as a marker; you must, at least, update that offset when there are insertions or deletions before it. There may be a pretty large number of markers, like, one per 16 bytes or so. You probably also need to be able to find the markers in a region of the buffer, like, for the purpose of storing line numbers. (Emacs markers only offer a subset of this functionality, because other primitives offer the other parts of it.)

This is also a reasonable description of the functionality a filesystem must provide, except that filesystems often do not support insertion and deletion in the middle of files efficiently. Nearly all, however, support appending to the end of files.

You could imagine a software system that elided these distinctions to some degree — where all your data was, conceptually, in a single sequence, into which you could insert data at any location, and which you could search through with impunity. Maybe you could even make it more efficient than current systems.

A byte array

The simplest data structure here is simply a contiguous byte array, reallocated when growing is needed. This used to be unusably slow, but now CPUs can `memmove()` 5 gigabytes per second, so now you're doing OK up to about 100 megabytes without doing anything special — you can `memmove()` 100 megabytes after every keystroke with barely detectable latency.

Gap buffers

A refinement of the raw byte array approach is the "gap buffer" approach, used by Emacs, where you leave a gap in the middle of the array where the cursor is. Moving the cursor (or, rather, inserting or deleting text in a new location) involves moving the gap, or rather moving text from one side of the gap to the other, but inserting or deleting text at the gap generally just involves adjusting the gap size. This means you only have to `memmove()` the 100 megabytes after every thousandth keystroke or whatever. (At the moment, my Emacs buffer gap is 1737 bytes.) This is quite fast, and it's from the at least 1970s (and may date back to Expensive Typewriter), although in the 1970s it was 80 kilobytes instead of 100 megabytes that you could move without the user getting annoyed.

As usual with things that are amortized cheap but occasionally expensive, you can make gap movement incremental at the cost of more memory usage and complexity. You need to keep track of a set of edits pending integration into the buffer (because the gap hasn't reached them yet), move the gap incrementally towards those edits (during idle time and during each keystroke), and include special cases in anything that *reads* the buffer to check the edits-pending-integration. Nobody does this, because instead they break things up into smaller pieces:

A single level of indirection

Supposing that your 100 megabytes or whatever are not enough (I have 16 gigabytes of RAM on my laptop, and plenty of files bigger than that). The next thing to try is to divide the buffer into a linear sequence of pieces, managed with, say, an array of pointers, or an array of structs. (For example, each piece's struct might contain a start pointer, a gap start offset, a gap end offset, and a total length. On a 64-bit machine, it would be reasonable for the first to be 8 bytes and the others to be 4 bytes, for a total of 24 bytes with padding.)

This is basically what the STL deque class does. It's also the approach taken by Finseth's editor Mince.

How many pieces to use? The general rule for this kind of thing is that you can reduce things that take O(N) work to O(√N) by dividing them into O(√N) pieces, so that the indirection table is about the same size as the pieces. In this case, if you have, say, a 128-gibibyte buffer, you could divide it into 2-mebibyte pieces, and there will be 65536 of them, and so your array of piece descriptors will take up about 1.5 mebibytes. This means that moving the buffer gap in any one of the pieces will take at most (2 mebibytes / 5 gigabytes per second) ≈ 0.4ms; moving a piece will take the same; and splitting a piece additionally involves inserting into the middle of that 65536-item array, which could take up to another 0.3ms or so. So the worst case is 0.7ms, and the average case (assuming the gap moves randomly) is half of that.

This is more or less the optimum piece size for this size of buffer. If you use smaller pieces, like 1 mebibyte, then moving the gap in a piece or splitting a piece will take less time, but updating the indirection array will take more time — a total of 0.8ms worst-case in that case. If you use larger pieces, like 4 mebibytes, updating the indirection array will take less time, but splitting the piece or moving it will take more time — a total of about 1ms worst-case in this case. Both of these numbers are worse than 0.7ms, but the threshold where this starts to matter to interactive response is where it becomes a significant fraction of a 17ms screen refresh.

Probably splitting pieces is much less common than moving the buffer gap a long way, so you might want to use somewhat smaller pieces in the interest of improving average-case response at the expense of worst-case response.

You can, of course, divide smaller buffers into smaller pieces.

If you're dealing with on-disk data instead of in-RAM data, you have to deal with both lower data rates and high latency. Reading or writing 2 mebibytes of data on spinning rust takes about 35ms, and seeking to it takes another 8ms or so, so the piece-splitting operation mentioned above that takes 0.7ms in RAM might take 90ms on

spinning rust. Worse, some disks are bigger than 128 gibibytes.

To deal with that kind of horror without nosing up into human-detectable operation times, we need to do better than O(√N). We need O(log N), and for that we need multiple levels of indirection:

B-trees, or multiple levels of indirection

B-trees are usually explained as an N-ary version of balanced binary search trees, a kind of ordered dictionary that's more amenable to sequential access than red-black trees and whatnot. But you can also use them for things that are just plain sequences. Each child pointer carries with it the total length of the sequence below it, so that you can index through the child pointers of a node until you find the pointer that contains the position you're interested in.

How many levels do you need? For spinning-rust usage, you probably want your B-tree nodes to be around half a mebibyte (so that you don't waste most of your time waiting for the head to seek to them); if your pointers and sizes are 40 bits, then a node holds about 48000 pointers, for about 23.4 gibibytes of child nodes, or 1.17 tebibytes of grandchild nodes. So basically with *two* levels of indirection you can get anything in two seeks and insert anywhere in three seeks and a mebibyte and a half.

For in-RAM usage, like for a traditional editor buffer, you probably want a lower branching factor and correspondingly deeper tree. For example, if we take our node size to be nominally 512 bytes and pointers to be 8 bytes, then our nodes have 64-way branching, or 32-way if they include the child weights. 128 gibibytes of buffer is 256 mebi-leafnodes, so you have up to five levels of branching, and about 1/31 of your RAM is taken up by internal nodes.

The leafnodes could still, if you wish, use buffer gaps. But it's no longer essential. Copying up to 512 bytes after every keystroke (and updating up to six ancestor weights) is not very expensive.

For more usual buffer sizes, you need less levels of indirection. 8 megabytes is only 16 kibi-leafnodes, so 528 internal nodes in a three-level tree suffices.

Compressed RAM

With the increasing gap between RAM speed and CPU speed, and the advent of new very-high-speed compression algorithms like LZ4, it's becoming reasonable to compress in-RAM data that isn't immediately being used; it's often faster to spend the CPU to decompress it than it is to take the locality hit of copying it from main memory.

Ropes

You'll note in the discussion about on-disk data structures that there was little discussion of locality of reference for writing, no discussion of fragmentation, and no discussion of crash recovery. We blithely assumed that updating in place presented no problems. Similarly, in the discussion of updating ancestor weights in RAM, there was no discussion of concurrency. And undo, of course, was taken for granted.

In practice, of course, these problems are ubiquitous. One way to

solve them (at the cost of unpredictable memory usage) is to *never modify data in place*: instead, always write new copies of the data, and eventually garbage-collect the old data once it's no longer used. This makes your data structures "persistent" in the functional-programming sense — you can reference their old state simply by keeping a reference to it, which makes undo quite simple — and allows you to choose where to write the updates to, which helps with fragmentation and write bandwidth. This is the approach behind the NetApp WAFL, behind Ousterhout's Berkeley LFS, and also behind the "rope" data structure from Xerox PARC's Cedar, which made its way into the SGI STL in a reference-counted form.

If you strictly follow these rules for B-trees, every byte you insert or delete involves creating a whole new set of treenodes up to the root. In our 128-gibibyte-buffer-with-six-levels-of-treenodes case, this is 3 kibibytes of data — plenty fast enough for keystroke response, but 3072 times bigger than the keystroke being written. It's going to give your garbage collector a headache, even though it won't trip any write barriers. There are a variety of different ways of solving this, including zippers.

The standard rope approach to this is to use unbalanced binary trees rather than B-trees, so that you only need a couple of small treenodes gluing together the parts of your buffer you aren't editing with the new string you're inserting. This way you need, say, 40 bytes of freshly allocated data to record each new keystroke, instead of 3072.

One of the simplest approaches, though, is to batch up a pending update until it's big enough to be worth it.

Update logs

In transaction processing, this is sometimes called a "side file": you log the inserts, updates, and deletes in a small file "to the side of" your main database. Any query must take care to ensure that its results take the changes in the side file into account, which typically involves traversing the side file sequentially; or you can maintain an updated version of the relevant part of the database in non-immutable memory.

This is more or less the approach that WAFL originally took to keep its disk write bandwidth manageable: it would buffer changes (in battery-backed RAM) until the once-per-second timer fired and it wrote a snapshot to disk.

In the case of a text editor, it would seem that a gap-buffer representation for the particular part of the buffer you're currently modifying should enable modification to proceed with very high efficiency; once the modifications are complete and you move elsewhere, they could be recorded in new B-tree nodes.

Markers

All of the above, however, disregards the problem of markers. If you have 128 gibibytes of text in your editor buffer, you might have 8 gibi-markers. It's no good if adding a character to your 512-byte leafnode is efficient if that requires you to increment 4 billion markers!

It's not obvious how to solve this problem scalably. But there is a way.

If you take the Mince approach, an array of separate 2-mebibyte gap-buffer pieces, maybe you could associate a separate marker table with each piece, containing the marker's physical position in that piece. This requires up to 65536 lookups to find a marker, and updating up to 131072 marks when the gap moves. (You'd probably want to doubly-index the marker table so that you can usually update fewer.) These sound like ridiculous numbers, but they're probably in the same ballpark as the cost of moving the gap normally.

In a mutable B-tree, though, there's a much more interesting possibility. Suppose a mark stores the addresses of all six of the nodes on its path down from the root, plus the byte offset into the leaf node. A mutation to nodes on that path might alter the offset within the node at which the pointer to the child is found, but it won't alter the value of that child pointer, so the mark remains valid. Only if the leaf node is mutated, or if a node it traverses is split so that the child is no longer present, will the mark need updating. And we could index the marks by these addresses so that we can find the marks that need updating when we're splitting a node.

We could go further, though. Suppose every node has a parent pointer. Now, the mark can simply consist of a pointer to the leaf node that it points into and an offset into that leaf node, and that leaf node can have a list of marks associated with it. No mutations higher in the tree will require updating the mark; updating the leaf node may require updating the mark, but finding it is easy.

This is THE SOLUTION to the scalable editor buffer problem.

Topics

Passively safe solar hot water

Kragen Javier Sitaker, 2012-10-15 (updated 2012-10-16) (6 minutes)

Suppose we want a solar hot-water-heating panel to never heat above 49 degrees under normal circumstances, so that the water it heats never becomes dangerously hot. "Normal circumstances" might involve a temperature of, say, 35 degrees outdoors. One way to do this would be to have a thermal resistance between the panel and the environment that allows a heat flux equivalent to the absorbed sunlight once there's a 14-kelvin difference: given, say, 800 watts per square meter, you need a U-factor of 800 watts/14 kelvin square meters, or an R value of 14 kelvin m²/800 W, or R = 0.018 K m² / W; that's about equivalent to an inch of concrete. However, typical outdoor air film values are around R = 0.03 K m² / W --- on each side, so 0.015 on both sides together.

If that wasn't enough thermal conductance, you could reduce the sunlight by angling the panels or painting them gray or shading them or whatever so that they only collect 400 W/m² or something, and then you should be fine. But there's another problem, which is that that 14-kelvin difference above ambient is still a 14-kelvin difference above ambient when ambient drops to 0 degrees, which will give you 14-degree "hot" water.

That's not really acceptable! To have a passively safe design that still provides adequate heat in winter, you need some kind of more consistent heat sink than the air. Maybe radiating to the sky could work: you use some insulation to keep from losing heat to the nearby air, and radiate your extra heat as infrared into deep space.

For this, you would have enough insulation to permit the difference from ambient air to be up to 50 degrees or so (R = 0.06 K m² / W, which should be achievable with nothing more than some trapped air spaces) while permitting infrared radiation to get to space, in particular around the ten-micron wavelength. Transparent polyethylene film trapping air spaces will apparently work for this (see patent http://www.google.com/patents/US5493126), but it photodegrades rapidly enough that it would need replacement every few months. Acrylic (plexiglass) transmits near infrared; I think it transmits thermal infrared (LWIR) as well, but I haven't tested. FT-IR spectra on the web for polyethylene terephthalate http://deepblue.lib.umich.edu/handle/2027.42/32476 suggest that it, too, transmits LWIR well, and it can survive solar ultraviolet for a long time.

Anyway, then, you just need to angle the panel so that it will be sufficiently coupled to the part of the sky that doesn't have the sun in it to shed the heat it acquires from the sun.

A nice thing about this kind of radiative cooling is that the power transmitted is proportional to the fourth power of the temperature. So if you're transmitting 800 W/m² at 49 degrees (322 K), at 43 degrees (316 K) you'll be transmitting 740 W/m², and at 30 degrees (303 K) you'll only be transmitting 627 W/m² --- you'll have a whole 173 W/m² pushing you toward your target temperature. Unfortunately this is still not enough to be very robust against efficiency losses in picking up the heat. arccos(627/800) is about 38

degrees (of arc!), and arccos(740/800) is only 22 degrees --- so whenever the sun is further than 22 degrees from at right angles to the panel, it won't be able to heat the water above 43 degrees C. Which means, at best, 3 hours a day.

But that's for a flat panel! If the collector isn't flat (for example, if it consists of multiple flat panels at different angles), it will absorb heat from the sun at a more consistent rate throughout the day. That solves the intermittency angle problem. (The requirement that the thermal emission equal the sunlight at the maximum safe temperature can be met, as before, by adjusting the angle of attack to the sunlight.)

The big problem with that approach is its low efficiency. Heating from 0 degrees to 43 degrees, you start off with about 48% efficiency (1-(273/322)^4) but end up with 7% efficiency (1-(316/322)^4). That means you end up with panels covering many times more area, for safety, than you would need simply to gather the appropriate amount of energy, simply because the vast majority of the energy gathered is re-radiated immediately.

There might be a way to avoid this problem: creating a sufficiently selective surface. Blackbody radiation has a fairly sharp cutoff at its top frequency, and absorption bands in plastics also have fairly sharp cutoffs. It might be possible to put together a blend of transparent plastics that block essentially all radiation longer than, say, 8000 nanometers, but have one or two orders of magnitude more transmissivity for shorter wavelengths. A black surface covered with such plastics would have an emissivity that jumped sharply (from, say, 0.001 up to 0.1 or 0.2) upon reaching a target temperature. A sufficiently large radiator of such a surface, backed by sufficiently good heat transport, would maintain a narrow range of temperatures over a wide range of heat flows.

Topics

Window systems

Kragen Javier Sitaker, 2018-10-26 (1 minute)

I have several notes on window systems:

Cached SOA desktop (p. 2229) is about a REST architecture applied to a windowing system, with each 16×16 tile being a separate resource, using a cache invalidation protocol for redraws.

Speculative plans for BubbleOS (p. 2128) talks briefly about Wercam, a secure windowing system.

What does a futuristic OS look like? (p. 2163) talks about what I think a "cool" or "futuristic" OS should look like, or rather what "outdated" OSes look like.

A minimal window system (p. 1545) talks about how to make a windowing system use a minimal amount of code, considering different design alternatives.

Pixel stream (p. 617) talks about a protocol for windowing systems.

Real time windowing (p. 891) asks about how to make windowing systems guaranteed responsive.

Topics

- Graphics (p. 3483) (91 notes)
- Graphical user interfaces (p. 3489) (23 notes)

Three-stack generic macro assembler (design sketch)

Kragen Javier Sitaker, 2019-04-30 (8 minutes)

IBNIZ is viznut's bytebeat and display-hack virtual machine; it uses a stack-based instruction set, and when it's in run mode, it's constantly executing the program as you edit it. The program runs repeatedly on the same stack, and whatever it leaves on the stack is interpreted as pixels to paint the screen contents. (Audio works similarly.)

I was just reading Longo's *Introduction to DECSystem-20 Assembly Programming* and thinking about how the DEC assemblers and the assembler described in Nova RDOS (p. 1724) didn't have even the machine's fundamental opcodes defined internally; they were defined as manifest constants and macros, but basically almost all the assembler did by default was put numbers into memory and build a symbol table, which always seemed like a really cool idea to me. (And it was fundamental to how Gates, Allen, and Davidoff wrote Microsoft BASIC, by incrementally hacking the DEC machine into emulating an 8080.)

The first example program in Longo begins as follows:

```
EXAMPLE 1, ADDING TWO NUMBERS
I:    3    ;I=3
K:    2    ;K=2
J:    0    ;save a spot called J
```

This assembles three numbers into memory and sticks three entries into the symbol table.

Now, symbol tables are awesome and really flexible, but it would be even simpler if we could do without them, at least at the fundamental level. And some constructs — notably nestable conditionals and loops — are somewhat awkward to write in macro assemblers using labels. I mean, it's not that the labels stop you, but they don't help.

It occurred to me that maybe a stack-based assembler that filled the object file the way IBNIZ fills the screen could work.

How it would work

If the assembler interprets the input `3 2 0`, it would first push 3 on the stack, then 2, then 0. If that's the whole input, those stack contents are, in some encoding, the contents of the binary file it will write out. Similarly, if the input contains `"ABC"`, some representation of that string gets pushed onto the stack — maybe 65 66 67 3, the UTF-8 values of the string and then its length, for example. But then you could define "macros", little functions that would pop things off the stack, change the state of the interpreter, and push more things back onto the stack. For example, you might have a `MOV` macro that would push the encoding of a MOV instruction onto the stack, popping off an appropriate number of operands.

(Such an assembler designed for AMD64 machines would need to

work in bytes, while a SPARC assembler might be easier if it worked in 32-bit words. You could in theory do the conversion at the end of the process.)

Advantages; the landscape

Nestable control structures — the whole raison d'etre of this goofy idea — would use a third stack for their jump addresses, in addition to the operand stack that's accumulating the output file and the macro execution stack that keeps track of which macros are invoking which. And adding some peephole optimization would be pretty trivial, although you'd have to be careful with variable-length instruction encodings, and you might want to have some kind of low water mark that prevented such rewriting, since you wouldn't want to invalidate a jump destination. (Your macros could still inspect the previously emitted code, though.)

You'd still have a symbol table, of course, so you can call named functions. It would just be built and maintained by macros.

Although you'd probably want to use some kind of RPN syntax, there's no reason the macros would have to be programmed at a FORTH-like level, except perhaps as a bootstrappability exercise. On the contrary, the compile-time macro data world can be garbage-collected, dynamically-dispatched, pattern-matching, backtracking, theorem proving, SAT solving, whatever is most convenient for writing bits of compiler macros. PostScript and Factor are not the limit.

If you gave the compiler macro language the ability to read from its own input stream — the way FORTH immediate words do in order to read the name of a word to define, for example, or the way PostScript programs do to incrementally read and render bitmap data — you could convert your macro assembler into a compiler for arbitrary languages by providing different preludes. It might not be a good idea, but you could do it.

What's the difference from emitting binary from whatever random high-level language? The smooth incremental path from manually typing in (or hexdumping) some binary data to refactoring parts of it to enable you to generate more of it.

Self-bootstrapping

Maybe also the smooth incremental path of building more capable assemblers. This might be a way to do something like StoneKnifeForth that gradually adds features from one stage of the language to the next, without ever requiring a backwards-incompatible language leap. The most basic bootstrap compiler could handle nothing more than converting binary to decimal, while valid input to it could remain valid for all later versions. Here's a somewhat small, though surely not minimal, implementation of such a basic bootstrap. This is for AMD64 Linux, and it doesn't check its input for validity.

```
## Convert space-separated decimal numbers to binary

## You know, I think this is almost the same as the i386
## system call interface, but maybe with different registers?
## Also, different system call numbers!  On i386 %eax=1 is
```

```
        ## _exit().

        .globl _start
        ## First, read a byte:
_start: xor %edi, %edi          # fd 0
        mov $buf, %esi          # into buf
        xor %edx, %edx
        inc %edx                # 1 byte
        xor %eax, %eax          # _NR_read is 0
        syscall

        test %eax, %eax         # quit if error or EOF
        jle done

        ## Only if we've got a space, write out the number:
        mov (buf), %eax
        cmp $32, %eax
        jne keep_converting

        ## We got a space, so we should write out our byte:
        mov %ebx, (buf)         # store byte into buf
        xor %edi, %edi
        inc %edi                # fd 1
        mov $buf, %esi          # write from buf
        mov %edi, %edx          # 1 byte
        mov %edi, %eax          # _NR_write is 1
        syscall

        ## Then repeat:
        xor %ebx, %ebx          # clear accumulating number
        jmp _start

keep_converting:
        imul $10, %ebx
        mov (buf), %eax
        and $15, %eax
        add %eax, %ebx
        jmp _start

        ## Got EOF or error, so exit.
done:   xor %edx, %edx
        mov $231, %eax          # _NR_exit_group in unistd_64.h
        syscall

        .data
        .align 8
buf:    .byte 0
        .byte 0
        .byte 0
        .byte 0

        .byte 0
        .byte 0
        .byte 0
        .byte 0
```

This can reproduce its own 119 bytes of machine code (well, of an earlier version of itself) given this input:

```
echo 49 255 190 96 1 96 0 49 210 255 194 49 192 15 \
5 133 192 126 55 139 4 37 96 1 96 0 131 248 32 117 26 49 255 255 199 137 \
28 37 96 1 96 0 190 96 1 96 0 137 250 137 248 15 5 49 219 235 199 107 219 \
10 139 4 37 96 1 96 0 131 224 15 1 195 235 182 49 210 184 231 0 0 0 15 \
5' '|./binout > binout.bin
```

Reproducing a valid ELF executable presumably requires a few dozen more numbers, hopefully not hundreds.

So once you have that working, you can patch in a couple more features and fix up some jump offsets.

Topics

Arcadian plastics

Kragen Javier Sitaker, 2019-11-19 (3 minutes)

The thought that we
must eliminate single-use disposable goods
must eliminate plastics
proceeds not from environmentalism but from Arcadianism
humans for a hundred thousand generations had to cling to dead things
good linen, stone houses, stone tools, bone needles, bronze, steel, brass, gold
cling to a scrap of cloth like a grandmother, for it too cares for you through the winter
this Arcadian longing for those lost generations is anti-life
life is not tasteful
life is not restrained
life is not frugal
life is not conservative
life is not lasting
those are not virtues for life
those are virtues for human capital accumulation

life is exuberant
life is a wild, ever-changing fire upon the surface of the earth
life is algae blooming in a red tide
life is ceaseless but never constant
life is the merciless
life is the struggle of a thousand salmon that leap frantically past the bears
before laying five thousand eggs, destroying their viscera with steroids
life is a sequoia seed awaiting its awakening by fire
to sprout in the ashes
life is eager dandelions growing brightly in the cracks of the sidewalk
life is desperate air plants drilling for moisture in the rain forest bark
life is a thousand-year-old ginkgo slowly consumed by a fungus in its heartwood
protecting its delicious nuts with the stench of ten thousand corpses
life is unstable, exciting, impermanent, and full of death
life is refulgent purple jacarandá flowers covering you in brilliant yellow pollen
life is a peaceful marsh where a crocodile is resting just under the water

when a mushroom is fabricated overnight in the forest
more intricate than the carvings of any sculptor
more beautiful than the brushstrokes of any painter
it is made of a single-use disposable biopolymer
and is gone inside a week

trees regret not shedding leaves made of single-use disposable biopolymer
wolves ferociously devour single-use rabbits made of biopolymers and leave disposable droppings

humans had to cling to dead scraps of cloth and care for them like

newborn babies
 could not change raiment with the seasons as the snow hare or the goldfinch do
 because a spool of yarn was a day's work
    a century ago a tarnished mirror was a sign of wealth
 today they lie discarded in the gutters
    those who know the secrets of life and the nature of matter
 can use its own materials for their products
 they are no longer limited to heavy, fragile clay, rusting iron, and dead wood
 but can use the diaphanous resins we call plastics
 to carry water or adorn their children
 they can evaporate aluminum onto glass that floated on tin
 their children can race carbon-fiber bicycles
    a cellophane wrapper, a sterile catheter, a condom
 betoken the liberation of the humans from the toil of a hundred thousand generations
 and pitiless disease that killed one out of every five children before adulthood

Interval raymarching

Kragen Javier Sitaker, 2019-11-02 (updated 2019-11-10) (6 minutes)

In talking with banyaszvonat, the following idea arose: I think it's possible to speed up raymarching with signed distance fields enormously using affine arithmetic and possibly even interval arithmetic.

Raymarching with signed distance fields

Raymarching with signed distance fields is a currently popular computer graphics technique (in particular Inigo Quilez has achieved visually amazing results, and perhaps as a result, the technique is widely used throughout the demoscene) and is also the basis of Christopher Olah's ImplicitCAD solid-modeling system. The "signed distance field" or "SDF" of an object is a function from points in 3-D space to the Euclidean distance to the nearest point on the object's surface, except that it's positive outside the object and negative inside. Since it's the distance from a point to the *nearest* point on the object, it's a *lower bound* on the distance from that point to *any* point on the object.

Constructing such a function is a very convenient and expressive way to model a wide variety of geometry.

Raymarching successively approximates the intersection between a ray and the object by advancing a point along the ray toward the object. The SDF enables us to use a variant of raymarching called "sphere tracing" in which the distance to advance is given by by the SDF at that point. This is guaranteed not to overshoot the intersection, since the SDF gives a lower bound to the distance to that intersection (as to any other point on the object), but if the intersection point is not the nearest point on the object — which it almost never is --- the point will only move closer to the surface, not reach it. If the intersection is in a smooth part of the surface, each new point will be closer to the surface by a factor of $1/\sin(\theta)$, where θ is the angle the ray makes with the surface normal.

This is known as "linear convergence" and results in iteration counts in the dozens to hundreds per ray in typical scenes.

Note that the above algorithm has no trouble with the case where the SDF is actually an *overestimate* of the true surface distance, but may fail if the SDF is an underestimate. Conservative-estimate SDFs are extremely useful for a variety of purposes.

Interval and affine arithmetic along the ray

It occurred to me that interval and affine arithmetic could perhaps speed up the process immensely.

The reason we can't just leap down the ray an arbitrarily long distance and see if we're inside the object is that, if it's not thick enough, we might leap right through it and out the other side, eventually hitting some other part of the object or just the sky. But if we use interval arithmetic to evaluate the SDF over an interval of the ray, rather than at a point, we can check to see whether that interval includes zero — that is, whether it is possible for the ray to intersect

the surface anywhere inside this interval. This allows us to test points on the other side of the surface without the risk of missing intersections, which should allow us to reliably linearly interpolate to find the zero of the SDF, effectively using the method of secants with its φ degree of convergence rather than the linear degree of convergence given by sphere tracing.

(Jorge Eliecer Florez Diaz wrote his dissertation, which I still haven't managed to finish reading, on using interval arithmetic to handle this case correctly for raytracing of implicit functions. I'm sure what's above is in there.)

By using affine arithmetic or reduced affine arithmetic instead of simple interval arithmetic, we get an affine form which gives the SDF value over the chosen interval of the ray as a linear function of the position along the ray plus an error bound. This allows us to find all possible zeroes of the SDF reliably and probably much more quickly than with just interval arithmetic, because the subinterval or subintervals in which the SDF *might* have a zero will often be very small compared to the original interval.

Interval and affine arithmetic on screen coordinates

Using interval values for the distance along the ray, as described above, can be combined with using interval values for the position and angle of the ray as well (a so-called "view frustum"), permitting in many cases the computation of an affine expression for the color of a part of the screen. See Reduced affine arithmetic raytracer (p. 2007) for more on this.

Gradients and surface normals

Since surface normals are needed as input to lighting calculations, we need some way to calculate them from the SDF; they are the gradient of the SDF at the surface. This is normally approximated by sampling the SDF in octahedra or tetrahedra near the intersection point, choosing the epsilon size comparable to the projected pixel size to attenuate aliasing of surface bumps with pixel sampling. But I think affine arithmetic inherently calculates an approximation of the gradient, which may be adequately precise for these purposes. Moreover, unlike automatic differentiation, the coefficients in the affine form potentially pertain to an entire interval, not just a point.

Related work that I probably should have read by now

In addition to Florez Diaz's dissertation, https://www.shadertoy.com/view/lssSWH is relevant, and maybe "Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry", Duff 1992.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)

Archival transparencies

Kragen Javier Sitaker, 2014-06-05 (updated 2014-06-29) (7 minutes)

I've written before a bit about "transparency holograms" and how they might be useful as an archival medium for text. Here I want to do some calculations on that.

Let's suppose we're printing out on both sides of a 4-mil (102µm) Mylar transparency foil, using the simple grille pattern where all but one pixel is blacked out in each N×N square; I'll take N to be 7 in this case, and assume 600dpi, so the grille has a hole spacing of 7"/600 = 296µm, or 85.7 dpi. The other side of the foil has the pixels of 49 different 85.7 dpi pages interleaved. (For the moment I'll assume this is for a viewing distance of infinity.)

If we use a 6-pixel-tall font for text, which works out to 5.04 points at 85.7 dpi, each of the 49 pages has space for 157 rather long lines of text, about 208 characters each if we assume an average of 3.5 pixels. (I don't remember if this is the right average from bible-columns.png.) That's about 1.6 megabytes of text on the transparency, and it should all be visible to the naked eye, hopefully.

Do the viewing angles work out? The angle between two pages needs to not be impracticably small, and the angle for the most extreme left, right, top, or bottom page needs to not be impracticably large. Mylar's index of refraction is about 1.6, so those 101.6 microns are, optically speaking, closer to 63.5 microns. But our 600dpi page has 42.3µm pixel spacing, which means that we have to go 25° to the left, right, up, or down to see the next pixel in that direction: 20 cm to the side at a viewing distance of 50 cm. The refraction linearizes this somewhat, so the next page is closer to being 20 cm further to the side than to being 25° further; about 60cm to the side gets you to the third page. So we can do 7×7 without extremely distorted viewing angles—barely! At the extreme corners of viewing angle, the page is foreshortened by 36% in both X and Y dimensions.

So at 600dpi on heavy-duty transparency film, it's barely doable; at 1200dpi it's eminently doable, with more like 6 megabytes of text per page; at 300dpi you would need thicker transparency film.

14-mil (356µm) and 7-mil (178µm) Mylar sheets are sold as art supplies for making stencils, pre-frosted on one or both sides, even. This would work even with a 300dpi printer, but necessarily with a wider grille spacing. "Matte Dura-Lar" is one brand name for double-frosted Mylar. "Drafting film" is another way this kind of thing is sold. I suspect that clear film will work better, though. Grafix, the manufacturer of Dura-Lar, claims that neither inkjet nor laser will print on Dura-Lar, but other people claim success with inkjet, while claiming that the laser-printer heat curls up Mylar.

A related idea: shrink film, like Shrinky-Dinks, might enhance the resolution of a printer at the same time as thickening the film; it shrinks 2:1 when heated, which should quadruple its thickness, initially 254µm. But I don't know if it's available in transparent, and it's made of polystyrene, which is apparently not archival-safe if exposed to ultraviolet or, maybe, oxygen. However, it seems that polystyrene's better-knn problem is that it's so stable that it can last a long time after being discarded, resisting biodegradation and

photodegradation for up to centuries. (Obviously you can't put shrink film through a laser printer.)

Mechanically, though, polystyrene is far, far more fragile than PET. PET stick-on films are used to make windows smash-resistant and even mildly bullet-resistant.

At the other end of the spectrum, polycarbonate ("bulletproof glass") can be recovered from CDs, and is, ounce for ounce, less fragile than almost anything, although it has a weakness for ammonia and other alkalis. Melting CDs also causes them to shrink, just like polystyrene shrink film; but they're optically clear. It's also an archival-stable plastic.

A test

I snarfed rosettaproject_zul_gen-1.pdf (SHA1 296b27e22a720a8d3c10fa971d262f798b27906d) from https://archive.org/details/rosettaproject_zul_gen-1 and converted it first to PS with pdf2ps, then to 100dpi PNGs. GhostScript by default just does point sampling rather than antialiasing, which is shitty and loses parts of letters entirely when rendering bitmaps at low resolution, so I antialiased by brute force with ImageMagick:

```
time gs -dNOPAUSE -r300 \
    -sOutputFile=rosettaproject_zul_gen-1-large-p%d.png \
    -sDEVICE=pnggray -dBATCH rosettaproject_zul_gen-1.ps
time for x in rosettaproject_zul_gen-1-large-p*; do
    mogrify -scale 827x1169! -threshold 90% "$x"
done
```

827×1169 is from `units A4paper` telling me A4 is 210mm by 297mm, and that those work out to be 827 and 1169 pixels respectively at 100dpi. The `!` is to indicate that the images are to be stretched and squeezed as necessary to make them fit, because by default ImageMagick preserves aspect ratios and doesn't fit the box. The `-threshold` option is to convert the image to bilevel, while dilating the letterforms somewhat to keep any lines from getting lost.

Anyway, then I use the `interleave.py` I just wrote using PIL; unfortunately this is only 5 pages, so to get to 3×3, I had to duplicate some.

```
time ./interleave.py \
    rosettaproject_zul_gen-1-large-p1.png\
    rosettaproject_zul_gen-1-large-p2.png\
    rosettaproject_zul_gen-1-large-p3.png\
    \
    rosettaproject_zul_gen-1-large-p4.png\
    rosettaproject_zul_gen-1-large-p5.png\
    rosettaproject_zul_gen-1-large-p5.png\
    \
    rosettaproject_zul_gen-1-large-p5.png\
    rosettaproject_zul_gen-1-large-p5.png\
    rosettaproject_zul_gen-1-large-p5.png
mv interleave-output.png bible-zulu-interleaved.png
```

`interleave.py` took like 3½ minutes to do this, and the result looks

okay.

(Now I guess I need to turn that into a PDF so I can print it, plus printing the grille.)

My theory on the numbers on this prototype is that, even if the number of pages changes, the size of the composite pixels shouldn't; there should still be about 86 of them per inch in each direction. This image has 100 of them per inch in each direction, and only 9 pages; this should be less challenging in the dimensions of legibility, viewing angle delta, brightness, and maximum viewing angle.

A thing that surprised me about this, but shouldn't have, is that some of the text is readable without the grille. In particular, since page 5 is repeated 5 times, it's quite readable despite the other four page images interfering.

A second test

I downloaded Murray R. Spiegel's "Mathematical Handbook of Formulas and Tables" from https://archive.org/details/MathematicalHandbookOfFormulasAndTables (SHA1 efc83d86ae251e4ef7bf6ec852caeb95708ba98d) and extracted pp. 27, 28, 30, 31, 37, 38, 39, 40, and 43 with Evince's "print; range; pages" to a PDF file. Then I processed this into PNGs:

```
time gs -dNOPAUSE -r300 -sOutputFile=spiegel-select-p%d.png \
    -sDEVICE=pnggray -dBATCH ~/Documents/spiegel-select.pdf
time for x in spiegel-select-p*; do
    mogrify -scale 827x1169! -threshold 70% "$x"
done
```

The 90% threshold came out shitty because this book contains gray halftoning with super important black text in it, but a 50% threshold washes out the text badly. 70% was better than either 66% or 75%, although it still has some problems.

```
time ./interleave.py spiegel-select-p*
mv interleave-output.png spiegel-interleaved.png
```

It seems to have worked pretty well.

Topics

How small can we make a comfortable subset of JS?

Kragen Javier Sitaker, 2018-11-27 (updated 2018-12-02) (3 minutes)

I just hacked together some crappy JS. I thought it would be good to write down a grammar for the subset of JS I happened to use, as a sort of benchmark of how big a grammar for a usable JSish language needs to be. Here I'm using lowercase for tokens.

```
Stmt ::= Expr semicolon | Decl | Function | Block | Controlflow
Controlflow ::= If | For | Forin | Forof | continue semicolon | Ret | Throw
Ret ::= return semicolon | return Expr semicolon
Throw ::= throw Expr
Expr ::= Assexpr
Assexpr ::= Orexpr eq Assexpr | Orexpr
Orexpr ::= Orexpr or Andexpr | Andexpr
Andexpr ::= Andexpr and Cmpexpr | Cmpexpr
Cmpexpr ::= Cmpexpr Cmpop Addexpr | Addexpr
Cmpop ::= eqeqeq | eqeq | leq | geq | lt | gt | noteqeq | noteq | in
Addexpr ::= Addexpr (plus | minus) Mulexpr | Mulexpr
Mulexpr ::= Mulexpr (times | divide | modulo) Incexpr | Incexpr
Incexpr ::= not Incexpr | Atom | Atom plusplus | Atom minusminus

Atom ::= Literal | name | Listdisplay | Methodcall | Property | leftparen Expr ri
ghtparen
Literal ::= string | null | true | false | regexp
Listdisplay ::= leftbracket Exprlist rightbracket
Exprlist ::= Expr comma Exprlist | Expr
Methodcall ::= Property leftparen Exprlist rightparen
Property ::= Atom period name | Atom leftbracket Expr rightbracket
Decl ::= (let | const) Decls
Decls ::= name eq Expr | name | name eq Expr comma Decls | name comma Decls
Function ::= function name leftparen (Params | ε) rightparen
Params ::= name | name comma Params
Block ::= leftcurly Stmts rightcurly
Stmts ::= Stmt Stmts | ε
If ::= if leftparen Expr rightparen Stmt (else Stmt | ε)
For ::= for leftparen Forinit semicolon Fortest semicolon Forinc rightparen Stmt
Forinit ::= Decl | Expr | ε
Fortest ::= Expr | ε
Forinc ::= Expr | ε
Forin ::= for leftparen (let | ε) name in Expr rightparen Stmt
Forof ::= for leftparen (let | ε) name of Expr rightparen Stmt
```

Noticeably missing are function expressions, object displays, try-catch statements, `break`, `new`, arrow functions, unary prefix operations + and - (same precedence as `!`), … spread operations, bitwise operators, and semicolon insertion. That's because I happened not to use them in the code in question.

The tokens are something like the following:

```
semicolon ::= ';'
```

```
continue ::= 'continue'
return ::= 'return'
throw ::= 'throw'
eq ::= '='
or ::= '||'
and ::= '&&'
eqeqeq ::= '==='
eqeq ::= '=='
leq ::= '<='
geq ::= '>='
lt ::= '<'
gt ::= '>'
noteqeq ::= '!=='
noteq ::= '!='
in ::= 'in'
plus ::= '+'
minus ::= '-'
times ::= '*'
divide ::= '/'
modulo ::= '%'
not ::= '!'
plusplus ::= '++'
minusminus ::= '--'
leftparen ::= '('
rightparen ::= ')'
string ::= '"' ([^"\\] | '\\' char)* '"' | "'" ([^'\\] | '\\' char)* "'"
null ::= 'null'
true ::= 'true'
false ::= 'false'
regexp ::= '/' ([^/\\] | '\\' char)* '/'
leftbracket ::= '['
rightbracket ::= ']'
comma ::= ','
period ::= '.'
let ::= 'let'
const ::= 'const'
name ::= [A-Za-z_\u0080-\u10ffff] [A-Za-z0-9\u0080-\u10ffff]*
function ::= 'function'
leftcurly ::= '{'
rightcurly ::= '}'
if ::= 'if'
else ::= 'else'
for ::= 'for'
of ::= 'of'
```

This omits whitespace, comments, and the complications of automatic semicolon insertion.

The tricky part for PEG implementation would be the left-recursion in Orexpr, Andexpr, Cmpexpr, Addexpr, and Mulexpr; the comma ambiguity (once you add the comma operator) probably requires distinguishing Exprs from Commalessexprs, and the if-else ambiguity is handled reasonably by PEGs.

Topics

Current hardware trends tend toward raytracing

Kragen Javier Sitaker, 2016-10-07 (4 minutes)

From at least the 1970s until the present day, semiconductor RAM has been the dominant form of memory measured in number of memory accesses, and the number of bytes of memory on an interactive computer has roughly equaled the number of instructions per second it could execute, from about a hundred thousand around 1980, to about a million a few years later, to about ten million in the early 1990s, to about a hundred million in the late 1990s, to about a billion in the early 2000s, to about ten billion today.

But in theory time-space tradeoffs are possible, and indeed we have been achieving reasonable performance on these machines by storing a lot of precomputed results in their memory.

But current microcontrollers have dramatically more computational power than they have onboard memory, and adding off-chip memory dramatically increases power usage. It's entirely typical to find a CPU capable of 50 or 100 million 32-bit integer instructions per second, but limited to 16 or 32 kilobytes of SRAM, a ratio of about 2000 instructions per byte.

The extreme on this axis, in a sense, is GreenArrays; each of the 144 cores on a GreenArrays chip executes about a billion 18-bit integer instructions per second, but only has 64 18-bit words of memory, which would be 144 bytes. That's about 7 million instructions per byte. I say "in a sense" because, with that little memory, it's really kind of in between being an FPGA and being a regular computer.

If nowadays CPU cycles are cheap compared to memory, how should we adapt our software to that? Maybe we can do more stuff with convolutional and recurrent neural networks, but how about regular things like graphical user interfaces?

You could imagine raytracing your GUI. Raytracers use almost no memory for intermediate rendering results, although they do usually build up a scene graph. I just ran a quick raytracer I'd written to benchmark, generating 1920×1080 pixels, and got these results:

```
==22338== Cachegrind, a cache and branch-prediction profiler
==22338== Copyright (C) 2002-2011, and GNU GPL'd, by Nicholas Nethercote et al.
==22338== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==22338== Command: ./raytracer 1920 1080
==22338==
--22338-- warning: L3 cache found, using its data for the LL simulation.
==22338==
==22338== I   refs:      1,348,631,580
==22338== I1  misses:            1,263
==22338== LLi misses:            1,230
==22338== I1  miss rate:          0.00%
==22338== LLi miss rate:          0.00%
==22338==
==22338== D   refs:        459,287,791  (331,081,906 rd   + 128,205,885 wr)
```

```
==22338== D1  misses:          21,360  (     19,098 rd   +       2,262 wr)
==22338== LLd misses:           2,292  (      1,719 rd   +         573 wr)
==22338== D1  miss rate:          0.0% (        0.0%     +         0.0%  )
==22338== LLd miss rate:          0.0% (        0.0%     +         0.0%  )
==22338==
==22338== LL refs:             22,623  (     20,361 rd   +       2,262 wr)
==22338== LL misses:            3,522  (      2,949 rd   +         573 wr)
==22338== LL miss rate:           0.0% (        0.0%     +         0.0%  )
```

So that took 1.348 billion instructions, about 650 instructions per output pixel. It missed in its level-1 data cache (which is 128 KiB) 21,360 times; its cache lines are 64 bytes, so that totals about 1.37 megabytes of data I/O, about 10% of which was output.

Suppose this 650 instructions per pixel is typical (although I was raytracing a very simple scene, you could imagine getting close to this performance on general scenes using all kinds of clever hacks). Then you could paint a 320×240 screen in 50 million instructions; with ten or twenty small processors you could raytrace a small-screen UI.

However, 2560×1440 tablets are becoming common now! These would require 2.4 billion instructions per frame at that rate.

So, maybe not for a while yet. The trends seem clear.

Topics

Sample reversal

Kragen Javier Sitaker, 2018-12-18 (updated 2019-01-17) (5 minutes)

One problem that can happen with sampled sounds is a discontinuity at the end of the sample. If the sample is long enough, this sounds like a click, but a short sample (e.g. a single oscillation of a waveform) can turn it into a sawtooth buzz, one that can overwhelm the rest of the signal.

One tool available in old MOD trackers was time-reversal of the sample, so instead of playing the wavetable from beginning to end repeatedly, it would play boustrophedonically, first from beginning to end, then backwards from end to beginning, and then repeat. This approach eliminates the discontinuity. It also cuts the size of your wavetable in half.

Audacity has a "zero-crossing finding" tool on the Z key which adjusts your selection boundaries to the nearest positive-going zero crossing. This also helps with the same problem.

The boustrophedon approach has an interesting effect on the signal, though. Since the resulting waveform is reflection-symmetric, it can only contain cosine waves; the phase of all the signal components is forced to zero, or rather all imaginary components are canceled, because the time-reversed part of the signal has conjugated phase. (All harmonics of the period of the doubled-length wavetable are possible, though.) So the extent to which a partial makes it through this procedure is dependent on its phase — if it happens to have perfectly real phase, it's untouched, but if it happens to have perfectly imaginary phase, it's entirely canceled. This means that filtering a signal with an allpass filter before this procedure can radically change the frequency content on the output, even though the signal going in may sound identical.

I propose a probably known extension to the procedure: rather than repeat(wavetable[1:] || reverse(wavetable)[1:]), use repeat(wavetable[1:] || reverse(wavetable)[1:] || 2·wavetable[0] - (wavetable[1:] || reverse(wavetable)[1:])), reversing the sample both in time and in sign, eliminating the discontinuity in the derivative at the beginning of the wavetable (though not its end) and extending the generated waveform to four times the length of the wavetable. This allows the generated waveform to contain all of its period's harmonics with any phase, rather than just real phases, and it cuts in half again the number of samples needed to represent a waveform of a given fundamental frequency.

For example, a 440-Hz A note at a sampling rate of 16kHz (a common rate in MODs) would require a wavetable of 36.3636 samples for a full oscillation; a 36-sample version of this wave would be out of tune by 1%, which sounds inconsequential but is 17 cents, easily audible. So you'd probably use three oscillations, 109 samples, which gives you 440.37 Hz, only off by 1.4 cents. If you try to use this approach with ¾ of an oscillation, it will work, but you're constrained to 27 or 28 samples, corresponding to 108 or 112 samples in the full three-period time, which would be 36 or 37⅓ samples per oscillation.

I was thinking that you could choose to duplicate the initial/final

sample of a segment, or not, to correct this detuning, but I think that probably a better way to inexpensively tune up or down by fractions of a semitone is to occasionally duplicate or omit samples at points where the sample value and, especially, derivative are zero or nearly so. I don't know how this will sound, but maybe if it works well enough, you could represent an arbitrary 440Hz harmonic sound with a 12-sample 16ksps wavetable. This is very appealing not just because it takes 24 bytes but because optimizing over a 12-dimensional space, whether in the frequency domain or the time domain, whether by hand or using an algorithm, seems much more tractable than optimizing over a 109-dimensional space.

For images, you can use this kind of approach to make a smoothly tiling texture out of any image by reflecting it at the border. A kaleidoscope works this way; the tiling need not be square but it does need to have only vertices of even degree, which probably limits you to variants of square and triangular tilings.

You can also use this approach to make smoothly stretching display window frames from a single example: by inserting reflected sections into the middle of stretched dimensions as necessary, you can make the window whatever size you want, down to the minimum of the original drawing. You won't introduce discontinuities in colors, but you may introduce discontinuities in line angles; if you can choose preferred direction-reversal points where the lines are all either perpendicular or parallel to the edge, you can keep that to a minimum, but you may not be able to eliminate it entirely.

Topics

Notes on higher-order programming on the JVM

Kragen Javier Sitaker, 2016-09-06 (6 minutes)

How do I dynamically generate bytecode?

Aside from understanding the JVM bytecode (for which `javap -c` and chapter 6 of the JVM spec are helpful), there's the question of how to get from a plan for code to generate to actually being able to run the code on the JVM, as described in chapter 5 of the JVM spec.

At the most basic level, you write a custom ClassLoader, which is five or six lines of code, and invoke `.loadClass(name)` on it. But then you still have to generate the bytecode that will define your class.

(There's an existing overview of bytecode-generation and -manipulation libraries for Java at java-source.net.)

That's what ObjectWeb ASM does; it's a library for JVM assembly programming, and it comes with a disassembler, which produces Java source code that invokes the ObjectWeb ASM APIs. It's used by CGLib, Hibernate, Clojure, Jython, JRuby, and so on. It may be actively maintained (it supports Java 8) but change is mostly limited to bugfixes at this point. It has a reputation for being simpler to use than BCEL, because it's more narrowly focused on generating, transforming, and analyzing byte arrays representing JVM-bytecode classes. Generating a hello-world class with it is 14 lines of code in the documentation, and generating a simple method as a sequence of bytecode ops is another six.

Javassist is an actively maintained 16-year-old free-software library for Java bytecode manipulation, with what seems to be a better-thought-out and more convenient interface, although it's hard to find reasonable documentation (Javadoc doesn't count!). It's integrated with its own Java compiler, so you can even specify bytecode to insert in the form of Java source code! (However, its Java compiler supports a subset of the full language.) It's part of the JBoss project now, and apparently uses ObjectWeb ASM.

Soot is an actively maintained framework for analyzing and optimizing Java bytecode, supporting different intermediate representations (each of which has a textual syntax). It also supports Android bytecode (which I assume means Dalvik), which makes it unique among the libraries I've looked at. It's mostly oriented toward program analysis (e.g. interprocedural dataflow analysis) rather than dynamic code generation, but you can also use it for dynamic code generation; the hello-world Soot dynamic class requires 28 lines of code to generate. Despite being actively maintained, it doesn't yet support Java 8.

BCEL, previously known as JavaClass, is a library for generating and transforming JVM bytecode,. It comes with, among other things, a Java disassembler to Jasmin syntax. It's at a very similar level to ObjectWeb ASM, but it looks somewhat more cumbersome to use, although there are some very cool facilities in it; its most basic example, called HelloWorldBuilder, is over 100 lines, although that includes things like try-catch blocks. BCEL seems to have been

abandoned in 2006.

SERP is another actively maintained library for, mostly, bytecode modification, apparently with a stateful DOM-like API. It's almost completely undocumented.

How do I profile?

The crudest tool is a thread stack dump, which you can get by typing control-backslash or by using `jstack` (included with the JVM) with the appropriate PID. This will show you the stack of each thread in your JVM process, from which you can see what it's currently taking too long to do.

There's also a profiler that comes with the JVM called HPROF; you invoke it for CPU-sampling profiling with `java -agentlib:hprof=cpu=samples YourProgram` or with entry-and-exit instrumentation, which slows the program down by an order of magnitude but gets accurate call counts, with `-agentlib:hprof=cpu=times`.

Profiler4j, open-source, abandoned in 2006, "in beta stage", based on bytecode instrumentation.

Some old profilers use the JVMPI interface, which has been phased out to be replaced with JVMTI in current Java.

The Netbeans folks wrote a profiler which is now this separate thing called VisualVM, which has now expanded far beyond just profiling. VisualVM is free software but apparently hasn't been ported along with OpenJDK, although there seems to be no licensing reason not to; maybe you can compile it from source, and it supports OpenJDK.

JIP is a profiler that hooks the classloader and uses ObjectWeb ASM to instrument bytecode as it's loaded. It's supposedly much lower overhead than hprof. But it's been abandoned since like 2008.

TPTP was a profiler integrated into Eclipse, built on JVMTI. It was abandoned in 2011.

Jvmtop monitors all the JVMs on a machine, like top, and includes a high-overhead sampling-based CPU console profiler. Jvmtop as a whole was abandoned in 2013.

GCViewer is a currently maintained viewer of GC statistics, using the GC profiling functionality built into the JVM, to help you understand GC behavior with pretty graphs and extensive statistics. Jörg Wüthrich took over maintenance from tagtraum industries in 2008.

Topics

• Programming (p. 3658) (286 notes)
• Java (p. 3531) (5 notes)

Can a simple nonlinear VCO enable super cheap oscilloscopes?

Kragen Javier Sitaker, 2017-05-04 (updated 2017-05-10) (5 minutes)

I want to build an oscilloscope out of recycled e-waste. This is problematic because parts like analog oscilloscope CRTs and 60Msps ADCs are not common in e-waste, and parts like TV CRTs and 2Msps ADCs are not adequate substitutes. A 100kHz "oscilloscope" is a joke. 20MHz is the minimum standard, and for an analog scope, that's just the 3dB cutoff frequency; it can still detect signals of higher frequencies, just attenuated and maybe phase-shifted.

Once I have a digitized signal, basically the hard part is over. Everything after that is a simple matter of programming. Digitizing the signal fast enough is the hard part.

To be concrete, I want to be able to sample a signal to the following specifications:

- at a rate of at least 60 million samples per second;
- at a resolution of at least 8 bits per sample;
- for at least 2048 data points;
- for at least two data channels.

(Ideally I could do this many times per second, but I don't think that's going to be a meaningful constraint.)

So I've been exploring a variety of different possibilities, and I think I might finally have a design approach that will work.

To digitize a channel, you first use it to control eight separate nonlinear VCOs with one-octave ranges and center frequencies distributed around 350MHz. At their lowest frequencies, they will be around 250MHz; at their highest, around 500MHz. The linearity of their voltage- frequency response matters almost not at all for this application; deviations from linearity of a factor of 2 or more can be tolerated. This should make it possible to use very simple, primitive designs for the VCOs; something like the 5¢ MMBT6428 npn RF transistor, with its 700MHz transition frequency, ought to be adequate for each VCO.

The spread in frequencies among the VCOs prevents them from locking to one another through unintentional cross-coupling. This probably means they'll need to be spread apart by more than 15MHz (thus 120MHz between the highest and the lowest), which I will disregard in what follows, hoping it's unimportant.

So now you have eight channels carrying waveforms, each of which is in the 250MHz to 500MHz range, and which collectively have between 4 and 8 billion transitions per second. This means that during a 17-nanosecond sample period (at 60Msps), they have between 65 and 133 transitions, depending on the input voltage and on how the phase of each oscillator happens to line up with the sample period. I think you should expect rms noise of .82 counts per sample period from the random phase thing, compared to rms noise of .29 counts for just quantizing at all (a factor of $\sqrt{8}$ smaller), which means ENOB is $\lg(66*.29/.82) \approx 4.5$, which is inadequate, even before some of those bits are spent on tolerating nonlinearity.

(I think the above calculation is still wrong. See https://en.wikipedia.org/wiki/Effective_number_of_bits.)

Then you need to count these pulses. If each channel is connected through a MOSFET to a capacitor and a 500MHz ripple counter, we can freeze the count by turning off the MOSFET and letting the ripple finish, and then we can latch it by reading out the ripple count data in parallel. You just need the capacitor's RC time constant with the MOSFET to be well under 1ns, and its discharge time while driving the ripple counter to be long enough to give us time to read the count (say, 1µs).

A 500MHz ripple counter has a first bit that transitions at 500MHz, a second bit that transitions at 250MHz, a third bit at 125MHz, a fourth bit at 62½MHz, and then we're down to ranges where any old ripple counter will do. But the first four bits of the counter might need to be built with special care.

This requires us to transfer eight 8-bit transition counts (64 bits) from the ripple counters to FIFOs every 17-nanosecond sample period, probably not over a shared 8-bit bus, in order to keep the bus clocks down at 60MHz instead of 480MHz. The FIFOs need to hold 2048 data points (128 kibibits in total, 16 kibibytes), but don't need to be dual-ported. Then we can read them at our leisure when we're not acquiring data. For example, if we trigger a data collection 60 times a second, we have 17 milliseconds to read the data out of the FIFOs, about a microsecond per data transaction or 125 nanoseconds per data transaction on a given FIFO.

It's probably necessary to measure the temperatures of the VCOs in order to compensate for thermal effects on their voltage-frequency relationship.

Using larger ripple counters is probably a good idea because it means you can get higher precision at lower sampling rates.

Topics

A language whose memory model is a bunch of temporally-indexed logs

Kragen Javier Sitaker, 2019-05-12 (updated 2018-05-21) (20 minutes)

The Mill CPU uses a randomly-readable queue, called the "belt", instead of stacks or registers; the last *n* instruction outputs (I think *n* = 16?) are addressable as inputs. This makes the instruction set encoding denser, due to the lack of need for output register fields in the instructions. It also makes superscalar scheduling simpler, because inter-instruction dependencies are simpler to detect.

Normal RAM is made of registers; imperative programming languages expose this in the language semantics, indeed centering it. A register is a function from time to values, computed from a sequence of writes at different times; at any time *t* its value is the value written by the last write at time before *t*. What if, instead of a RAM made of registers, we had a RAM made of belts, each storing the entire sequence of writes to that location? Indeed, what if it were made of infinite belts?

Similar language features

I was debugging some Octave† code today. In Octave, an assignment statement like `x = y + z` generates a line of output by default saying something like `x = 5.4405`. To suppress this, you have to append a semicolon, which is only legal on assignment statements. So if you write a piece of Octave code without any semicolons and then run it, it outputs a history of all the values assumed by variables during its execution, interleaved by time, a trace which is very valuable when debugging. If only one variable is thus semicolonless, the trace shows the history of its values. By examining the overall behavior of the trace, you can often understand things about your program's behavior that are difficult to see from its state at a single point in time.

Of course you can do something analogous in almost any programming language with `print` statements, and this "printf debugging" or "logging" is a common tactic, superior to stepping in a debugger under many circumstances; my friend Charity is doing a very influential startup right now called honeycomb.io based on elaborations of printf debugging. But in Octave, logging every new value assumed by every variable is a default. You have to turn it off explicitly. (Unfortunately, you do that by editing the code.)

Another thing I've wanted to do frequently with Octave when writing numerical algorithms that compute a series of x_i values is to change the code between returning a vector x of all the values or just returning the last value x_n. Typically one direction of this transformation amounts to replacing `x =` with `x(end+1) =`, and `x` otherwise with `x(end)`.

The DDD debugger frontend to the GDB debugger also has facilities to capture values of specified variables every time a given breakpoint is hit; then you can plot or list the logged values.

Bret Victor has done a couple of experimental programming environment sketches which log each new value produced and annotate your source code with them, placing time on the horizontal axis — one aimed at defining and exploring recurrence relations, another intended to make imperative looping more understandable to new programmers.

Python and ES6 have a "generator" feature in which you can define lazily-computed sequences of values by writing an imperative function to `yield` them one by one; for example:

```
def lc_words(fileobj):
    for line in fileobj:
        for word in line.split():
            yield word.lower()

def pairs(seq):
    last = next(seq)
    for item in seq:
        yield last, item
        last = item
```

† Octave is a free-software clone of MATLAB.

A language with nondestructive assignment statements

What if assignment statements like `x = 3` still changed the value of `x`, but left its previous values accessible, say, as `x@1`, `x@2`, and so on? Then you could inspect all the previous values when you were trying to debug a program. Moreover, you wouldn't have to do anything special to enable my iterative-calculation Octave functions to return their whole list of values — though if the number of iterations wasn't known ahead of time, they'd have to return the length of the list, too.

Many DSP algorithms would become slightly simpler to write, since you could just write `x` when you wanted the latest value of x, and `x@1`, `x@2`, etc., when you wanted x_{i-1}, x_{i-2}, etc.

The downside is that refactoring would become somewhat trickier; `x = f(y); x = f(x)` would no longer be equivalent to `x = f(f(y))`, `x = 0; if (g(n)) x = n` would no longer be equivalent to `x = g(n) ? n : 0`, and functions would expose the entire value history of their return value, not just its final value, even if only the final value was intended to be used.

As described in Usability of scientific calculators (p. 2379), writing down numerical state machines in such a notation is very convenient.

It might be worthwhile to provide an `#x` operation to tell you the total number of values that have been written to x so far. This would allow you to note `#x`, call some function that appends things to x, and then subtract from the new `#x` when its execution finishes to see how many values it produced. I think this is probably pretty bug-prone, though, since it tempts you to write code that assumes that nothing has ever written to x previously, and puts you in danger of writing code that will break if `#x` has some unexpected value. Maybe if `#x` starts at 1'000'000 or something, that would ameliorate this problem. (Alternatively, you could set `#x` to 0 in the language model for local

variables `x`.)

In the framework of *Essentials of Programming Languages*, the *expressed values* of the language, those that expressions can evaluate to, would still be atomic values like numbers and characters, but its *denoted values* would be integer-indexed limitless logs of such atomic values.

There's no way for a subroutine in such a language to reduce its memory consumption; all previously produced values are available for inspection. So it probably needs to be short-lived.

Could we get by without any other kind of indexed memory addressing?

If the memory in our language's virtual machine consists not of registers but of limitless logs of past values, any single variable can then contain an entire array, and indeed a limitless number of arrays one after the other.

For the first ten or twenty years, say from 1945 to sometime between 1955 and 1965 in different lineages, the memory addresses to access (typically "registers of the store", at the time) were hardcoded into the instruction. This made it hard to write loops like this one:

```
int total = 0;
for (int i = 0; i < n; i++) total += x[i];
```

That's because the different items of `x[i]` are stored at different memory addresses, and the same addition instruction has to access all of them. So you had to modify the instruction. Typically you would add the index to it, since the source address field was typically stored in the least significant bits of the instruction, though this could obviously cause surprising effects if the arithmetic generated carries out of that field.

The solution to this problem was the "index register", and later also "base registers" and "pointer registers", which are just CPU registers whose values enter into the computation of the effective address. This allows you to compile code like the above without generating self-modifying code — you just store `i` in an index register, or maybe even `&x[0]` if the index register is wide enough. (C goes further and declares array indexing to be merely a matter of pointer arithmetic.)

Could we replace array and structure indexing entirely with this kind of indexing into the past? Maybe our instruction set could disallow indexing space and only allow indexing time.

It turns out that you can, and it doesn't really cost you performance except in the case of random indexed writes, for which it costs you a logarithmic slowdown. Programs are arguably slightly clearer than the same programs written using index registers, but not as clear as programs written using indexing and field access in modern programming languages.

Arrays

Well, if the memory consists not of registers but of infinite logs of past values, a single variable `x` can then contain the entire array:

```
int total = 0;
```

```
for (int i = 0; i < n; i++) total += x@i;
```

This has the potential disadvantage that, as in core LISP, reading happens in the opposite order from writing. For this case it doesn't matter, but when it does, you can write `x@(n-i)`, for Octave-style 1-based indexing, or `x@(n-i+1)` for C-style 0-based indexing.

If `n` is a constant, you can access item *i* of the *j*th previous value of *x* with `x@(n*j - i + 1)`.

On the other hand, if the past values of `n` are the lengths of past arrays stored in `x`, then we can index the *j*th of them with *O(j)* work; the length of the previous `x` array is `n@1`, and its latest value is `x@n`. The one before that ends at `x@(n+n-1)` and is of length `n@2`, the one before that is `x@(n+n@1+n@2)` and is of length `n@3`, and so on.

If we want random access to the past `x` arrays, we can store the prefix sum of their lengths instead of just the lengths themselves. If the prefix sums are stored in `s`, then the length of the latest `x` array is `s-s@1`, the previous one is `s@1-s@2`, and the one before that is `s@2-s@3` (and its last element is `x@(s-s@2)`).

This storage scheme sort of prevents you from appending elements incrementally to the latest `x` array, which is maybe undesirable; in that case you can store its length in a variable `n` that you can increment, and only store the cumulative sum of the *completed* array lengths in `s`. So then you have `n+s-s@(j-1)` elements after the end of the *j* th-from-last array, for $j > 0$.

If no typing discipline prevents it, you could get by with writing the lengths and their cumulative sums and whatnot to the same variable as the array values themselves, but that gets awkward to program.

These schemes generalize with little difficulty to multidimensional arrays; if you want to regard the *n* last items of *x* as a two-dimensional $w \times h$ array, such as a grayscale image, you can index item (i, j) as, for example, `x@(w*i + j)`.

Statically-typed immutable heap records

Consider structures less regular than arrays, though — binary search trees, for example. You might have a single type of tree node with a key field and nullable left and right pointers. You can store this as parallel arrays `K`, `L`, and `R`, and a count `n`. (Parallel arrays really suck if you're deleting objects, but fortunately that's something we don't have to worry about here.) `L` and `R` can be integer indices. You can't mutate anything, but you can add new nodes, which is enough to allow you to create a new updated root for a new tree state.

Here's a binary tree search using this representation:

```
next = root
while next != 0:
    node = next
    if needle == K@(n - node): break
    next = (needle < K@(n - node)) ? L@(n - node) : R@(n - node)
```

Generally, though, when we have a bunch of records floating around in the heap with links to each other, we have multiple different types of records (classes of windows, productions of syntax, etc.), so we want dynamic typing. Dynamic typing is a bit hairier.

Let's call the structure formed by `K`, `L`, `R`, and `n` a "table", and `K`, `L`, and `R` its "columns". For a sum type, we need one table per variant type, plus a sum-type table with a tag column (telling which table the real value is in) and a pointer column (giving its index in that table), or with one pointer column per variant type. Optionally, if there's some data in common among all variants, you can move that data into an extra column in the sum-type table, which gives you more or less the conventional RDBMS approach to inheritance subtyping if you continue it a bit further.

(Syntactic sugar for this "table" construct might be necessary to make this approach actually usable.)

Random writing to arrays

We've talked about reading from random indexes into arrays. But what about algorithms that want not to *read* at random indexes but to *write* at them? For example, suppose we want to draw a diagonal Bresenham line on the `x@(w*i + j)` grayscale image mentioned just above. There's no direct way to do this; we can only write to the end of `x`.

You could, though, stick all your lines and other graphical primitives in a buffer and use the scanline rendering rasterization algorithm to compute the scan lines of the image in order. This is potentially more difficult for hairier primitives like splines and IFS fractals.

However, we can totally construct a sparse matrix in COO format — a table with columns `row`, `column`, and `value`. You can append onto these things in whatever order you like. Converting a COO sparse matrix to a dense matrix is conventionally done in linear time with random writes, but you can also do it by sorting the items and then traversing the sorted list. This adds a logarithmic slowdown, but perhaps with a well-tuned sort, that would be tolerable.

(Technically radix sorts are linear-time in input size because your number of distinct sort keys can't approach infinity without their length also approaching infinity, but in the approximation where we have less than, say, 2^{64} different sort keys, their execution time is more nearly linearithmic, like comparison-based algorithms.)

Implementations in hardware or software

Programs written in this bizarre fashion have the pleasant feature that their memory writes are in some sense sequential, which — aside from whatever debugging benefits it may or may not provide — works well with WORM media (which are not very important at the moment, but may be in the future) and block-erased NAND Flash.

However, it's not really plausible that future computers will be without at least a few kilobytes of traditional register-style RAM, writing to which will be much cheaper than writing to WORM or Flash. So pushing the sequentiality of writing down to the atomic program operation level may not make sense. Also, if you want to keep the writes actually sequential in real life, you can't use a separate memory for each variable; you need some system that dynamically apportions blocks of data written to the sequential medium among the variables that are actually being written to, and adds enough indexing that you can index far into those variables' past quickly,

without using up too much space.

Binary instructions (whether executed by software or in hardware) accessing a memory made of belts could use a smaller number of bits to select a belt in RAM than normal instructions use to select a byte or word in it.

If you're implementing this model in software on today's hardware, with its supercharged multilevel SRAM caches and gigabytes of SDRAM, you probably should store the current value of each variable in a fixed memory location, allocating a page or so of buffer to log its previous values into; you may be able to take advantage of page faults, Electric-Fence-style, for overflow detection, and thus avoid the overhead of bounds-checking instructions on every write (at the expense of TLB pollution).

Garbage collection

Generally, as I said, garbage collection is impossible, but there may be cases where some memory can be reclaimed.

First, of course, if variables are local to a subroutine, they can be freed when the subroutine returns.

The following presumes that there's no Fortran-style array aliasing going on when we pass parameters to subroutines; otherwise, the problem becomes quite a bit hairier.

Some variables may never be indexed at all; if their previous values are saved at all, it's only for debugging purposes. Some variables may only be indexed by constants, for example, in DSP code, so their previous values can be saved in a small circular buffer.

Bounds on variable index expressions are more difficult. It may be deducible that a variable i is always in the range [0, n), or is never negative; this would allow us to deduce that `x@(n-i)` can never index further back than n. But for this to allow us to discard values of x older than n, we need to also be able to show that n cannot increase, or at least cannot increase any faster than x is extended.

Instead of rigorously proving such bounds correct, you could try using relatively conservative allocation sizes for different variables, and restarting the program from a checkpoint if it suffers a bounds violation, indicating that it was trying to access values that it had already forgotten. For code that generates the forgotten values in linear time — that is to say, at an asymptotically constant speed, rather than one that increases without bound — doing this with a multiplicative increase in the space for the bounds-violated variable is "only a constant factor" slowdown. But it's a constant factor that's a multiple of the number of such variables, if you just increase the size of the one. (Generating values at a speed that increases without bound is of course impossible; you can at most generate one value per instruction.)

Alternatively, spilling memory to disk could permit your program to generate a terabyte or more of past values before any must be discarded.

You could imagine some class of programs might be easier to automatically parallelize or strength-reduce in this memory model, since in a way what it supports natively are simple recurrence relations like those discussed in Notations for defining dynamical systems (p. 2872). The idea is that you write your program as an initial state and a reduction function that updates the initial state, either according to

an input datum or on its own; then the compiler turns some or all of the program into elementwise operations and parallel prefix-sums using some subset of the program it's proven associative so that the parallel prefix-sum algorithm applies. Thus the different reductions aren't really running sequentially.

Topics

Flux deposition for 3-D printing in glass and metals

Kragen Javier Sitaker, 2016-07-03 (15 minutes)

Suppose you do a powder-bed 3-D printing process where the binder you deposit is a flux that lowers the melting and sintering temperatures of the powder filler, then bake the block at a temperature high enough to sinter or melt the binder-modified part of the structure. This might make 3D printing in new, unusual, or very inexpensive materials feasible.

The baseline I'm coming from here is that a spool of fucking PLA on Amazon goes for fucking US$23/kg. And, yeah, PLA is light and dimensionally stable and doesn't require your hotend to get super hot (180° is enough), but it's also pretty weak (50 MPa), and it sure takes a long time to come out of that teeny little hole.

Soda-lime glass

There are many possible binder-filler systems that could be used, but the one that seems most promising to me at the moment would use quartz sand as the filler and a mixture of sodium and calcium carbonates as the binder, forming soda-lime glass. Soda-lime glass has a glass transition temperature of about 570°, and so sintering should be possible a bit below that, while unfluxed quartz melts at 1670°. With 1100° of headroom, it should be very easy to keep the kiln temperature in the range required to sinter or melt only the fluxed part, not the unused filler. Indeed, it might be possible to pit-fire the piece, which typically reaches heats between 1000° and 1200°.

The usual mixture for soda-lime glass is about 73% silica, 13% sodium oxide, 10% calcium oxide, and 4% other impurities. Sodium and calcium carbonates, which are themselves fairly safe to handle, decompose to form these caustic oxides at 851° and 840°, respectively, though I suspect the presence of quartz will lower the required temperature. Sodium bicarbonate (aka sodium hydrogen carbonate), which is even safer to handle, decomposes to sodium carbonate starting at 50°, releasing water and CO_2. All of these ingredients are safe enough that they are used in cooking.

If calcium is omitted, the resulting "waterglass" material is water-soluble at high temperatures; this may not be a concern, depending on the application. The potential advantage of this method would be that sodium carbonate and bicarbonate are water-soluble, up to about 5% or 10% in warm water. However, that isn't enough to properly flux the quartz, so it will have to be applied in solid form.

You could have a robot with a valved funnel carefully dribbling the appropriate amount of mixed carbonates onto each layer of silica sand, or you could have an entire row of such nozzles.

Soda-lime glass doesn't start to flow freely until about 1000°, so as long as the temperature doesn't get that high, the fluxed part of the print won't soak into the rest of the unused filler too badly.

Lead glass might be an alternative: rather than calcium, you use lead(II) oxide (or, with air, galena or just plain lead), and rather than sodium oxide, you use potassium oxide, or in practice, potassium

carbonate. Lead glass, however, is less viscous than soda-lime glass, which is undesirable in this case, and doesn't have a lower softening point (still around 600°).

Feldspar?

The only mineral more common (and thus, I hope, cheaper) than quartz is feldspar, or rather, the feldspars, which melt at a wide variety of temperatures, from 600° to 1200°. I'm not sure what, if anything, to flux them with, other than quartz itself. I guess calcium-rich plagioclase can be fluxed with sodium-rich plagioclase.

Sodium and potassium feldspars are traditionally used as fluxes in pottery to vitrify silica and boron trioxide, so in a sense this is just a slightly different angle on the soda-lime glass from the previous section.

Alumina ceramic

Alumina (aluminum oxide, also known as ruby or sapphire) is the hardest mineral commonly used in ceramics, because the harder diamond and carborundum (silicon carbide) are tricky to deal with, cubic boron nitride is expensive (I think?), and tungsten carbide is about as hard as alumina, but more expensive.

Alumina doesn't melt until the truly unreasonably hot 2072°, which was a great difficulty in the development of the Hall-Héroult process that converted aluminum from a precious metal into a cheaper substitute for steel. The trick that made it feasible was fluxing the alumina in cryolite, Na_3AlF_6, so that the mix melts at only 1000°.

So you could imagine fluxing an alumina powder bed with just enough cryolite to get the grains to sinter together into a glass at around 1000°. You can probably do this with an arbitrarily small amount of cryolite; it melts at 1012°, and I believe it will wet the alumina grains immediately and begin to dissolve them, recrystallizing or vitrifying upon cooling. So the question is merely how much cryolite is needed to wet the alumina grains enough to form a solid mass.

I don't know how hard or strong the resulting cryolite-cemented alumina aggregate will be.

Lead-tin solder and type metal

Lead melts at 327°; tin melts at 232°; but 63% lead and 37% tin melts at 183°. So you could flux lead filings with tin filings, and then heat the piece to anywhere between 183° (or even a bit less) and 327°. Lead costs about $2.20/kg; tin costs about $22/kg. So the mixture costs about US$10/kg, which is not outrageous, but not cheap either. (You could probably reduce the price further by reducing tin content, at the cost of a higher and less crisp melting point.)

However, lead is very dense (11.3g/cc), so a kilogram of this metal is not very much, and the tin-lead alloy is very soft; you can nick it with your fingernails. Tin itself, at 7.4g/cc, is considerably less dense.

Another problem with tin-lead solder is that it shrinks when it solidifies, resulting in a rough surface. Type metal, Gutenberg's great invention, is a variant with a significant quantity of antimony (US$6.60/kg, melts at 630°), which prevents this shrinkage and

improves hardness further. The eutectic alloy is supposedly 84% lead, 12% antimony, and 4% tin, which works out to US$3.50/kg; it melts around 241°. (The antimony-lead eutectic alloy melts at 252°.)

Type metal has a Brinell hardness of around 20, which is four times that of lead, but one sixth that of soft steel. This suggests that its tensile strength might be .36 * 9.8 * 20 = 70 MPa, a little better than PLA.

Another possible alternative is pewter, which is tin fluxed with about 1% copper (US$4.40/kg) and 5% antimony. This is considerably harder than tin because of the alloying elements, and I believe immune to tin pest, but it's even more expensive.

Brass and bronze

Bronze (copper, which melts at 1084°, alloyed with about 12% tin; bronze melts at about 950°) and brass (copper alloyed with around 40% zinc, US$2.20/kg; zinc melts at 420°, brass a bit past 900°) are other possible alloys that could be shaped by this method. Sprinkling 12% pricey tin filings into a copper bed to lower its melting point by 134° seems ideal.

Using one metal as a binder for other metals, as in brazing

Brass is commonly used as a binder for other metals in brazing; you could, for example, use a powder bed of iron filings and deposit brass filings onto it before baking. You could easily get most of the strength and cost of the final piece from the iron filings.

The inexpensive metals — those less expensive than copper — are aluminum (US$2.20/kg), arsenic (US$2.20/kg), iron (US$0.88/kg), manganese (US$0.80/kg), silicon (US$2.40/kg), and zinc (US$2.20/kg); and we should include carbon, since iron is commonly alloyed with carbon (super cheap, depending on purity; flake graphite costs US$1.50/kg) to make steel or cast iron.

Among these, zinc in particular (even without making brass by being mixed with copper) seems like it would be a good choice as a binder for an iron or steel filler, or possibly even for aluminum. Zinc and aluminum form a variety of useful alloys, and apparently there's a technique called "diffusion soldering" similar to this. I'm not sure what would be needed to remove the aluminum-oxide layer from the surface of aluminum powder that has been exposed to air, though, and both zinc powder and aluminum powder are a bit of a fire hazard.

Cast iron

Cast iron melts around 1150° to 1200°, while pure iron melts at 1538°. Steels have intermediate melting points; mild carbon steel is the most common family, and ASTM A36 is a typical mild carbon steel; it has up to 0.29% carbon and 0.28% silicon, and according to the iron-carbon phase diagram, its melting point should still be above 1500°. Cast irons typically contain 1–3% silicon and 2–4% carbon, although the eutectic point is at 4.3%. At 3.5% carbon, the melting point is reduced to 1200°.

So, you could take a bed of ASTM A36 filings and selectively flux them with 3.25% carbon and 1% silicon, then heat them up to almost

1200°, or maybe a bit more, but not past 1500°. The part you've selectively fluxed should sinter, and then you should be able to bake it more thoroughly to make a more homogeneous cast-iron part; deformation from the printed shape should be almost zero due to the silicon content forcing carbon to remain in graphite form.

It might be a better idea to use a much smaller amount of carbon and/or silicon, so that when the powder is heated, only a small part of each filing around each carbon grain is melted, rather than the entire filing; this way, the printed part will not liquefy completely, and the finished part will be high-carbon steel rather than cast iron.

Tungsten carbide

Tungsten carbide, one of the most important industrial ceramics, can be made by reacting metallic tungsten with carbon at 1400° to 2000°; it melts around 2800°, while tungsten melts at 3422° and graphite sublimes at 3600°. So you could "flux" a graphite powder bed (US$1.50/kg) with powdered tungsten (US$200 per "short ton unit", which is 7.19 kg of tungsten, thus US$28/kg) and heat it up to 2000° or a bit less.

Alternatively, you could "flux" the graphite with tungsten trioxide , which melts at 1473°, and heat it only to 900° to react it with the graphite and immediately produce the tungsten carbide.

All of this might need to happen under pressure, I'm not sure.

Alpaca: raising the melting point instead of lowering it

Here in Argentina, the common alternative to stainless steel or silver used in silverware and whatnot is an alloy, invented by Qing China, called "alpaca", from the 19th-century brand name of a German company; worldwide this is typically 60% copper, 20% nickel (US$22/kg, melts at 1455°), and 20% zinc, working out to US$7.50/kg. It looks like silver, it's bactericidal like silver, and it's strong and easy to electroplate with, for example, silver.

The CRC Handbook of Mechanical Engineering gives the melting point of the ASTM B122 formulation of alpaca in its quaint folk units of measurement as 2030°F and its modulus of elasticity as 18 Mpsi, which are 1110° and 124 GPa in SI.

Alpaca probably cannot be 3D-printed in the way discussed above, because it melts at a higher temperature than copper, zinc, or alloys thereof. If you were to try selectively depositing copper and zinc into a bed of powdered nickel, you would have the problem that the product formed would be much larger than the powder it was deposited into.

However, perhaps you could deposit powdered nickel into a bed of powdered 75%-copper brass, and then heat it up to about 1000° or 1100°. If the nickel has sufficiently diffused into the brass, the brass will melt and run away, leaving a solid alpaca object coated and perhaps permeated with liquid brass, as long as the nickel doesn't diffuse too far and become too dilute to prevent melting. This approach would eliminate the extra processing steps that commonly attend powder-bed 3-D printers: the careful brushing of the powder from the recesses of the solid part and extra processing steps to eliminate porosity from the solid part.

This process may depend sensitively on grain sizes and morphologies and on the temperature profile of the process. It's necessary for the nickel to have diffused enough into the alpaca part to solidify it, and for its grains to have sintered together enough to hold together, before the unmodified brass becomes liquid; but if the nickel diffuses too far, you will lose surface detail, as some brass outside the desired part acquires enough nickel to keep it from melting, while some of the alpaca inside the desired part loses enough nickel to allow it to melt.

Using the alpaca approach for steel printing

This approach should also work for printing carbon steel: by increasing the carbon content of the unwanted part of the metal to the eutectic 4.3%, it should become a low-viscosity liquid cast iron at a sharp eutectic melting point of 1148°, permeating the pores of the sintered higher-melting steel and contributing carbon to it.

Adding so much carbon is not quite as trivial as it sounds, because the iron weighs 7.8 g/cc, while carbon black weighs only about 2 g/cc, and graphite about 2.2 g/cc. So 4.3% carbon by weight is about 15% carbon by volume, which is significant. I think it should still fit into the interstices of the iron filing bed if the carbon particles are sufficiently smaller.

(And yes, this is an even bigger problem for alpaca.)

Topics

How to make Dercuano work on hand computers?

Kragen Javier Sitaker, 2019-05-18 (updated 2019-12-30) (56 minutes)

Foreseeably, most personal computers now are hand computers, commonly called "cell phones" or "mobile phones", for archaic reasons (with a few exceptions called by names like "e-readers"). Less foreseeably, they mostly run user interfaces that limit the user's power over them considerably; in particular, although they generally have WWW browsers and most of them can download files and save them locally, they cannot extract a .tar.gz file full of HTML and browse it.

This poses a problem for Dercuano, because right now I am publishing it as a .tar.gz file full of HTML. But its objective is to remain readable even if my server or domain name fails, as they inevitably will someday. It's really important (to me, anyway) that people be able to continue reading Dercuano in that case. There are a variety of possible alternative formats that could work well on hand computers.

The problem: gratuitous handicaps and tiny screens

Hand computers have an additional problem, aside from being gratuitously crippled in a way that requires compatibility hacks: **their screens are tiny**. For example, until I broke the screen, I was using a discount hand computer with a 45×63 mm screen; a more modern one I looked at last night has a 64×115 mm screen. Also, the screens used to be low resolution: the PalmPilot was 160×160 (monochrome!), and the original iPhone was 320×480. (At 163 pixels per inch, that was 50×74 mm, bigger than the one I broke.) Modern cellphones have much higher-resolution screens, and e-readers generally have much larger screens, though with fewer pixels.

Making text readable at all on such small screen sizes requires serious compromises in typographic design. For example, the typography I'm using at the moment (see Dercuano stylesheet notes (p. 374)) is "22px", with max-width of 45em and line-height of 1.5 (em), and 1 em of padding around the body; on my 158 dpi laptop screen, that's a font size of 3.5 mm or 10 (PostScript) points, with 5.3 mm from one baseline to the next. I use a ragged right margin extra vertical whitespace between paragraphs, as is normal on the WWW, and a somewhat smaller font size for <pre> blocks.

At this font size, on my 45×63 mm screen in portrait mode (my observations on the subway and bus suggest that people strongly prefer using their hand computers in portrait mode, only switching to landscape mode to watch landscape-mode videos, play landscape-mode video games, or occasionally read PDF files whose lines are too long), the 7 mm of padding on the left and right would leave room for almost 13 ems of text, about four or five words' worth. Using the greedy paragraph-filling algorithm web standards short-sightedly require (at least in the case where there are floats, according to pcwalton), and especially without hyphenation, this

would frequently have lines with only one or two words on them. Less than 12 of these tiny lines would fit on the screen, one of which will frequently be consumed by a paragraph break, so you might have 40 words of actual text on the screen.

Worse, Chromium's Blink HTML engine, like the WebKit and KHTML engines it derives from, doesn't support hyphenation at all; Firefox's Gecko engine is the only significant WWW browser engine that does, and on hand computers, almost nobody uses Firefox.

Once you add any extra block indentation, like that in a blockquote or indented list, the situation quickly deteriorates to one or two words per line.

Reducing the text size to a less-comfortable size is a necessary compromise to avoid such uncomfortably short line lengths. (Generally, when I read things on it, I also used portrait mode.) Also, though, using less padding around the text is very helpful (in this example, using 0.5em instead of 1em padding would increase the text column width from 13em to 14em). The line length will still necessarily be shorter, which reduces the need for leading between lines to avoid disorientation when moving from one line to the next.

It's possible to do far worse than my default style on hand computers, though. The worst reading experiences on hand computers are when you have very long lines in PDFs or ASCII text files with hard line breaks, such that even in landscape mode, you can't fit an entire line on the screen at a readable font size. This requires you to scroll left and right on every single line to read the text.

Somewhat less annoying are academic papers which preserve the traditional book layout of two columns of text per page, rather than the single-column layout that has become popular recently, since about 1850. The columns are generally narrow enough to be readable on the tiny hand computer screen, which is a great blessing, but once you reach the end of one, you have to spend several seconds panning diagonally across the page to find the top of the next one — and, half the time, that's the wrong thing to do, because the next column is on the next page.

(I lied, though. The worst reading experiences on hand computers are file formats you don't have an app for.)

Some kind of adaptation to widely varying screen sizes is necessary, since hand computers in common use range from the kind of tiny 45×63 screen I mentioned up to Amazon Swindles with 600×800 screens at 167 dpi grayscale, which works out to 91×122 mm, almost 4× as big, and 51% bigger than the 64×115 mm "cellphone" I mentioned above. (For comparison, a page of a paperback book is 105×175 mm and about 600 dpi, but without grayscale.)

Possible formats

DHTML with offline reading via cache-manifest or service workers

The first thing that occurred to me was that I could just add a cache-manifest to the HTML generated for Dercuano so that when a browser loads one page, it loads them all into the appcache, and (at least if you bookmark the thing) the whole thing remains accessible even if you're offline or the server goes down.

This has the advantage that anything that works in the current HTML tarball incarnation of Dercuano would keep working the same way. In fact, more things would work — the difficulties with full-text indexing I mentioned in Dercuano search (p. 1532) wouldn't exist.

This is the lowest-effort approach, but it wouldn't work very well. Although the cache-manifest mechanism is widely supported, including on pretty much all hand computers, it's considered obsolescent (the documentation for it has been removed from the current version of the WHATWG standard), to be replaced with the new and shiny service-workers mechanism. Since Firefox 60 and Chrome 69, it's also unavailable if you aren't using HTTPS. It enjoys invisible resource limits — the amount a browser is willing to cache is not exposed to the user, but typically it's 5MB or 10MB, and if the download fails because not enough space is available, no error message is given; it just fails when you're offline or the server is down.

There's a sort of polyfill to support the cache-manifest API on top of ServiceWorker, but ServiceWorker also requires HTTPS.

The bigger problem, though, is that both service workers and the appcache are totally dependent on, and vulnerable to, the origin server. This violates my intent with Dercuano in three ways:

• If my server is down, one person with a copy of Dercuano would not be able to give it to another person, except by giving them their entire browser state. This means that once my server is gone, copies of Dercuano would gradually diminish one by one until they are all gone, rather than being shared with new people who want them.
• If malicious actors gain access to my server or my domain, they could use that access to delete all the copies of Dercuano, if it were using service workers or appcache. Malicious actors have gained access to the vast majority of domains that were on the web 20 years ago, usually to put generic linkspam pages on formerly high-PageRank domains, so it's a good bet that this will happen sooner or later to canonical.org.
• If a patent examiner reads some idea in their copy of Dercuano, and Dercuano uses service workers or appcache, they can't tell if that idea was inserted into their copy of Dercuano the last time they connected to the internet, or ten years earlier. This means that ideas in Dercuano would not be able to serve as prior art to invalidate patent claims, as "rapid genetic evolution of regular expressions" did.

MobiPocket .mobi format

A more reasonable alternative approach, for which I am indebted to cajg, is to convert Dercuano into some kind of ebook format. Ebook formats in general solve the three problems I mentioned above.

The popular Amazon Swindle hand computer uses a variant of this format. I don't know much about it, but it's not fully documented in public. Its text is formatted with (X)HTML and CSS. Mobipocket themselves did a bunch of work on hyphenation, but their work is no longer available (except on the Swindle), and other .mobi readers may not have such good hyphenation support.

Support for .mobi files is not available on most e-readers (except the Swindle), and on cellphones it is available but not installed by default. You can install, for example, Okular or FBReader to be able

to read them.

.mobi doesn't seem to have very good graphics support — in particular, nothing like SVG or EPS, *but* it does support embedded JS which could, in theory, implement that kind of thing, maybe. It supports embedded GIFs and JPEGs, but with a size limit of 63 KiB.

I'm not sure if one part of a .mobi file can contain a hyperlink to another arbitrary part of it, although it does of course support tables of contents. This is important for Dercuano.

.ePub format, the modern replacement for .mobi

EPUB, as it's sometimes written, continued to evolve after .mobi forked from it around 2005, and the current version *does* support SVG images. It's fully documented, not suffering from the reverse-engineering problem .mobi does. Otherwise (in terms of supported features, preservability, file size, and so on) it seems to be pretty similar.

One giant HTML file

At first I didn't think of this as an option, since my experience with hand computers is that they typically can't read HTML offline reliably.

Recent versions of (Chrome on) Android are capable of saving HTML pages for offline reading, including the CSS and JS and whatnot, so combining the entire contents of Dercuano into a single fifteen-megabyte, six-thousand-page HTML file might be a possible alternative. This would probably require fiddling with the CSS and JS a bit to get it to scale and not clash, but perhaps more importantly, I think Blink may choke on such large HTML documents; it's designed for HTML files two or three orders of magnitude smaller. Even Dillo might balk.

It appears Chrome is saving a multipart/related MIME document with a filename ending in ".mhtml", which is a totally reasonable way to do this, and provides a reasonably readable file adhering to well-known standards, in a single file. It does, however, have a couple of significant drawbacks:

• Basically any useful access to it requires reading the whole thing, though that's really probably the least of your troubles if 90% of it is a 15-megabyte HTML document.
• If you open the file in Chrome from a file manager, Chrome renders it as plain text. It's only when you load it from the "downloads" app that Chrome opens it as expected.

I'm not clear on how easy it is to transfer these from one hand computer to another, which, as I was saying earlier, is a sine qua non. I was hoping it would be a matter of just copying the .mhtml file across, but it doesn't seem to be.

However, the one-giant-HTML-file approach might be useful as a first step in other workflows, like creating PDFs or ePubs.

PDF

That brings us to PDF, which is usually in last place in anyone's list of candidate document formats, due to decades of painful experiences; PDF doesn't support text reflow†, so using it for hand computers whose screens vary by a factor of about 4 would seem, at best, perverse. However, for better or worse, PDF is supported by almost

all hand computers (Android, iOS, and Swindle all ship with PDF support out of the box), and it always looks the same, within the limits of the screen or printer, while maintaining a file size similar to that of gzipped HTML. It supports hyperlinks, including hyperlinks within the document, and it supports vector graphics, including transparency (though not, as far as I know, SVG-like convolution filters). PDF is designed for random access, so a few thousand pages in a document is not a problem on modern computers, including hand computers.

PDF also has the advantage that there are a lot of people out there who take seriously the problems of archiving PDFs and making them searchable. The ISO has a PDF standard and also a standard for a "PDF/A" subset designed for archival. (Well, several non-backwards-compatible versions of the standard, actually, which likely defeats the purpose, but possibly they'll pull their heads out of their asses at some point.)

The worst problems with reading PDF on hand computers, as I said above, result from formatting with long lines. Wide margins are a secondary offense, since in many readers they mean you have to zoom to a readable size every time you switch pages, and when panning on touchscreens, you're always at risk of panning a little bit diagonally and losing the last few letters of the column you're trying to read.

Typically, though, PDF viewers only let you pan diagonally when you're zoomed in in two dimensions. If you have the entire page width visible, you can only pan vertically, and if you're looking at the entire page, you can't pan at all.

† Recent versions of acroread do claim PDF reflow support, but I haven't tried it.

.chm

Microsoft distributes help files in CHM format, which, like ePub, is an archive (in ".cab" "cabinet" format, IIRC) full of HTML files. This used to be popular as a way to distribute technical books, and maybe it still is, but support on hand computers is limited. Play Store app reviews suggest that nowadays it's found a niche for distributing medical reference books to doctors.

My proposed solution: PDF with pages of 24 ems × 60 ems with ½ em of margin all around

Maybe PDF's vices can be turned into virtues.

Consider a page that measures 24 ems by 60 ems, with 1.2-em line spacing and ½ em of margin, so eight to twelve words per line, much like a paperback book, but with much taller pages: 49 lines. On my tiny 45×63 mm hand computer, these numbers give a barely bearable 5.3-point font in portrait mode and a tolerable 7.4-point font in landscape mode, when the page is zoomed to fit the width of the display rather than its height. On the larger 64×115 one I mentioned earlier, these numbers are a tolerable 7.6-point font in portrait mode and an eminently readable 13.6-point font in landscape mode. Indeed, even fitting the height of the page to the display gives a bearable 5.4-point font on that machine.

These four possibilities — landscape zoom-to-width, landscape

zoom-to-height, portrait zoom-to-width, and portrait zoom-to-height — provide four roughly evenly spaced magnification levels covering a linear zoom range of about three to four times, or an areal zoom of about 12 to 20 times. None of them suffer the janky diagonal panning problems that plague PDF reading on hand computers, since none of them require zooming in so far that diagonal zooming is possible. The number of words per line is suboptimal but readable.

Some screen real estate to the left and right of the page is left unused. On a 91×122 mm Swindle, zooming to fit the whole 60-em-tall page in portrait mode gives you a 5.8-point font, but only the middle 49 mm of the display is used. Many PDF readers (I don't remember about the Swindle's) offer an option to view pairs of facing pages next to each other, rather than single pages; doing this on a Swindle-sized screen would give you a 5.4-point font, which is still bearable, and two pages of text at a time.

If we think of an em as nominally representing 12 PostScript points, the 24×60 em page size is 102 mm (4 inches in archaic units) by 254 mm (10 inches in archaic units). So this column size actually closely approximates the size of a column in a traditional two-column folio page, or a two-column A4 or US letter-sized page.

Given how precious hand-computer screen real estate is, we'd probably want to use indentation, rather than extra vertical space, to demarcate paragraphs, in the way that has been standard for several centuries. The addition of PDF's unavoidable page breaks with ragged right margins adds an additional rationale for this: if a sentence starts at the beginning of a line at the top of a page, how can we tell if it starts a new paragraph or not? It will have extra whitespace above it simply because of the page break.

A hypothetical PDF reader that supported zooming to fit the page height, with more than two pages next to each other, would allow reading any number of such columns with horizontal scrolling.

To some extent, small font sizes can be compensated by holding the computer closer to your face, wearing reading glasses, and squinting, but a more absolute limit — without resorting to temporal antialiasing, anyway — is the actual number of pixels. I've done a 3½×6 pixel font that is marginally readable, and I think you can do better than that with antialiasing and especially subpixel rendering, but usually a minimum for reasonable letterforms is 5×8 pixels, and standard VGA fonts were 8×16. But at these line widths, that's not going to be a problem. If we divide the original iPhone's 320-pixel width by 24 ems, we get a line height of 13 pixels, so an average glyph of around 6×13 pixels. And modern hand computers have considerably more pixels than that.

Given that all these point sizes are a little on the small side, and the actual paperback book I was looking at has lines of only about 20 ems wide and is eminently readable, you'd think I could get by with a font size about 10% or 20% larger than what's implied above (and thus 21% or 44% less areally dense). 45 mm / 21 em would be 2.1 mm per em, which is a 6-point font; in landscape mode, the same tiny screen would have 63 mm / 21 em = 8.5 points, which is easily readable. But the other force pushing for smaller fonts and wider lines is the occasional <pre> block, which needs to be able to accommodate 80 columns, nominally 40 ems. That's a text size of 0.6 em for the <pre>.

Using an even larger font size for the normal body text would cause an even larger disharmony between the two text sizes.

Hyperlinks in PDF

PDF supports tables of contents and hyperlinks, but at least the default PDF viewer on Android 7.0 (which is the Google Drive PDF viewer) doesn't seem to have any way to see them. It has a fairly effective scrollbar, though, so page numbers may be a reasonable replacement — but they need to count monotonically from 1 at the beginning, since the page numbers displayed in the Android viewer do that; even though PDF supports page numbers that do things like "i, ii, iii, iv, 1, 2", they are not displayed.

ZUI in PDF for navigating illustrations?

Illustrations (see Dercuano drawings (p. 64)) are a really hard problem in HTML-based formats for small screens: your lines are already too short to flow text around large pictures, and small pictures are unreadable unless they contain only a little bit of information, like sparklines. But if we assume that the reader is using a hand computer with pinch-to-zoom, and our image format is vector, perhaps we can rely on zooming to provide more information about illustrations on demand, and even some degree of hierarchical navigation.

Hyperlink navigation within the illustration is probably not supported, though, and the maximum zoom is probably quite limited; the popular AndroidPdfViewer open-source component defaults to 3× as its default maximum zoom, but the Android 7.0 default PDF viewer defaults to 10×. It also permits zooming *out* until several pages are on the screen, though, sadly, stacked vertically.

Hyphenation and equations in PDF

The major advantage of PDF over the HTML-based formats is that things will look exactly as I formatted them. This means that I don't have to rely on hyphenation support on the reader's computer; I can use a decent hyphenation algorithm, and if necessary I can tweak the text to deal with rotten formatting (although, honestly, I'm trying to import a couple of million words of unfinished notes into this thing; I can't stop to futz with per-paragraph formatting on more than a tiny part of it).

Also, an enormous advantage accrues to math formatting (see Dercuano formula display (p. 495)). In theory, EPUB supports some part of MathML, but MathML rendering is generally kind of shitty (where it's not done through MathJax), and writing MathML is worse. With PDF, I can render equations at build time using T_EX, subsetting Computer Modern fonts as necessary to include just the glyphs I'm using, and get well-formatted formulas.

Further progress

2019-12-28

I've hacked together a janky PDF by parsing the Dercuano output HTML as XML, and now most of the content of Dercuano is readable in this format.

Page sizes and typewriter font woes

Initially I tried the "24 ems × 60 ems with ½ em of margin"

configuration described above, but I found it to be uncomfortably narrow. For regular running text it was reasonably okay, and for low-resolution cellphones that probably means "ideal", but for 80-column-wide <pre> blocks, it was terrible --- that's 0.3 ems per character, and Courier really wants more like 0.63 ems per character, which would be over 50 ems, making non-<pre> text of the same size uncomfortably wide and also requiring a high-resolution screen for readability without constant diagonal scrolling.

(I haven't actually implemented <pre> proper yet.)

Another pressure is that 24 ems is too narrow for a large number of URLs. At some point I guess I'll have to implement some kind of line continuation for long strings like that, but having less broken lines like that will always be better.

However, to some extent text dimensions are fungible. Making text taller makes it more legible, as does making it wider. The much harder constraint on <pre> text is its width; scrolling more because it is taller than would be ideal is far preferable. So, a reasonable alternative is to use a compressed font. I found Bogusław Jackowski and Janusz M. Nowacki's font Latin Modern Mono Light Condensed, which comes in regular and oblique versions (but no bold), which is derived originally from Knuth's Computer Modern Teletype, which is in the public domain; but Latin Modern has much broader coverage of some 760 Unicode characters than cmtt does.

lmtlc, as this font is called in the TeX Live distribution, demands only about 0.36 ems of horizontal space per character, and is still quite readable, although visibly compressed. I had to use FontForge to convert it from the OTF on CTAN because Reportlab said, "TTF file "lmmonoltcond10-oblique.otf": postscript outlines are not supported."

So I've widened the page width to some 29 ems (and extended it vertically to 66 ems, purely for reasons of silly nostalgic printer traditions --- US letter paper is, in medieval units, 11 inches long, and a standard 12-point line height thus gives you 66 lines). This reduces the page count from some 4700 to 3700. Even 3700 seems large for a book of only 1.3 million words or less, but 500 of those pages are the topic listings at the end.

As I said before, a key consideration is for the PDF version of Dercuano to be readable on hand computers without diagonal scrolling or reflowing, because reflowing a PDF is pretty hard. This has two aspects: pixel readability and absolute size.

As for pixel readability, reviewing dimensions from above, the PalmPilot was 160x160, and the iPhone 1 was 320x480. At 24 ems wide in landscape mode, 480 pixels is 20 pixels per em, like a 10x20 xterm font; this is quite comfortable. 160 pixels across 24 ems is only 6.7 pixels per em, which is at the very edge of readability. So, by going to 29 ems, I'm sacrificing PalmPilot readability, which would be 5.5 PalmPilot pixels per em, but 16.6 original-iPhone pixels per em --- still quite readable in landscape mode.

In addition to avoiding pixelation to prevent unreadability in an absolute sense, I'd also like to keep the letters reasonably large in millimeters to avoid sacrificing readability-without-a-magnifying-glass. The original iPhone was 50x74 mm; 50 mm across 29 ems is 1.72 pixels per em, which is 4.9 printer's points. That's a pretty small font! That's why I was trying to

make do with 24 ems. But in landscape mode on an iPhone-1-sized device that would be a 7.2-point font, suboptimal but not outside the realm of readability. On the discount hand computer I was using earlier this year, the screen was 45x63 mm. 29 ems across 63 mm makes it a 6.1-point font: painful to read, but, again, not infeasible.

If that hadn't worked, maybe /usr/share/texlive/texmf-dist/fonts/opentype/public/cm-unicode/comuntt.otf would have been another possibility, maybe with some kind of coordinate transformation.

Remaining major bugs

I have a number of showstopper bugs left in the PDF generation; among them:

• The vertical positioning is wrong, so PDF links are displaced vertically relative to their target text, and I have to leave a bunch of extra bottom margin to minimize the number of pages that get truncated.
• I haven't implemented `<pre>` yet.
• The 3% or so of notes that aren't well-formed XML are getting totally mangled, with mojibake and total loss of formatting. For many of these, getting the formatting totally right would require implementing tables and SVG, which may not be in the cards this weekend, but surely I can do better than this.
• I haven't implemented font cascade fallbacks yet for missing characters.
• The ET Book license needs to be included.
• `<li><p>foo</p></li>` puts the paragraph on a separate line from the `<li>` bullet.

There are also a lot of other bugs that aren't showstoppers but might be easy to fix:

• Headers aren't red.
• Headers aren't underlined.
• Line spacing is too tight.
• Blockquotes aren't visibly distinct at all.
• `<script>` and `<style>` contents are treated as text.
• I don't have page numbers on links yet.
• An extra space gets added after the ends of every HTML element.
• Notes aren't in any order in the PDF file.
• I think it's splitting on no-break spaces as well as normal spaces, so they aren't no-break.
• The link to lua-%23-operator may be broken.

And other bugs that are serious but maybe aren't in either category:

• There are no per-note tables of contents.
• There are no superscripts or subscripts.
• `<ol>` isn't bulleted.
• The PDF is huge, like 12 megs.

Font cascade fallback fonts

As a fallback for monospaced text, /usr/share/fonts/truetype/droid/DroidSansMono.ttf might work, although it's going to be much wider than lmtlc and only covers 874

codepoints (though some of those are things I use that aren't in lmtlc!). /usr/share/fonts/truetype/ttf-liberation/LiberationMono-Regular.tto of covers only 663. /usr/share/fonts/truetype/ubuntu-font-family/UbuntuMono-R.ttf has 1225, comparable to the 1259 in /usr/share/fonts/truetype/msttcorefonts/cour.ttf. /usr/share/fonts/truetype/dejavu/DejaVuSansMono.ttf has 3197, and /usr/share/fonts/truetype/freefont/FreeMono.ttf has 4126. Moreover, FreeMono has 3511 codepoints that lmtlc doesn't, and DejaVu Sans Mono has 2645, of which 515 are also not in FreeMono.

So, for monospace coverage, if you had to choose a single fallback font with no worries about licensing, it would be FreeMono, expanding lmtlc's 760 codepoints to 4271, but if you could choose a second one, DejaVu Sans Mono would expand that to 4786.

For serif body text, ET Book (a copy of Bembo) covers only 233 codepoints. The corresponding brand-name fallback fonts would be /usr/share/fonts/truetype/freefont/FreeSerif.ttf with 6450 codepoints and /usr/share/fonts/truetype/dejavu/DejaVuSerif.ttf (my browser's standard fallback) with 3331 codepoints. From the size, it is clear that neither of these covers Chinese; the built-in PDF font that seems to work best for Chinese (in Reportlab, the PDF-generation library I'm using) is `reportlab.pdfbase.cidfonts.UnicodeCIDFont('STSong-Light')`, which is sadly a gothic monoline (I would say "sans-serif" but of course what's missing isn't really serifs) font. Also, I've figured out how to tell which codepoints a TrueType font covers using Reportlab: `reportlab.pdfbase.ttfonts.TTFontFile( '/usr/share/fonts/truetype/ubuntu-font-family/UbuntuMono-R.ttf').charToGlyph` is a dict. I don't know how to do this for STSong-Light, so I don't know how to fall back from it.

Freefont is a GNU project, although it seems to have largely gone idle in 2012. The licensing is GPLv3+, which is somewhat aggressive as fonts go, and it's not clear that there's a legal way to embed it, or a subset of it, into a PDF file and then convey that PDF file to others.

Oh, actually there's a special exception for document embedding in its README, which Debian left out of /usr/share/doc/fonts-freefont-ttf/copyright:

> Free UCS scalable fonts is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.
>
> The fonts are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
>
> As a special exception, if you create a document which uses this font, and embed this font or unaltered portions of this font into the document, this font does not by itself cause the resulting document to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the document might be covered by the GNU General Public License. If you modify this font, you may extend this exception to your version of the font, but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version.

DejaVu is an extended version of Bitstream Vera, which was distributed under a BSD-like license that requires changing the name of extended versions; the DejaVu changes are in the public domain. They are far from complete Unicode coverage, lacking even some Greek and Cyrillic and most Arabic, as well as all the Indic scripts. Still, I think it might cover most of the characters I actually use.

DejaVu Serif isn't very harmonious with ET Book; it's a slab-serif font with little emphasis and a tall x-height --- roughly as far as it could be from ET Book while still being technically a serif font. It does have ℤ and ² and ³ and ^{64} and μ and × and ∞ and ÷ and Ω and ≈ and ⇒ ∃ ε ∈ $_{01}$ †, though many of them copy and paste wrong. Combining arrow above $\vec{v}$ is missing (renders as an empty box), but maybe I'm outputting it wrong. And it's missing ℓ. But those are the only things I've seen missing so far.

The elusive 'ℓ' is found in FreeMono, Liberation Mono, (Microsoft's) Courier New, and Droid Sans Mono, and likely their non-monospaced equivalents as well. Liberation is a Red Hat font set licensed under the GPLv2 with a document-embedding exception plus some other weird anti-Tivoization exception.

Liberation Serif covers ≈, †, ∞, ←↓↑→, ² and ³, and Greek, but not $^{-6}$ or ɑ or $_2$ or ^{48} or ℤ. It's somewhat more harmonious with ET Book.

Freefont's FreeSerif is considerably more harmonious with ET Book than the others, and it does contain ℓ.

Misparsed data

I've been trying to use ElementTidy to read in the things ElementTree can't handle directly, about 30 of the 997 notes in Dercuano, but this has been failing completely. One reason is that the tag names it gives me are bullshit like '{http://www.w3.org/1999/xhtml}html'. Another is that it seems to be parsing things as some incorrect encoding.

elementtidy is apparently dead having just been removed from Debian a few months ago, so it may not have been the best choice...

This seems to work to solve the mojibake problem:

```
>>> b = TidyHTMLTreeBuilder.TidyHTMLTreeBuilder(encoding='utf-8')
>>> b.feed(open('dercuano-20191226/notes/nova-rdos.html').read())
>>> t = b.close()
```

Although honestly, looking at the source, I think this does the same thing without TidyHTMLTreeBuilder:

```
import _elementtidy
t = ET.XML(_elementtidy.fixup(open(
       'dercuano-20191226/notes/nova-rdos.html').read(), 'utf8')[0])
```

...although that's not without ElementTidy, just without its Python. It still has the namespace problem, though.

But the fixup() function there seems to just give us the stdout and stderr we would get from invoking HTML Tidy. Which, as it turns out, has options `-ashtml` and `-utf8` that would probably do the right thing here without saddling us with an `xmlns`. I wonder if Python tidylib has a way to get that?

This looks promising:

```
>>> xs = tidylib.tidy_document(
        open('dercuano-20191226/notes/nova-rdos.html').read(),
        {'input-encoding': 'utf8',
         'output-encoding': 'utf8',
         'output-html': True})
>>> print xs[0][:1024].decode('utf-8')
```

That *almost* works:

```
>>> t = ET.XML(xs[0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 124, in XML
cElementTree.ParseError: undefined entity: line 16, column 43
```

It's complaining about ` `.

Well, can I use ElementTidy (or for that matter tidylib) in XML mode, but just strip off the namespace tags?

```
>>> t = ET.fromstring(_elementtidy.fixup(
    open('dercuano-20191226/notes/nova-rdos.html').read(), 'utf8')[0])
>>> def deprefix(tree):
...  for kid in tree:
...   deprefix(kid)
...  tree.tag = re.compile('{.*}').sub('', tree.tag)
...
>>> deprefix(t)
```

That seems to have worked! And tweaking the tidylib recipe above (no output-html, yes numeric-entities) allowed me to excise the ElementTidy dependency. So that's one out of six showstopper bugs fixed.

`<pre>`

As MDN says about CSS "white-space: pre":

> Sequences of white space are preserved. Lines are only broken at newline characters in the source and at `<br>` elements.

Right now I have a "font stack" which is separate from the element stack and also a "current link" which is restored from the element stack. But now I'd need to have a "white-space" stack so that I restore white-space to its `normal` value at the proper place.

I think a better alternative is to use the element stack to restore elements of the current style, which can include link destination, font-family, font-size, and white-space. Then I can just pass the current style to `render_text`.

`white-space: pre` is simple to implement:

```
#words = re.split('[ \n\r\t]+', text)
words = re.split('\n', text)

    if True or t[0].getX() + width > max_x:
        newline(c, t, font)

    t[0].textOut(word)# + ' ')
```

Okay, I have that working. Four showstopper bugs left: newlines after bullets, the ET Book license, vertical positioning, and font cascade fallbacks.

In the process, though, the PDF seems to have grown by about 700K and become slower to display in some PDF viewers. I suspect resetting the font on every word may be causing this, so I'm going to try adding another level of indirection so I can make an apples-to-apples test.

Indeed, without redundant font setting, it takes 4m37s of user time and produces a 12.4 MB PDF, while with redundant font setting, it takes 5m45s and produces a 12.9 MB PDF. So the extra complexity to avoid redundant font setting is worthwhile.

2019-12-29

Well, so, I revisited the code that emits textobjects, and I made it emit a new textobject for every line, which can be positioned with some appropriate Y-offset, and now I have the links actually over the text they're supposed to be (except when a link splits across pages, which is still a bug) although at the cost of an extra megabyte. This also enables me to eliminate the fudge-factor margin at page bottoms, which cuts the book down to 3718 pages.

So, that's one more showstopper bug down, and so now it's down to three remaining showstopper bugs: the missing ET Book license, the newlines following bullets, and tofu. I think FreeSerif and FreeMono are reasonable fallback fonts, but I have to figure out how to do the fallbacking in practice.

Font cascades

So, I've added FreeFont FreeSerif as a fallback font and added a bunch of logic for font fallback. I probably should add some kind of regexp-based fast path for when all the characters in a word are in ET Book, because it's noticeably slower, but it does seem to cover nearly all the characters I use. FreeSerif-Italic seems to be missing a bunch of subscript letters I use, though, unless I'm screwing something up.

It takes about 9 minutes instead of about 5 minutes to generate the PDF now.

It turns out that subscript index letters like ₖ or $_k$ (bold: $\mathbf{_k}$, bold italic: $\boldsymbol{_k}$) are not in (this version of) FreeFont, but they *are* in DejaVuSerif and DejaVuSerif-Italic. They're used in Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) and Observable transaction possibilities (p. 2086). So I'm going to go ahead and add the relevant DejaVu fonts, including for typewriter text (ₖ, $_k$, $\boldsymbol{_k}$, $\mathbf{_k}$), to the font cascade.

That seems to solve the problem. So maybe I can declare the Dercuano PDF pipeline tofu-free, at the cost of dozens of megabytes of fonts added to the source repository.

So, the remaining showstopper problems are the ET Book license and newlines following bullets. Then I can work on some problems that are annoying but less critical, like subscripts and superscripts, page numbers, blockquote formatting, ligatures, per-note tables of contents, extra spaces, extra newlines in <pre>, header colors, and header padding.

(Although, as it turns out, I spent some time adding caches to the font cascade code to see if I could make it a little less slow. This cut the PDF build time from 9.5 minutes to 8 minutes.)

Newlines following bullets

This happens with the construct `<li><p>fulano</p></li>`. Entering the `<li>` causes one newline (followed by a bullet), and entering the `<p>` causes another. The correct solution is to make the `<p>` a box vertically nested inside the `<li>` box without any extra padding, so that, unless there's something above it to push it down, their tops are at the same position. But the janky PDF generation script doesn't have a box model; it just has newlines. How could we avoid generating a newline?

Well, the problem is specifically when a block element is the first thing inside another block element. So maybe I can just have a boolean about whether we're at the top of a block.

Okay, I was able to hack that in, and as a bonus I can use it to eliminate paragraph indents when paragraphs are the first thing inside a block element. That avoids list items having an indent on the first line next to the bullet. It will probably also help with blockquotes. Gah, blockquotes.

That just leaves the ET Book license as a showstopper.

2019-12-30

No, wait, another one just popped up: in between the two 6's in "mooTzNujJpx 66\n" in the middle of powerful-primitives.html, the font switches to italic, which it doesn't in the browser, and stays that way for the rest of the note. The culprit seems to be `mOoTzNujJpx 6<I=EInw>6\n`. I tweaked this to use `` in the Markdown, which hopefully will make the problem go away, but even if it interpreted that as an `<i>` tag, it should have at some point found an end to it --- maybe Tidy worked too hard here...

Oh also I implemented **bold** as `typewriter` due to what looks like a copy-and-paste error. Fixed!

I've fixed another couple of problems mentioned earlier in a drive-by fashion: `<script>` and `<style>` contents and no-break spaces.

So, the biggest remaining problems with the PDF, in more or less priority order (a sort of mix of estimated effort with estimated benefit):

- The link to the ET Book license is broken.
- Headers don't have enough padding above them.
- Links are not indicated in any printable way, so they're invisible in, for example, MuPDF. Also, they don't have page numbers, even when they are internal links, so PDF viewers without link support (such as printers and Google Drive's PDF viewer, the default on Android) can't use them.
- Overlong lines are cut off rather than wrapped.
- Individual notes have no tables of contents.
- `<sub>` and `<sup>` are not implemented.
- Character-level markup like `<em>` produces spurious spaces.
- Character-level markup inside `<pre>` produces spurious newlines.
- The notes are not in any particular order, but they should be in the roughly chronological order they are in the table of contents.
- URL-encoded links are broken.
- I haven't imported file `ceramics-notes`.
- The PDF is humongous, 13.5 megabytes.
- Tables aren't implemented.

• Ordered lists <ol> are bulleted rather than numbered.
• Headers aren't red or underlined.
• Line spacing is too tight.
• Blockquotes aren't visibly distinct at all.
• Text isn't hyphenated, justified, or ligated (even things like "fi" and "fl".)
• SVG isn't implemented.
• Sometimes there are spurious blank lines between paragraphs and list items.
• Superscript 0 and 456789 are larger than superscript 123.
• Fallback characters in <pre> have the wrong width.

This is a lot to fix in the next 22 hours and I'm definitely not going to finish all of it, but I ought to be able to make a significant dent.

Okay, adding the ET Book license was a little harder than expected, but done.

I sort of have the padding thing working. It's not working for the "Topics" section at the end because that's the first thing in a and my check for paragraphs in list items means that they don't get a newline. I guess I could make that specific to list items.

Now I have the page numbers thing sort of working, although as with LaTeX, page number references will only work the *second* time you generate a document, which in some sense doubles the generation time from 8 minutes to 16. This is sort of alarming given that I have 20.5 hours left, which is 82 times 16 minutes. I can only do 82 more full rebuilds at that pace. This code will only ever be run 82 more times, ever.

How about overlong lines? I could use a regular "\" to indicate the wrapping of the line, although I think an outdented "–" would be nice. But then I need to chop the overlong word up into lines somehow. If I were just positioning it at an (x, y), I feel like I could easily enough position it a full column width to the left, but the textobject seems to be taking care of positioning for me, unfortunately. So maybe a better option is to binary search on word widths.

Damn it, I just scalded my hand with this teakettle. Not sure being awake at 4 AM is such a good idea. But I still have almost 20 hours left.

All right, I have overlong lines chopped and marked with little circles, sort of like flowchart connector circles. And it's almost sunrise, and damn is it hot, despite the air conditioner going full blast.

Adding tables of contents with ElementTree shouldn't be rocket science, but those tables of contents will be sort of lame without bookmarks to jump to. So I'd need to add a bookmark for each header, which I might as well try to add to the document's outline as well; actually with some PDF viewers that would be a sufficient navigational interface.

However, the 1300 or so outline items are already a bit of a problem for many PDF viewers; I'm not sure how well they'll handle another order of magnitude. I may put this off for a few hours and work on other problems.

Superscript and subscript are supposedly implemented by reportlab.pdfgen.textobject.setRise. That can be made to work... although line breaking between a base and the exponent is possible and pretty undesirable. Also it seems like the line spacing below

increases for a superscript and decreases for a subscript, which is pretty bogus; this very note has some trouble with that with the word "TEX". This is maybe enough of an implementation for the moment --- now it's a formatting problem instead of a semantic problem --- but it sucks pretty bad.

Sunrise is well underway, though the streetlights have not yet gone out.

The character-level markup problem is because the way words get separated in the document is that every word gets a space appended to it, regardless of whether what followed it was a space, the element end, or an element beginning. Originally I was using `words = text.split()` but now that I'm using `re.split` this problem should actually be easy to fix:

```
>>> re.split(r'[ \t]+', 'a b')
['a', 'b']
>>> re.split(r'[ \t]+', 'a b ')
['a', 'b', '']
>>> re.split(r'[ \t]+', ' a b ')
['', 'a', 'b', '']
```

So, I only want to append a space if the word is not the last word in the string.

This small change yields an enormous improvement in formatting. I was whispering, "Holy fuck, holy fuck, holy fuck," as I looked at the results. Exponents are better (in, *e.g.*, Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138)); formulas with italic letters are better (for example, Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138)'s presentation of the law of cosines); inline typewriter text is better (in, *e.g.*, Eur-Scheme: a simplified Ur-Scheme (p. 876), though it still has major problems with `<pre>`); italic words in paragraphs are better; links are better.

A similar but somewhat larger change fixed the `<pre>` problem in Eur-Scheme: a simplified Ur-Scheme (p. 876), although now I am getting to the point where I am surprised when some code works, which is probably a danger sign that I'm introducing bugs. It's now light enough outside that the streetlights have gone out, though there is still no direct sunlight. Maybe I should sleep for a while if I can manage it; I have almost 18 hours left in the day.

I also just tweaked the link boxes to go 0.1 ems to the right as well as 0.1 ems to the left of the link text, and tried URL-decoding URLs to fix the problems with links to Improving Lua #L with incremental prefix sum in the ∧ monoid (p. 2008) and $1 recognizer diagrams (p. 1264), which seems to have worked.

So, now that I've solved the above eight problems, that leaves the following problems:

- Individual notes have no tables of contents.
- The notes are not in any particular order, but they should be in the roughly chronological order they are in the table of contents.
- Blockquotes aren't visibly distinct at all.
- Superscript and subscript screw up the line spacing. One of the

worst examples is on the second page of Some thoughts on SDF raymarching (p. 312), which is totally unreadable. The beginning of Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) also has a somewhat less egregious example.
• I haven't imported file `ceramics-notes`.
• The PDF is humongous, 16 megabytes.
• Tables aren't implemented.
• Ordered lists `<ol>` are bulleted rather than numbered.
• Headers aren't red or underlined.
• Line spacing is too tight.
• Text isn't hyphenated, justified, or ligated (even things like "fi" and "fl".)
• SVG isn't implemented. (It's only used in Dercuano drawings (p. 64) and A mechano-optical vector display for animation archival (p. 3047), so translating the seven or so diagrams manually into PDF paths and operations might be a reasonable option.)
• Sometimes there are spurious blank lines between paragraphs and list items.
• Unicode superscript 0 and 456789 are larger than superscript 123.
• Fallback characters in `<pre>` have the wrong width.
• Superscripts and subscripts don't nest; see Dercuano stylesheet notes (p. 374) in the first example of superscripts to see an egregious error.

I'm pretty pleased with how the result looks now, actually, although there are still clearly places where it does the wrong thing.

After sleeping

I slept 6 hours and now have 10 hours left.

I tried an optimization that turned out to slow things down by 10%, that of handling word spaces separately from the words. I hacked in a reasonable approximation of English spacing, in the sense of larger spaces after sentences: colons, periods, exclamation marks, and question marks get a double-sized space after them, except when it's a period at the end of an abbreviation. (In particular, an extra space is added after ordinals in sequences like "1. Ready. 2. Set. 3. Go!")

This is a small change, but I think it improves nearly every paragraph, even though there are still much worse problems than the use of French equal sentence spacing throughout the book.

It would probably work better to use the indication of double spaces (or newlines) after periods in the original Markdown, since I'm pretty consistent about doing that, but, bleh.

So, what next? I think I should see if any of the next three items turn out to be relatively yielding: individual-note tables of contents, ordering of notes, and blockquote formatting.

I was thinking as I went to sleep that hanging indents (as are conventional for bulleted lists) should be relatively easy to handle: add a property to the style that has a string to place before each new line, and set it to a few spaces. This is a crude approximation of proper indentation for bulleted lists, but it might be adequate, and in particular for blockquotes. This does not exist as a CSS stylesheet property, except in the sense that `margin-left`, or `padding-left` can be used to indent the contents of a block with whitespace, and `text-indent` can be used to give the first line of each paragraph an extra indent.

An absolute minimum thing to do for blockquote formatting is to reduce the font size, and that's easy, so I'll do that.

Hmm, not quite so easy, because I haven't implemented font inheritance for block elements, so paragraphs inside a blockquote don't inherit the font. So I implemented font inheritance for block elements.

Also I added `<ol>` and `<ul>` back as block elements, causing them to have space at the top, so that the quoted lists in Flexures (p. 2211) don't collide with the headers right above them.

Oh! I think I know how to fix the subscript/superscript problem. I just need to do the opposite setRise before setRise(0), because the implementation of setRise in Reportlab adds an increment to `self._y`:

```
def setRise(self, rise):
    "Move text baseline up or down to allow superscrip/subscripts"
    self._rise = rise
    self._y = self._y - rise    # + ?  _textLineMatrix?
    self._code.append('%s Ts' % fp_str(rise))
```

So, when we restore the rise back to zero, we need to also restore `_y`. So if we previously did setRise(6) we should do setRise(-6) before setRise(0). Well, that will work for the simple case of 0; what if the rise was previously 2? We don't want setRise(2) to result in an additional 2 points of displacement for `_y`. So we should do setRise(-8) before setRise(2). So, I got that fixed, although sometimes the fix does the wrong thing when it restores the rise on a different line.

Hmm, "P_{out}" looks like shit maybe because of inconsistent font fallbacks; it's the same problem as the superscript digits.

Okay, so with those fixes, I have 6 hours left.

Topics

- Politics (p. 3639) (39 notes)
- Archival (p. 3322) (34 notes)
- Compression (p. 3384) (28 notes)
- Dercuano (p. 3406) (16 notes)
- Hand computers (p. 3492) (10 notes)
- Fonts (p. 3458) (9 notes)
- HTML (p. 3508) (6 notes)
- Browsers (p. 3351) (6 notes)
- Typography (p. 3760) (5 notes)
- Zooming user interfaces (ZUIs) (p. 3782) (4 notes)
- CSS (p. 3398) (3 notes)
- MathJax (p. 3567) (2 notes)
- The PDF file format

Changing the basis to a more expressive one with better affordances

Kragen Javier Sitaker, 2016-09-29 (5 minutes)

Tile graphics hardware with sprites, like that on the NES, was designed to make scrolling color video-game graphics achievable in 1983 at a reasonable cost. But it also makes certain kinds of visual effects fairly easy to achieve, like wallpapering an area with a repeating pattern or scrolling or globally changing the appearance of a certain kind of tile. Other effects, like changing the on-screen size of anything, are harder.

Sprites themselves are an affordance for making moving objects on a static background; without hardware sprites, given mutable tiles, you could always just use four tile types per sprite, compositing the sprite in the appropriate place in each tile type each frame. I saw MS-DOS text-mode programs that did this for the mouse pointer.

Given enough tile definitions to have a separate definition for each tile on the screen, and enough colors that you can

Nowadays our affordances run more to gradients and alpha-blending, although to some extent that's only because our program output doesn't go through a video codec, which has different affordances.

Polynomials

There's a variety of ways you can express the space of polynomial functions.

You can express a degree-N polynomial as N+1 coefficients, which is convenient for many calculations but doesn't make it particularly easy to achieve any given effect.

Given some k, you can express it as N+1 coefficients of a polynomial in (x-k). If k happens to be one of the zeroes of the polynomial, you only need N coefficients, since the constant term is zero.

Extending this idea to its limit, you can express it as a scale factor and a set of zeroes: $3(x-1)(x+2)(x+5)$ has a scale factor of 3 and zeroes at 1, -2, and -5; its representation as zeroes is thus [3, 1, -2, -5]. It works out to $3x^3 + 18x^2 + 9x - 30$, so its representation as coefficients is [3, 18, 9, -30], although sometimes people prefer to write that in the other order. (Transforming from zeroes to coefficients is easy; going the other way can be hairy.)

You can express it with its values at some given abscissas and then use Lagrange interpolation. For a given set of abscissas, such as the nonnegative integers, you have a constant set of Lagrange basis polynomials; the polynomial is a weighted sum of its values at these points, which is to say that the transformations between this Lagrange form and the form as coefficients are linear. But the Lagrange form allows you to express the polynomial function in terms of its value at some given points, which is often more convenient.

That of course leaves open the question of the set of abscissas to

choose. Sometimes it's more expressive to be able to choose which abscissas to use, but this is somewhat dangerous with Lagrange interpolation, just as with splines of orders greater than 3; a bad choice of nodes can easily lead to unwanted oscillations (the Runge phenomenon). If you're approximating some existing smooth curve, using the Chebyshev nodes minimizes this.

The Chebyshev polynomials form another orthogonal basis into which you can linearly transform a polynomial.

For a given step size and starting point, you can express the polynomial in terms of the initial state of the method of divided differences for tabulating values of the polynomial. These values are a linear transform of the coefficients and thus also of the Lagrange form; you can easily derive them by calculating some values of the polynomial and writing a difference table. They are called the "Newton form" of the polynomial. The Newton form is in some sense not very expressive, in that the numbers in it don't correspond very directly to any interesting feature of the polynomial itself. However, it's very convenient for calculation.

Order-n polynomials on the real numbers (or even the rationals, or any dense set) are entirely determined by their value and first n derivatives at any arbitrary point; this is the basis of the Taylor-series approximation of a function, which is the analytic equivalent of the Newton form. A polynomial function is equal to its Taylor series as soon as the Taylor series' order equals or exceeds that of the original polynomial. So the derivatives at some given point are yet another representation of the polynomial, another one which is linearly related to the coefficients.

It isn't necessary for all the derivatives to be at the same point; it's adequate to specify N+1 values or derivatives as long as no two of them are the same.

State machines

You can specify state machines as regular expressions...

Topics

Dumb vocoder

Kragen Javier Sitaker, 2017-05-10 (2 minutes)

Watching TinselKoala's (TKLabs's) transmission-line model demonstration, which uses an analog delay line made of inductors and capacitors to set up a standing wave that varies by frequency, with LEDs connected to show where the antinodes were, it occurred to me that perhaps you could use this approach for analog detection of voice frequency spectrum.

He's using the interference patterns of the reflection in the unterminated analog delay line to get different patterns of lights on the LEDs. This is purely a linear phenomenon, so the pattern of standing voltages that will show up for signals at different frequencies is the sum of the voltages for the individual signals.

I think that you can get signals to bounce back and forth in the transmission line many times by using a large resistor on the input. Only harmonic frequencies will produce recognizable patterns of nodes and antinodes, I think? I don't know how well this will work to separate harmonic frequencies from inharmonic ones.

Anyway, can you use these standing-wave shapes to detect a set of frequency buckets spread across several octaves? I think so. And you don't need, necessarily, a compact analog delay line made of capacitors and inductors. You may be able to use AM modulation to upconvert a waveform from audio frequencies up to frequencies high enough that you can fit them into a short piece of coax, although perhaps they won't be resonant frequencies.

Topics

Iterative string formatting

Kragen Javier Sitaker, 2013-05-17 (9 minutes)

Every language seems to have string formatting as some kind of DSL. C has its `%3.4g` nonsense. JS, Perl, and Python programmers often use the same thing. C++ has iostream format manipulators `cout << setw(3) << setprecision(4) << x`. BASIC had `PRINT USING`. Fortran had FORMAT. Common Lisp has its own insanely powerful FORMAT, taking strings like "~3,4g", which means something almost, but not quite, completely unlike `%3.4g`. Even Excel has its own numeric formatting language; if you want your lakhs and crores properly displayed, apparently you can say "Format Cells → Custom" and then `[>9999999]##\,##\,##\,##0.00;[>99999]##\,##\,##0.00;##,##0.00;[<-9999999](##\,##\,##\,##0.00);[<-99999](##\,##\,##0.00);##,##0.00`

Forth's embedded domain-specific language for "pictured numeric output"

Forth "pictured numeric output" is, like C++'s approach, an embedded DSL, but much more direct in its functioning; the typical example is something like

```
: .pesos <# # #  [char] . hold  #s  [char] $ hold #>  type ;
```

which has the format string actually backwards, because that's the order in which you do decimal conversion, and is also using double-precision fixed-point math because that's the way you do things in Forth.

Because it's an embedded DSL, you can use the standard methods of abstraction. For example, if you want more flexible currency display, you can factor out parts of your format string into subroutines or variables:

```
defer currency
: pesos [char] $ hold ;
: pounds C" £" dup 1+ C@ swap 2 + C@ hold hold ; ( assuming UTF-8 )

variable decimal-point

: .$ <# # # decimal-point @ hold #s currency #> type ;

: argentina  [char] , decimal-point !  ['] pesos is currency ;
: usa        [char] . decimal-point !  ['] pesos is currency ;
: england    [char] . decimal-point !  ['] pounds is currency ;

8.4300 7.8300 d+ d2/ d>s constant dolar-blue \ www.dolarblue.net 2013-03-31
1.5195 d>s constant £                        \ in dollars; xe.com 2013-03-31
: /$ 10000 swap m*/ drop s>d ;
: convert 2dup ." AR"    argentina                .$
          2dup ."  = US" usa     dolar-blue /$    .$
          2dup ."  = "   england dolar-blue /$ £ /$ .$
          2drop ;
```

```
cr 1500.00 convert   \ my rent in Buenos Aires
```

You could even put a complex iterative state machine into your format string; in this case we take advantage of Forth's compile-time metaprogramming to factor out a duplicative control structure:

```
char , constant thousands-separator
: ?#  postpone #
    postpone 2dup  postpone d0=
    postpone if  postpone exit  postpone then ;  immediate
: #,s  begin  ?# ?# ?#  thousands-separator hold  again ;
: #             s  ?# ?# ?#  thousands-separator hold
  begin ?# ?#  thousands-separator hold  again ; ( for e.g. 1,00,00,00,000 )
: .,  <# #,s #> type ;   : .              <# #             s #> type ;
```

Forth's particular implementation of this concept is quite limited, though; you have to build the output string backwards, the buffer area is quite limited in size, and it's eager.

The more usual approach of external DSLs

But, when it's a non-embedded DSL, the DSL only goes so far. I seem to end up writing a lot of string formatting code that looks more or less like this:

```
    def __repr__(self):
        return 'Note(%r, %r, %r, %r)' % (self.instrument,
                                         self.start_time,
                                         self.pitch,
                                         self.volume)


    def __repr__(self):
        return '[[ %r %r %r ]]' % (self.left, self.op, self.right)
```

This is, in all likelihood, grossly inefficient. Erlang's approach, which I used in Ur-Scheme, is called "IO lists", and it's similar to "ropes": represent strings with arbitrary unbalanced binary trees, waiting until I/O time to walk the trees in O(N) time. This gives you O(1) concatenation, which makes string concatenation efficient, but it doesn't really solve the DSL problem.

(The description makes it sound complicated but the implementation is about 23 lines of code in Ur-Scheme, and it could have been shorter with a little more cleverness.)

Here's a use of IO lists in Erlang:

```
join([])          -> "";
join([W])         -> W;
join([W1, W2])    -> [W1, " and ", W2];
join([W1, W2, W3]) -> [W1, ", ", W2, ", and ", W3];
join([W1|Ws])     -> [W1, ", ", join(Ws)].
```

When invoked with `io:format("~s; ~s; ~s; ~s; ~s.~n", [commalist:join([]), commalist:join(["Apple", "Banana", "Carrot"]), commalist:join(["One", "Two"]), commalist:join(["Lonely"]), commalist:join("abcdefg")])`., this produces, "; Apple, Banana, and Carrot; One and Two; Lonely; a, b, c, d, e, f, and g."

Smalltalk's approach: output to a string (and capture it if necessary)

Smalltalk has an interesting approach; in general, instead of concatenating strings (which is APLish , in Smalltalk) you sequentially write them to a stream, and if you need them in a string, you can use `WriteStream on: String new` for that string, and there's even a `String#streamContents:limitedTo:` method for this. Here's the code that gets run from `Date today asString` in Squeak 3.9, eventually yielding '31 March 2013', with some added commentary in case you're not familiar with Smalltalk syntax:

```
asString
    "Answer a string that represents the receiver."

    "^, read 'answer', means 'return' and was once displayed as ↑."
    ^ self printString

printString
    "Answer a String whose characters are a description of the receiver.
    If you want to print without a character limit, use fullPrintString."

    ^ self printStringLimitedTo: 50000

printStringLimitedTo: limit
    "Answer a String whose characters are a description of the receiver.
    If you want to print without a character limit, use fullPrintString."
    "The following declares a local variable limitedString."
    | limitedString |
    "_ is assignment and was traditionally displayed as ←.
    Nowadays it is usually written := instead.
    [:s | self printOn: s] is JS's function(s){return self.printOn(s)}.
    So the following in JS would be
    var limitedString = String.streamContentsLimitedTo(function(s) {
        return self.printOn(s);
    }, limit);
    "

    limitedString _ String streamContents: [:s | self printOn: s] limitedTo: limi
ot.
    limitedString size < limit ifTrue: [^ limitedString].
    ^ limitedString , '...etc...'

printOn: aStream
    self printOn: aStream format: #(1 2 3 $  3 1 )  "$  is space, #() array"

printOn: aStream format: formatArray
    "Print a description of the receiver on aStream using the format
    denoted the argument, formatArray:
        #(item item item sep monthfmt yearfmt twoDigits)
        items: 1=day 2=month 3=year will appear in the order given,
        separated by sep which is eaither an ascii code or character.
        monthFmt: 1=09 2=Sep 3=September
        yearFmt: 1=1996 2=96
        digits: (missing or)1=9 2=09.
```

```
    See the examples in printOn: and mmddyy"
    | gregorian twoDigits element monthFormat |
    "The #dayMonthYearDo: method strikes me as bizarre in this context:"
    gregorian _ self dayMonthYearDo: [ :d :m :y | {d. m. y} ].
    twoDigits _ formatArray size > 6 and: [(formatArray at: 7) > 1].
    1 to: 3 do:
        [ :i |
            element := formatArray at: i.
            element = 1
                ifTrue: [twoDigits
                        ifTrue: [aStream
                                nextPutAll: (gregorian first asString
                                        padded: #left
                                        to: 2
                                        with: $0)]
                        ifFalse: [gregorian first printOn: aStream]].
            element = 2
                ifTrue: [monthFormat := formatArray at: 5.
                    monthFormat = 1
                        ifTrue: [twoDigits
                                ifTrue: [aStream
                                        nextPutAll: (gregorian middle asString
                                                padded: #left
                                                to: 2
                                                with: $0)]
                                ifFalse: [gregorian middle printOn: aStream]].
                    monthFormat = 2
                        ifTrue: [aStream

                                nextPutAll: ((Month nameOfMonth: gregorian middleo
o)

                                        copyFrom: 1
                                        to: 3)].
                    monthFormat = 3
                        ifTrue: [aStream

                                nextPutAll: (Month nameOfMonth: gregorian middle)o
o]].
            element = 3
                ifTrue: [(formatArray at: 6)
                            = 1
                        ifTrue: [gregorian last printOn: aStream]
                        ifFalse: [aStream
                                nextPutAll: ((gregorian last \\ 100) asString
                                        padded: #left
                                        to: 2
                                        with: $0)]].
            i < 3
                ifTrue: [(formatArray at: 4)
                            ~= 0

                        ifTrue: [aStream nextPut: (formatArray at: 4) asCharactero
o]]]
```

And down inside of String:

```
padded: leftOrRight to: length with: char
    leftOrRight = #left ifTrue:
        [^ (String new: (length - self size max: 0) withAll: char) , self].
    leftOrRight = #right ifTrue:
        [^ self , (String new: (length - self size max: 0) withAll: char)].
```

And the `Number#asString` method that's being used to do the actual conversions ends up writing to a string stream, then reversing it:

```
printStringBase: base
    | stream integer next |
    self = 0 ifTrue: [^'0'].
    self negative ifTrue: [^'-', (self negated printStringBase: base)].
    stream _ WriteStream on: String new.
    integer _ self normalize.
    [integer > 0] whileTrue: [
        next _ integer quo: base.
        stream nextPut: (Character digitValue: integer - (next * base)).
        integer _ next].
    ^stream contents reversed
```

This demonstrates that Smalltalk code is not uniform in simply using `nextPutAll:`. Indeed, since Smalltalk has garbage collection, code that uses string concatenation is simpler than code that repeatedly writes to an output stream; it's just less efficient.

The streamContents:limitedTo: method mentioned earlier takes advantage of XXX

```
streamContents: blockWithArg limitedTo: sizeLimit
    | stream |
    stream _ LimitedWriteStream on: (self new: (100 min: sizeLimit)).
    stream setLimit: sizeLimit limitBlock: [^ stream contents].
    blockWithArg value: stream.
    ^ stream contents
```

The "limitBlock" here can be invoked from deep inside of whatever code runs from `blockWithArg`, leaping over many stack frames to discard half-completed execution and return the so-far-accumulated data. (I assume it executes `ensure:` blocks on the way?)

Coroutines for string formatting

Instead of a buffer or a stack, use a channel, and run the string-formatting code as a coroutine. That way you get the benefits of an embedded DSL, along with laziness, laziness which in this context will very often reduce memory usage rather than increasing it; and you can avoid generating intermediate copies of parts of the string you're generating.

This requires efficient coroutine support in your language, and likely a certain amount of buffering: probably at least a machine word, if not a whole cache line. Otherwise you need a coroutine context switch on every byte transferred, which will slow some applications significantly.

Topics

Notes on reading eForth

Kragen Javier Sitaker, 2007 to 2009 (9 minutes)

These notes are on stuff I got out of EFORTH.ZIP, 61213 bytes, which I downloaded from http://www.baymoon.com/~bimu/forth/ linking to http://www.baymoon.com/~bimu/forth/eforth/EFORTH.ZIP. Beware! This software is not under an explicit free-software license, and the web page says, "Permission is granted for non-commercial use, provided this notice is included."

Bill Muench's eForth may be the closest thing I've seen to a minimal FORTH kernel. The assembly-language kernel of 8086 eForth ITC16i 971014.1, an indirect-threaded FORTH system, implements only these 36 words:

```
EXIT ( -- ) ( R: a -- ) ( 6.1.1380 )( 0x33 ) \ ITC
EXECUTE ( xt -- ) ( 6.1.1370 )( 0x1D ) \ ITC
_LIT ( -- n ) ( 0x10 )
_ELSE ( -- ) ( 0x13 )
_IF ( f -- ) ( 0x14 )
C! ( c a -- ) ( 6.1.0850 )( 0x75 )
C@ ( a -- c ) ( 6.1.0870 )( 0x71 )
! ( n a -- ) ( 6.1.0010 )( 0x72 )
@ ( a -- n ) ( 6.1.0650 )( 0x6D )
RP@ ( -- a )
RP! ( a -- )
>R ( n -- ) ( R: -- n ) ( 6.1.0580 )( 0x30 )
R@ ( -- n ) ( R: n -- n ) ( 6.1.2070 )( 0x32 )
R> ( -- n ) ( R: n -- ) ( 6.1.2060 )( 0x31 )
SP@ ( -- a )
SP! ( a -- )
DROP ( n -- ) ( 6.1.1260 )( 0x46 )
SWAP ( n1 n2 -- n2 n1 ) ( 6.1.2260 )( 0x49 )
DUP ( n -- n n ) ( 6.1.1290 )( 0x47 )
OVER ( n1 n2 -- n1 n2 n1 ) ( 6.1.1990 )( 0x48 )
CHAR- ( a -- a )
CHAR+ ( a -- a ) ( 6.1.0897 )( 0x62 )
CHARS ( n -- n ) ( 6.1.0898 )( 0x66 )
CELL- ( a -- a )
CELL+ ( a -- a ) ( 6.1.0880 )( 0x65 )
CELLS ( n -- n ) ( 6.1.0890 )( 0x69 )
0< ( n -- f ) ( 6.1.0250 )( 0x36 )
AND ( n n -- n ) ( 6.1.0720 )( 0x23 )
OR ( n n -- n ) ( 6.1.1980 )( 0x24 )
XOR ( n n -- n ) ( 6.1.2490 )( 0x25 )
UM+ ( u u -- u cy )
REDIRECT ( asciiz -- f )
!IO ( u -- ) ( initialize I/O device )
?RX ( -- c -1 | 0 )
TX! ( c -- )
BYE ( -- ) ( 15.6.2.0830 )
```

And these "procs" --- not FORTH words but machine-code

routines:

```
PROC RESET ( cold start entry )
PROC LIST1          ( entry for : words ) \ ITC
PROC VCOLD             ( cold start entry )
```

Those 39 primitives are the basis for implementing everything else. Here are some brief notes on them. It took me a while to understand how "next," works in eForth; it's defined in EMETA.X86 and inserts a single JMP instruction to the "NEXT1" label. I'm also not quite sure about the conditionals. So my instruction counts may not be quite right.

- EXIT (--) (R: a --) (6.1.1380)(0x33) \ ITC
  This pops the return-stack pointer to return from a colon definition.
  5 instructions.
- EXECUTE (xt --) (6.1.1370)(0x1D) \ ITC
  This calls a word that's on the stack.
  2 instructions.
- _LIT (-- n) (0x10)
  Pushes the next cell in the colon definition.
  3 instructions.
- _ELSE (--) (0x13)
  Unconditional branch in a colon definition.
  3 instructions.
- _IF (f --) (0x14)
  Conditional branch in a colon definition.
  7 instructions.
- C! (c a --) (6.1.0850)(0x75)
  Store a byte.
  4 instructions.
- C@ (a -- c) (6.1.0870)(0x71)
  Fetch a byte.
  5 instructions.
- ! (n a --) (6.1.0010)(0x72)
  Store a cell.
  3 instructions.
- @ (a -- n) (6.1.0650)(0x6D)
  Fetch a cell.
  3 instructions.
- RP@ (-- a)
  Push the return stack pointer.
  2 instructions.
- RP! (a --)
  Set the return stack pointer (e.g. to throw an exception or switch threads)
  2 instructions.
- >R (n --) (R: -- n) (6.1.0580)(0x30)
  Push something on the return stack.
  3 instructions.
- R@ (-- n) (R: n -- n) (6.1.2070)(0x32)
  Copy something off the return stack. (Not in the minimal set.)
  2 instructions.

• R> (-- n) (R: n --) (6.1.2060)(0x31)
Pop something off the return stack.
3 instructions.
• SP@ (-- a)
Get the stack pointer (e.g. for .S).
3 instructions.
• SP! (a --)
Set the stack pointer (e.g. to throw an exception or switch threads.)

2 instructions.
• DROP (n --) (6.1.1260)(0x46)
2 instructions.
• SWAP (n1 n2 -- n2 n1) (6.1.2260)(0x49)
5 instructions.
• DUP (n -- n n) (6.1.1290)(0x47)
4 instructions.
• OVER (n1 n2 -- n1 n2 n1) (6.1.1990)(0x48)
Not in the minimal set.
6 instructions.
• CHAR- (a -- a)
Subtract 1. Not in the minimal set.
4 instructions.
• CHAR+ (a -- a) (6.1.0897)(0x62)
Add 1. Not in the minimal set.
4 instructions.
• CHARS (n -- n) (6.1.0898)(0x66)
No-op. Not in the minimal set.
1 instruction.
• CELL- (a -- a)
Subtract 2 (the size of a 16-bit cell). Not in the minimal set.
4 instructions.
• CELL+ (a -- a) (6.1.0880)(0x65)
Add the size of a cell (2). Not in the minimal set.
4 instructions.
• CELLS (n -- n) (6.1.0890)(0x69)
Multiply a number by the size of a cell (2). Not in the minimal set.
4 instructions.
• 0< (n -- f) (6.1.0250)(0x36)
See if a number is less than 0.
4 instructions.
• AND (n n -- n) (6.1.0720)(0x23)
Bitwise.
5 instructions.
• OR (n n -- n) (6.1.1980)(0x24)
5 instructions.
• XOR (n n -- n) (6.1.2490)(0x25)
5 instructions.
• UM+ (u u -- u cy)
Unsigned add, pushing a carry flag.
8 instructions.

The code words up to this point are the fundamental internal operations of the virtual machine. They total 117 instructions. The next few code words are OS interface primitives:

• REDIRECT (asciiz -- f)
Open a file as stdin using INT 21h calls; returns “f” success or failure.
15 instructions.
• !IO (u --) (initialize I/O device)
No-op, for compatibility with some other eForth systems I don’t know about.
2 instructions.
• ?RX (-- c -1 | 0)
Read a key if ready using INT 21h, otherwise return 0.
16 instructions.
• TX! (c --)
Emit a character using INT 21h.
4 instructions.
• BYE (--) (15.6.2.0830)
Exit program with INT 20h.
1 instruction.
Those MS-DOS interface primitives are 38 more instructions.

• PROC RESET (cold start entry)
Placed at the beginning of the program, to jump to VCOLD, wherever it might be.
2 instructions.
• PROC LIST1 (entry for : words) \ ITC
Pushes the instruction pointer onto the return stack and sets a new one.
5 instructions.
• PROC VCOLD (cold start entry)
Sets up registers and starts the interpreter.
14 instructions.
So there are 21 more instructions; the whole thing is 117 + 38 + 21 = 176 machine-code instructions, if I counted it correctly. EFORTH.COM is 7936 bytes, of which the last 157 are “junk DNA,” all lower-case ‘b’, presumably so it would end on a 256-byte boundary; the part of EFORTH.COM up to the the end of the definition of TX! is 762 bytes, including the dictionary structure and copyright notice, and I think that encompasses basically the above machine-code words. (BYE and VCOLD are at the end, so they’re not included in the 762.)
Some things not included in the machine-language subset (that maybe should be): multiplication and division; subtraction; negation; PICK; string I/O; bit shifts; memory block copying.
The rest of eForth is about 700 lines of FORTH, defining 191 more subroutines:
NOOP _VAR _CON HEX DECIMAL ROT NIP 2DROP 2DUP ?DUP + D+ INVERT NEGATE DNEGATE S>D ABS DABS - PICK 0= = U< MAX MIN WITHIN LSHIFT UM* * RSHIFT UM/MOD SM/REM FM/MOD /MOD MOD / +! COUNT BOUNDS /STRING ALIGNED 2! 2@ MOVE FILL -TRAILING >ADR >BODY _USR 'S _PASS _WAKE PAUSE STOP GET RELEASE SLEEP AWAKE ACTIVATE BUILD DIGIT? >NUMBER NUMBER? HERE PAD <# DIGIT HOLD # #S #> SIGN CATCH THROW ABORT ?KEY KEY NUF? EMIT SPACE EMITS SPACES TYPE CR _" _S" _." _ABORT"

S.R D.R U.R .R D. U. . ? PACK DEPTH ?STACK ACCEPT SAME? _DELIMIT _PARSE NAME> WID? SFIND _[[SOURCE PARSE-WORD EVALUATE ASCIIZ STDIN FROM QUIT ALIGN ALLOT S, C, , COMPILE, LITERAL CHAR [CHAR] ' ['] PARSE .((\ SLITERAL ,C" S" ." ABORT" _]] GET-CURRENT SET-CURRENT DEFINITIONS ?UNIQUE HEAD, IMMEDIATE COMPILE-ONLY REVEAL RECURSE POSTPONE CODE next, :NONAME : ; _DOES> DOES> CREATE VARIABLE CONSTANT USER HAT WORDLIST ORDER@ GET-ORDER SET-ORDER _MARKER MARKER BEGIN THEN RESOLVE MARK IF AHEAD ELSE WHILE UNTIL AGAIN REPEAT .S !CSP ?CSP >CHAR _TYPE _DUMP DUMP .ID WIDWORDS WORDS NAMED? SSEE SEE COLD

Which is pretty much just a normal FORTH a bit on the minimal side, with just a few extras (multitasking, a decompiler), minus blocks (FORTH's low-budget "virtual memory") and an assembler.

(There are also some variables, which I haven't counted.)

The resulting MS-DOS executable, as I mentioned, is 7936 bytes.

The "metacompiler" is in a separate source file and is not included in those 7936 bytes; and Muench did not include the source to his assembler, just an executable, called B.EXE, which is relatively large.

So we have an "inner core" of 176 instructions in 39 routines, about 700–800 bytes including debug info; an "outer core" of another 191 FORTH routines, about 7000 more bytes (about 1000 of which is just their names); and presumably your program on top of that.

(It actually uses only the 22 instructions MOV, JMP, SUB, ADD, ADC, LODSW, POP, PUSH, AND, OR, JZ, JNZ, JB, XOR, SHL, CWD, XOR, INT, DEC, CLI, STI, and CLD, although there are a variety of operand types in use with some of those; so writing a minimal assembler to support it would be pretty straightforward.)

Looks like this isn't the original eForth though...

Topics

- Programming (p. 3658) (286 notes)
- Small is beautiful (p. 3714) (40 notes)
- Instruction sets (p. 3526) (40 notes)
- Forth (p. 3461) (19 notes)

Introduction to closures

Kragen Javier Sitaker, 2019-12-07 (5 minutes)

Many programmers who haven't delved deeply into functional programming are puzzled about what closures are and why they would care. And this confusion sometimes gets worse when they find explanations that explain how closures are implemented, namely by storing some extra values along with a pointer to some function code and supplying those values to the function when it is invoked. But that doesn't get to the heart of the matter, which is:

Closures are the language feature that allow you to create new functions at runtime.

Here's an example. You can express the function (+) that adds two numbers in just about any programming language. In old-style JS you would write `function foo(a, b) { return a + b; }`, for example. And similarly you can express the function (3+) that adds three to things: `function foo(b) { return 3 + b; }`. But (3+) is obviously just one example of a large class of functions like (4+), (5+), (-3 +), and so on; it would clearly be nice to be able to generate instances of this class of functions automatically instead of copying and pasting code and editing the constant in it.

Closures are the language feature that make this possible; in JS, for example, you can write `function adder(a) { function foo(b) { return a + b; } return foo; }` and you have a function which, at runtime, creates arbitrary new instances of this adder class. This clearly requires the binding of `a` to, say, 3 or 4 or 5 or -3, to stick around somewhere, rather than being discarded when `adder` exits, which you will note is not at all explicit in the original code. Forth doesn't have closures but gets a similar ability in a different, more explicit way, "at compile-time", that is, when you're building the dictionary; you say `: adder create , does> @ + ;` which allows you to say things like `3 adder 3+`.

You might think that closures only allow the creation of a limited class of copy-and-paste functions at run-time, but in fact they allow you to create *any computable function* at run-time. In fact, you only need *one function that creates closures* to do this; Moses Schönfinkel showed in the 1920s† that it was possible with two curried functions, conventionally called S and K:

```
function S(x) {
    function S2(y) {
        function S3(z) { return x(z)(y(z)); }
        return S3;
    }
    return S2;
}

function K(x) {
    function K2(y) { return x; }
    return K2;
}
```

Or, in modern JS:

```
const S = x => y => z => x(z)(y(z)), K = x => y => x
```

And, in 2001, Chris Barker demonstrated that you can do it with just one, which can be written as `function ι(f) { return f(S)(K); }`. The reductions from things like ordinary arithmetic, to the λ-calculus, to S and K, to Barker's ι combinator, are an interesting kind of mind-bending, the kind that makes you wonder why it took you so long to understand them once you finally do understand them.

Pascal supports closures in a limited form that keeps them from surviving the function that instantiated them, while some other programming languages like Smalltalk-80 and GCC C have that restriction but don't enforce it, so your program will probably just crash if you violate it. Modern Smalltalk has full-fledged unlimited-extent garbage-collected closures like JS and Scheme, as do most modern languages: modern C++, Perl since Perl 5, Ruby since forever, Kotlin, Java since Java 8 (?), and so on. Smalltalk is particularly interesting in this regard because it uses closures instead of conditionals and loops, using an extremely lightweight syntax and some cheats in the compiler to make this practical. Some Scheme implementations actually use closures to implement not only conditionals and loops but even local variable declarations and statement sequencing; Olin Shivers wrote a widely-cited dissertation on how to make *that* insanity practical after struggling with the problem for years.

That might be more information about closures than you wanted, but hopefully it's enough to orient you and let you figure out what you want to know more about.

Footnote

† Actually, Schönfinkel invented the SKI-combinator system in the 1920s, but Curry's further work on it and Church's invention of the λ-calculus had to wait for the 1930s, and it wasn't until the 1940s that the concept of "computable functions" was really clear, thanks to the work of Curry, Gödel, Church, and Turing in the 1930s; at which point it became clear what Schönfinkel had really proved. At least that's my understanding of the history, but I've never read Schönfinkel's paper.

Topics

Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once?

Kragen Javier Sitaker, 2018-06-17 (updated 2019-07-29) (7 minutes)

Could you listen to every FM radio station at once on your PC?

Standard FM radio runs from 87.5 MHz to 108 MHz with channels typically every 800 kHz or so in a given geographical area, although in theory they can be spaced as close as 200 kHz apart. That's 20.5 MHz divided into 103 200 kHz channels, of which typically about 25 are used. If an SDR is to pick up all of that 20.5 MHz at once, it needs to sample at 41 Msps or, preferably, substantially more, like 60 Msps, which is probably feasible with some work — I mean 640×480 video at 60 fps is 18 Msps per color channel, 55 Msps in total. (Direct downconversion sampling may be feasible with some filtering. Sampling at baseband would require 215 Msps.)

A couple of different algorithms occurred to me to do this: one using a bank of phase-locked loops ("PLLs") and one using a phase vocoder. Both seem likely to be feasible on a desktop PC, but the phase-vocoder approach should scale to a larger number of channels more efficiently.

Analyzing channels with PLLs

But then there's the issue of how to analyze the channels. You can of course run a PLL on each channel — say, 25 or 30 PLLs in all. Officially FM mono has 15 kHz of audio bandwidth, but unlike in AM, in FM there isn't a simple relationship between audio bandwidth and radio bandwidth — a 1 Hz audio signal could be encoded by swinging the frequency of the FM carrier back and forth over a "frequency deviation" of 1 MHz. The frequency deviation actually used is ±75 kHz. You need at least 15 kHz of audio out of each of those channels, so you need frequency information out of each of the PLLs at at least like 30 ksps. To decode FM stereo, you need to decode oscillations of the carrier frequency at up to 53 kHz, and thus your PLL needs to give you a frequency readout at 106 ksps or more.

The frequency reported by the PLL is always in some sense an average over some time period, and that's what these numbers mean. If it's "30 ksps" then the frequency needs to be able to slew from −75 kHz to +75 kHz in 33 μs, a slew rate of 4.5 GHz/s, and the frequency can't be an average over much more than those 33 μs. If it's "106 ksps" then it's 16 GHz/s and 9.5 μs. At 60 Msps, that's averaging the oscillation over 2000 samples for mono and 600 samples for stereo, which seems eminently feasible.

This approach requires about 15 operations per sample per PLL, which works out to some 400 operations per sample, 24 billion operations per second. It's possible to implement this without any multiplies at all.

Analyzing channels with a phase vocoder

An alternative to using PLLs might be to use a phase vocoder. This

amounts to taking an STFT of the signal often enough to reliably unwrap the phase — at least three times per cycle of the beat frequency, say — with enough frequency resolution to have only one sinusoid at most in each frequency bin.

As before, we need to divide the spectrum into frequency bins small enough that at most one station is in each, but the FFT bins are evenly spaced from 0 up to Nyquist, with one bin for every two samples in the window. We can't choose the bin center frequencies freely the way we could with the PLL approach.

If we use about 400 kHz spacing, then we need at least 52 frequency bins, so at least 104 samples in the STFT window, say 128, which gives us 64 frequency bins, ranging over 30 MHz if we're at 60 Msps; this gives us 468'750 Hz per bin. But the windows can overlap by as much as you want.

A somewhat tricky issue is that, in a phase vocoder, the rate at which you need to inspect the phase of each bin is not determined by how fast it is changing *frequency* (as in the PLL case), but by how fast it is changing *phase*. In a bin of 468-kHz width, the putative partial in the bin can only vary by ±234 kHz from the bin center frequency. This means we need about 600'000 STFT windows per second, which thus only overlap by 28 samples.

I think that, roughly speaking, this involves $(7 = \lg 128) \cdot 2 \cdot 5.1$ multiplies per sample, which works out to 71, which means we need 4.3 billion multiplications per second at 60 Msps. This sounds feasible but very challenging. For a CPU, anyway; it should be easy for a GPU.

(Intel and NASoftware reported in 2011 that a 256-point call to the VSIPL function `vsip_ccfftip_f`, which is probably not the fastest FFT function for this since its input is complex, takes 440ns using AVX on one core of a Core i7-2710QE when running at 2GHz, so it should in fact be feasible on a modern CPU, if not on my laptop. In the same slide deck, they also report results on a Core i7-2715QE (?) as 23273 megaflops, which I guess means each FFT is 10240 operations, or 40 operations per sample, which is a lot more than the 16 multiplies I was guessing.)

That sounds almost an order of magnitude better than the PLL approach, but it requires multiplies. So it might turn out that the approaches actually have similar efficiency in practice.

You might think to reduce computational load by using a smaller number of STFTs per second. But if you are using fewer STFTs of the same size per second, without increasing the size of the STFTs, you lose the ability to track frequencies near the edges of the bins; their phases vary too fast to be unwrapped, and they alias into frequencies closer to the center. To avoid this, you must increase the size of the STFTs exactly proportional to the reduction in the number of windows you shingle each second with. This almost exactly cancels out the original reduction in computational load, *except* that now you can decode more channels, *and* the logarithmic factor of the FFT complexity increases. So, for example, if you do 1024-sample STFTs instead of 128-sample STFTs, you can decode 512 radio channels instead of 64, at the cost of about 25% more computation.

This consideration suggests that, for a small number of channels, the PLL approach should be more efficient, and for a large number of channels, the FFT-based phase vocoder should be more efficient.

They just happen to be about equal at about the number of channels that exist in FM radio.

Topics

Lexical gaps

Kragen Javier Sitaker, 2017-06-15 (1 minute)

Seen on Twatter, "Some English lexical gaps."

candor	candify	candific	candid	candible
fervor	fervify	fervific	fervid	fervible
horror	horrify	horrific	horrid	horrible
liquor	liquefy	liquific	liquid	liquible
livor	livify	livific	livid	livible
lucor	lucify	lucific	lucid	lucible
pallor	pallify	pallific	pallid	pallible
rigor	rigify	rigific	rigid	rigible
stupor	stupefy	stupific	stupid	stupible
terror	terrify	terrific	terrid	terrible
torpor	torpify	torpific	torpid	torpible
vigor	vigify	vigific	vigid	vigible
tepor	tepify	tepific	tepid	tepible

Topics

- Humor (p. 3511) (9 notes)

Ideas to ship in 2014

Kragen Javier Sitaker, 2014-04-24 (updated 2019-05-05) (35 minutes)

These are some 50 ideas I have that I think are important for me to pursue and in some form "ship", either by explaining and thinking about them ("shipping" an essay), researching them ("shipping" a review), or experimenting with them ("shipping" experimental results and, in some cases, software). The question is, which ones are the *most* important to pursue.

I can't possibly do all of them, or even a large minority of them, so I'm making this list for two reasons: first, to help me prioritize them (feedback welcome!), and second, to get enough information out there that someone else can maybe tackle the ones that I'm not going to do.

Here I'm using "AOWTBITF" as an abbreviation for "Amount of work to bring it to fruition" and "SNSTWBU" for "Smallest next step that would be useful".

The natural relationship between violence and religion

Summary: an essay about how boundaries between religious groups inevitably produce violence as long as both groups believe in self-defense.

Benefit: if I'm wrong, maybe someone will tell me; otherwise, maybe people will understand violence and religion better, better enabling them to organize society to minimize violence.

AOWTBITF: 8 hours. **SNSTWBU**: write an outline, ½ hour.

The effectiveness of Leviathan in reducing violence

Summary: an essay exploring whether Hobbes' proposed justification for the existence of states is empirically justified by what our current state of anthropological, epidemiological, zoological, and archaeological knowledge, which has advanced substantially since 1651.

Benefit: if I'm wrong, maybe someone will tell me; otherwise, people might understand violence better, better enabling them to organize society to minimize violence.

AOWTBITF: 32 hours. **SNSTWBU**: dig up a copy of Pinker's angel book, 4 hours.

The path toward a world not organized around violence

Summary: an essay exploring whether there's an alternative to Leviathan.

Benefit: if I'm wrong, maybe someone will tell me. Otherwise, maybe people will better understand violence and its role in society and have a clear plan for making the world less violent; and my own point of view on violence will be publicly declared.

AOWTBITF: 16 hours. **SNSTWBU**: write an outline, 1 hour.

De-heroizing mass killers

Summary: an essay exhorting people not to honor the military. Related to http://lists.canonical.org/pipermail/kragen-tol/2010-June/000916.html.

Benefit: if I'm wrong, maybe someone will tell me. Otherwise, if the argument is effective, it weakens the credibility of those with the strongest inclination to argue against pacifism and nonviolence.

AOWTBITF: 16 hours. **SNSTWBU**: write an outline, 1 hour.

The equivalence between violence and censorship or dishonesty

Summary: an essay arguing that advocacy of free speech necessarily entails advocacy of nonviolence. Gandhi talked about this, as did the Redonditos de Ricota. Related to http://lists.canonical.org/pipermail/kragen-tol/2008-February/000880.html, which discusses how censorship causes violence.

Benefit: if I'm wrong, maybe someone will tell me. Otherwise, people might understand the seriousness of censorship and dishonesty and be less willing to endorse them as a result, although presumably some people will go the other way and conclude that permitting free speech amounts to collective suicide.

AOWTBITF: 8 hours. **SNSTWBU**: write an outline, ½ hour.

The prospects for material abundance, the path to an abundance-based economy

Summary: an essay about the future of society: we're living in an age where many things that were once scarce have become so abundant that anyone could have as much as they want, and it seems likely that more and more things will be abundant in this way. How do we take advantage of this? Mentioned in http://lists.canonical.org/pipermail/kragen-tol/2010-June/000916.html.

Benefit: if I'm wrong, maybe someone will tell me. Otherwise, maybe I'll come up with an idea or two that's worth implementing and improves our chances of surviving as a civilization.

AOWTBITF: 8 hours. **SNSTWBU**: write an outline, ½ hour.

The obsolescence of the Marxist classes for understanding modern society

Summary: an essay, or perhaps book, proposing more accurate replacements for Marx's concepts of "bourgeoisie" and "proletariat", which made some sense in the 19th and 20th century, however oversimplified they may have been, but today are more misleading than useful; social dynamics have changed fundamentally, and even in the 20th century, the predictions Marx made based on his theory could hardly have been more incorrect.

Benefit: I'll understand both society and Marxism much better by writing the essay. Surely other people will tell me I'm wrong, which might improve my understanding further if they have something more substantive to say than just "false consciousness!". Ultimately, I'll provide a framework for understanding 21st-century society that

supplants the Marxist framework commonly used by Marxists and capitalists today, but writing this essay is just the first step in that process. (Fuck, that sounds arrogant, doesn't it? I hope the high aims I'm setting for myself don't backfire on me as they did for Aaron.)

AOWTBITF: 512 hours. **SNSTWBU**: finish carefully reading Capital, 128 hours.

Why people do things

Summary: a series of essays analyzing human motivations in different cross-cutting ways: a division between personality, situation, and free will; a division between instinct, culture, and self-interest; a division between loyalties to the group, beliefs in ideals, and egoism; and a division between habit, improvisation, and planning. The essays will integrate the best current understanding from experimental psychology with a poorly-understood version of Buddhism, while remaining accessible through parables and analogies.

Benefit: I'll understand human motivation better, including, in particular, my own, which might help improve my ability to act consciously, but also that of others, which will improve my ability to cooperate with other people. Hopefully other people will tell me where I'm wrong and improve my understanding further. Maybe by reading the essays, other people will understand these issues better too.

AOWTBITF: 64 hours. **SNSTWBU**: translate the first draft of the first essay into English, revising it, and post it, 8 hours.

How to be right

Summary: an essay on techniques I have found useful in abandoning beliefs that are at odds with reality and adopting new ones that are more accurate, and explaining why I think this is a desirable thing to do; in short, a bite-sized introduction to philosophy.

Benefit: anyone will be able to point to this essay whenever someone accuses them of having a strong desire to be right, as if that were a bad thing, perhaps citing some one-liners from it. Maybe someone will point out that some of these techniques aren't as useful as I think they are, or that I've left out others, which could substantially improve my ability to be right in the future if I learn from them.

AOWTBITF: 16 hours. **SNSTWBU**: brief outline, 1 hour.

In favor of the ontological legitimacy of ghosts and moods

Summary: an essay arguing that, although the modern mindset scoffs at entities such as "ghosts", "demons", and "God", these concepts seem to be pragmatically at least as useful as metaphors for grappling with our incomprehensible universe as other concepts that the modern mindset accepts without question, and they ontologically seem to have the same kind of objective existence as such everyday entities such as "laws", "dollars", "theorems", "moods", "nations", "rights", "corporations", and "birthday parties".

Benefit: I don't know, probably people will think I'm nuts, but hopefully it will keep me from getting too attached to my own

conception of the universe. Maybe it'll change some other people's minds and keep them from being too sure of themselves, too.

AOWTBITF: 8 hours. **SNSTWBU**: brief outline, ½ hour.

States as a kind of corporation and corporations as a kind of state

Summary: an essay describing the parallels between states and other corporations, both historically and at present, and arguing that it doesn't make much sense pragmatically to relate to the two institutions on different terms; also, exploring the possible near-future evolution of these and related institutions. The concept is mentioned in passing in http://lists.canonical.org/pipermail/kragen-tol/2007-October/000870o2.html.

Benefit: maybe someone will point out important differences between the institutions that I've overlooked. Or, if I'm correct, maybe other people will sharpen their understanding of these institutions, which will benefit them and society.

AOWTBITF: 8 hours. **SNSTWBU**: brief outline, ½ hours.

FIR kernel factorization

Summary: find out if FIR filtering in the time domain can be done with less computation by factoring a FIR kernel, approximately or exactly, into a convolution of two or more sparser FIR kernels, or FIR kernels with more heavily quantized coefficients; then publish the results, a comparison with the efficiency of convolution in the frequency domain, and software for reproducing them.

Benefit: if the technique works more efficiently than known IIR and DFT-based techniques for a significant set of applications, which is unlikely, people will be able to do some kinds of linear filtering of signals with substantial improvements in efficiency, which will have applications in medicine, communications, remote sensing, music, and data compression, particularly in cases with very low hardware budgets or very high performance requirements.

AOWTBITF: 32 hours. **SNSTWBU**: implement some particularly easy filters in the time domain and compare the computational efficiency to frequency-domain convolution, 4 hours.

LZ77 modified with carefully chosen contexts

Summary: LZ77 wastes most of its sliding window on substrings that are unlikely to appear again. Tampering with the sliding-window contents in a deterministic fashion that the decoder can reproduce, which doesn't seem to have been tried before, could produce substantially better compression. Try this out and publish the results and resulting software.

Benefit: if it works, it could produce a novel variant of LZ77 with compression comparable to LZMA, but perhaps faster; this would improve the efficiency of everything that stores data on a disk or transmits it over a wire, if it's sufficiently adoptable that people adopt it. Unfortunately, for the same reason, it's pretty unlikely to work; there are a *lot* of smart people panning for gold in this river.

AOWTBITF: 32 hours. **SNSTWBU**: ask A. if someone's tried the idea already, 1 hour.

The magic kazoo

Summary: toy pianos are popular because they're easy to use and kids love making music, but they're still a little hard to use, and they're not very portable. An electronic synthesizer whose pitch and rhythm was instead controlled by the human voice would be much easier to use, and could be downloaded as an Android or iOS app or manufactured as a separate electronic device the size of a stick of gum.

Benefit: lots of new people could make music with synthesizers. It would be a hell of a lot of fun, and it would provide a much more comprehensible answer to "What have you been up to lately?" than, say, "exploring the ontological legitimacy of ghosts and moods".

AOWTBITF: 128 hours. **SNSTWBU**: write a software vocoder, 8 hours.

Mechanical computation with lookup tables

Summary: make a general-purpose mechanical computer using some version of the heightfield lookup table mechanism described in http://lists.canonical.org/pipermail/kragen-tol/2010-June/000919.html. Modern manufacturing techniques, while not necessary, should make this much easier than before.

Benefit: I would be the first person to construct a fully-programmable mechanical computer. Nobody has ever done it. Seven generations later, the dream of Babbage, patron saint of irascible eccentrics everywhere, would finally become reality. Zuse's Z1 and the Analytical Engine are the closest anyone has ever come, and the Z1 not only failed to work reliably, but it lacked control flow; while the Analytical Engine has not yet been built, because the cost of Babbage's inefficiently- designed mechanisms is astronomical. The dramatic reduction in parts count provided by heightfield LUT mechanism makes general-purpose mechanical computation feasible for the first time in history, if it works. It would change people's conception of the achievable and demonstrate conclusively that the missing factor that delayed automatic computation for over a century until the 1940s was not in manufacturing technology or in materials science, but merely in our motivation and our logical understanding of the nature of computation.

AOWTBITF: 1024 hours. **SNSTWBU**: construct a mechanical 4-bit multiplier, 64 hours.

Bicicleta

Summary: construct an interactive IDE for Bicicleta, my ς-calculus-based purely-applicative programming language, that other people can practically use.

Benefit: many of the benefits sought by projects like Bret Victor's "learnable programming" and Jonathan Edwards's Subtext are much more achievable in the ς-calculus than in languages based on the λ-calculus or ALGOL; the resulting programming environment will provide unprecedented power to expert programmers while also

being unprecedentedly accessible to novice programmers, replacing Excel for many purposes. Consequently, if the platform is developed to the point where people use it, there will be many more programmers, and they will program many more things; and platforms like Android that allow their users to program will gain an advantage over platforms like iOS that do not; additionally, people adopting Bicicleta instead of Excel will substantially weaken Microsoft's market power.

AOWTBITF: 512 hours. **SNSTWBU**: get a DHTML-based interpreter running, however slowly, with live display of results, 16 hours.

Telecommunication using clouds (of water vapor, in the sky)

Summary: reflect lasers off clouds in the sky for long-distance communication; probe the available bandwidth.

Benefits: This seems like the most practical way to establish long-distance high-speed data links without onerous licensing restrictions, extremely low bandwidth, or need for massive amounts of fixed capital that introduce single points of failure. It may be possible to reach ranges of up to 1000 km at megabits per second. If it works, it reduces the investment needed for intermittent high-speed round-the-world communication to some 20 ground relay stations.

AOWTBITF: 4096 hours. **SNSTWBU**: get laser communication working in my living room with laser pointers reflected off my living room wall, 16 hours. Maybe start by getting L's oscilloscope back from D.

Automatically-controlled low-temperature hot-water tanks

Summary: build a slightly more sophisticated microcontroller-based solar hot-water system using simple insulated-back plastic flat-plate solar collectors with two-sun or three-sun illumination from flat aluminum reflectors and multiple valve-controlled superinsulated hot-water tanks.

Benefits: dramatically cheaper and safer domestic hot water. Most of the expense of domestic hot-water systems in general, and solar hot-water systems in particular, is a result of the inefficiently high temperatures at which the water is maintained, the even higher temperatures reached by flames and electric heating elements, the high hydrostatic pressure within the tank, and the primitive control systems still in common use in hot-water heaters. A microcontroller-based system can maintain water temperatures in a safe range that can be contained with inexpensive bacteriostatic materials rather than expensive materials such as stainless steel. A transition to systems like the one described here would eliminate a significant percentage of world marketed energy consumption, while providing on-demand hot water --- one of the greatest luxuries I have ever experienced --- to a much greater fraction of the population. In http://lists.canonical.org/pipermail/kragen-tol/2007-December/000080875.html I talked about climate control a bit, but mostly in the

context of cooling things down.

AOWTBITF: 256 hours. **SNSTWBU**: get that solenoid-driven valve I salvaged from the discarded washing machine running off the relay I salvaged from the discarded microwave oven, 8 hours.

Ghettobotics

Summary: write a manual for bootstrapping a self-sustaining electronics lab from the municipal waste stream. **Benefits**: ending the perception that electronic gadgetry is something that only the rich and the Chinese can make. Improving my electronics skills to the point where I can solve problems of substantial commercial interest.

AOWTBITF: 2048 hours. **SNSTWBU**: write a working Tinkerer's Tricorder program for Arduino. No, wait, buy another Arduino: 4 hours.

Automatic dependency-driven recomputation

Summary: build a prototype of a rearchitecture of the personal computer platform around the caching of computational results, as described in http://lists.canonical.org/pipermail/kragen-tol/2012-July/000963.html. Related work includes Meteor and http://facebook.github.io/react/, with comments in https://news.ycombinator.com/item?id=5789055. **Benefits**: substantially simpler and more efficient personal computing, which means both that it can run longer on batteries and that it's more practical to experiment with alternative ways of doing things.

AOWTBITF: 16384 hours. **SNSTWBU**: build a chat app on Meteor, 8 hours.

Improvements on PWM

Summary: make improved approaches to PWM practical. PWM, or "pulse width modulation" is a popular method of providing varying output power from a circuit without increasing the circuit's power dissipation enormously; its use for controlling motors, LEDs, and incandescent lights is extremely widespread, and it can also be used to provide a "virtual analog output" from a digital output, most notoriously in Arduino. But it has serious drawbacks: heavy harmonic distortion in the output signal, plus time-domain artifacts that often have quite visible effects, can damage equipment and degrade quality of operation in other ways, such as undesired stroboscopic illumination. Four sometimes-better methods that often go unused because of a lack of good examples and libraries (aside from Don Lancaster's Magic Sinewaves) are dithered PWM, PDM (pulse density modulation), rational-approximation PWM, and PWM for less-significant bits combined with a different approach, such as an R-2R DAC, for more-significant bits. **Benefits**: nobody else will blow out tweeters like I did, and taillights and fading power LEDs will stop doing that annoying flickery thing.

AOWTBITF: 64 hours. **SNSTWBU**: try rational-approximation PWM for audio output on the Arduino and write up my results, 8 hours.

Free software is like owning your own home

Summary: essay advocating a new metaphor for advocating free software. Richard Stallman's standard similes liken using proprietary software is like being enslaved, collaborating with enemy occupiers, or breaking promises. These are not accurate reflections of the modern computing environment, where using proprietary software generally does not require agreeing to NDAs and is substantially less disheartening than living in captivity performing forced labor. Consequently, many people reject Stallman's ethical vision as out-of-touch, and unfortunately, the free-software movement as a whole. A better metaphor is that free software is a home you own yourself, or lease long-term rather than renting on an at-will basis from a landlord: it provides you with better security, greater privacy, and greater individual autonomy, at the cost of being responsible for the maintenance of the thing, whether DIY or outsourced. **Benefits**: a new and more comprehensive vision of the importance of freedom of software that, I hope, provides a more nuanced understanding of the serious issues at stack, and which will resonate better with the current public.

AOWTBITF: 16 hours. **SNSTWBU**: write an outline, 1 hour.

The cheap-junk laser display and camera

Summary: Draw pictures on the wall, in the dark at least, with a laser pointer and some speakers; and use the same low-cost apparatus, plus a photodiode to capture the time-domain reflectance signal, to capture reflectance-range images. Described in some detail, except for the camera part, in http://lists.canonical.org/pipermail/kragen-tol/2010-July/000922.html. **Benefit**: high-resolution big-screen displays for US$10 for everybody else. Awesome show-and-tell device for my living room and get-togethers.

AOWTBITF: 256 hours. **SNSTWBU**: hook up a spinning mirror to a motor, bounce a laser pointer off it, and modulate the laser to make a *one-dimensional* pattern and measure its response time, 32 hours.

The user interface as a real-time program

Summary: essay and proof-of-concept software that uses hard-real-time software techniques to ensure that a layer of the user interface remains *always* responsive, regardless of machine load. **Benefits**: you can immediately fix even badly overloaded servers, using personal computers becomes much more pleasant, and even machines with little RAM and CPU become highly usable, allowing much longer battery life, Kindle-style, for general-purpose applications.

AOWTBITF: 256 hours. **SNSTWBU**: some kind of real-time GUI for a Linux running under a real-time hypervisor (e.g. L4Linux, RTLinux), 64 hours.

A language for real-time programming

Summary: Develop a language that permits the verification and automatic satisfaction of maximum-time and maximum-space

properties for real-time embedded software. Some discussion of requirements at http://lists.canonical.org/pipermail/kragen-tol/2012-January/000943o o.html. **Benefits**: C is still the lingua franca for real-time and embedded programming, but it falls far short of what can be achieved, leaving a lot of work to the programmer that could be done by the computer. This language would do that work for you, allowing an improvement in the productivity of real-time programming comparable to the productivity improvement provided by very-high-level languages like Python or Matlab for software that is primarily computational rather than reactive in nature; this would dramatically extend what amateurs can do with systems like Arduino, where amateurs can easily do simple things but rapidly hit a glass ceiling when going further requires them to understand issues like stack-heap collisions, nondeterministic interrupt response times, shared-state concurrency, and scheduling.

AOWTBITF: 2048 hours. **SNSTWBU**: a minimal compiler that can provide time and space guarantees for a tiny concurrent Actors language, with compilation to Arduino, 32 hours.

Natural language is a digital phenomenon

Summary: an essay describing the difference between analog and digital representations of information (for computation and communication), and explaining the obvious (to me) and surprising (to many people) conclusion that natural language (what we use for verbal communication) is a digital representation of information, not an analog one; and explaining the importance of this fact in human history up to the 20th century.

As the Wikipedia entry for "Digital" says, "Although digital signals are generally associated with the binary electronic digital systems used in modern electronics and computing, digital systems are actually ancient, and need not be binary or electronic. [For example,] Written text in books (due to the limited character set and the use of discrete symbols - the alphabet in most cases)".

Benefits: hopefully people will finally understand this and stop pissing me off by talking about "analog books", as if there is such a thing. More seriously, hopefully people will understand the distinction between "digital" and "computerized", which will improve their ability to predict and deal with 21st-century technology.

AOWTBITF: 8 hours. **SNSTWBU**: an outline, ½ hour.

aaronsw (as Vincent)

Summary: write my long-overdue eulogy for Aaron Swartz, quoting Don McLean's song "Vincent", which is painfully apt. **Benefits**: I will pay some of my karmic debt to my strange friend and painfully missed mentor, and maybe I can stop crying about his suicide. I mean, shit. It's been almost five months and I can't write even this without tears in my eyes.

AOWTBITF: 4 hours. **SNSTWBU**: a first draft, 1 hour.

Phyle sousveillance

Summary: an essay exploring 21st-century approaches to personal

safety and crime, taking as given that nation-states are becoming less effective and more corrupt, and may hollow out dramatically (to inflict one of John Robb's neologisms on you), while the available alternative approaches are exploding. (The term "phyle" is from Stephenson's *The Diamond Age*, but while I think "phyle" is a useful concept, I don't share Stephenson's enthusiasm for violence. I think "sousveillance" is a neologism due to Steve Mann, but Howard Rheingold has also popularized it.)

Benefits: maybe some of the approaches I propose will be practical to implement, and discussion will show some of them to be socially beneficial, and maybe other people will point out that some of them are not workable. Then we can see about trying some of them out.

AOWTBITF: 8 hours. **SNSTWBU**: an outline, 1 hour.

Queer numbers

Summary: some years ago (XXX include link), I proposed a way to produce stable identifiers for individual paragraphs of a changing document, which I called "queer numbers". There is now plenty of data easily available to evaluate the effectiveness of my proposed algorithm. I should do the experiment and publish the results.

Benefits: if it doesn't work, it won't waste anybody's time any more. If it does work, future hypertext systems can incorporate queer numbers to enable robust fine-grained hyperlinks and transclusion.

AOWTBITF: 16 hours. **SNSTWBU**: politely crawl the history of a single Wikipedia article and try the algorithm out on it, 4 hours.

File similarity

Summary: some years ago, I proposed some general algorithms for efficiently finding files with textually similar contents. Current thoughts at http://lists.canonical.org/pipermail/kragen-tol/2010-December/000o0931.html. **Benefit**: substantial improvements in data compression, spam filtering, genomics, virus detection, intrusion detection, and queer numbers, if it works.

AOWTBITF: 64 hours. **SNSTWBU**: try the simplest possible implementation and see if I can get it to work, 4 to 16 hours.

Set-valued bloom filters

Summary: compare the generalization of bloom filters I developed in 2006 or 2007 (XXX include link) to signed-hash full-text indices (XXX is that the right term?), comparing their performance rigorously enough to publish the result as a peer-reviewed paper. **Benefit**: other people would be more likely to be able to find out if this data structure is useful for their purposes, and it might produce marginal improvements in the performance of full-text search engines. Also, I'd have another academic publication.

AOWTBITF: 128 hours. **SNSTWBU**: contact a researcher in the area to see if they'd be interested in helping to guide me through the publication process, 4 hours.

The post-HTTP web

Summary: in 2006, I wrote "What's Wrong With HTTP" (XXX include link) but I never published the promised follow-up essay

describing how to solve the problems. There are now lots of systems in the field demonstrating pieces of the solution. I should rewrite the essay and publish it at last.

AOWTBITF: 16 hours. **SNSTWBU**: find the draft I wrote in 2006, 2 hours.

The colectivos app

Summary: I want the Guia "T", the standard guide to Buenos Aires's bus system, in a free-software Android app. Except that it can be much, much better, because people can contribute. **Benefits**: I'll be able to travel around the Capital more easily. So will other people using free-software Android phones. There will be a wealth of free bus traffic data if the app becomes popular.

AOWTBITF: 128 hours. **SNSTWBU**: get an Android phone.

A free-software predictive input method for Android

Summary: I want something as good as SwiftKey, but free software. Initial thoughts at http://lists.canonical.org/pipermail/kragen-tol/2012-July/000961.html; more details at http://lists.canonical.org/pipermail/kragen-tol/2012-July/000965.html. **Benefits**: free-software Android on common (non-QWERTY) devices would become practical for writing. I'd have a really kick-ass program to show off for job interviews.

AOWTBITF: 256 hours. **SNSTWBU**: get an Android phone.

"Pick activism tactics as if they might work"

Summary: essay advocating care in the choice of activism tactics, in particular condemning the widespread current practice among Democrats in the US of calling for boycotts against people or groups for advocating unpopular political positions, but more broadly, discussing common unintended consequences of poorly-thought-out political advocacy. **Benefits**: my aunt Jessie has already blocked me on Facebook for advocating this position, and I can expect to offend many more people, but explaining the idea in a way that isn't personally directed at a particular other person should have a better persuasion-to-offending ratio. My earlier clumsy advocacy of this position on kragen-tol (XXX include link) provoked some thoughtful discussion. Perhaps someone will even persuade me I'm wrong.

AOWTBITF: 8 hours. **SNSTWBU**: outline, 1 hour.

Binate

Summary: implement Binate, the database query language based on binary relations. XXX include link. **Benefits**: Binate is designed to support live feedback on queries as you're constructing them, which will make fans of Bret Victor happy, and perhaps can woo away some users from Excel; it lets you write common queries in five or six words that would take five or ten lines of SQL; and it is dramatically better at abstraction than SQL is, so you can avoid

writing the same thing over and over again, the way you must in SQL.

AOWTBITF: 128 hours. **SNSTWBU**: a Binate interpreter that lets you interactively query MySQL databases from a browser with Comet, 16 hours.

Suffix-array construction

Summary: write a full-text search engine using one of the linear-time suffix-array construction algorithms discovered in the last decade. **Benefits**: practical full-text substring and regexp search on the desktop.

AOWTBITF: 64 hours. **SNSTWBU**: an in-memory implementation of one of them, 8 hours.

FeML

Summary: implement the FeML minimalist programming language (XXX include link). **Benefits**: the safety of ML, the flexibility of Python, the speed of C, if it works.

AOWTBITF: 256 hours. **SNSTWBU**: a FeML interpreter in OCaml, 16 hours.

Matchscheme

Summary: a Scheme that gets OO and conditionals from built-in pattern-matching, in email with Darius Bacon. **Benefit**: an interesting bottom layer for a language stack.

AOWTBITF: 32 hours. **SNSTWBU**: find the old email thread, 4 hours.

Constrained image approximation using automated image quality assessment and AI

Summary: use a model of the human visual system to guide general AI algorithms in constructing images. Initial thoughts at http://lists.canonical.org/pipermail/kragen-tol/2012-April/000949.html. **Benefit**: a huge range of visual-artistic possibilities (fake pencil portraits, ASCII art, mosaic designs, stencil designs, texture synthesis to match hand-drawn line art, shadow painting), and dramatically better image compression.

AOWTBITF: 1024 hours. **SNSTWBU**: read Taylor's thesis thoroughly, 32 hours.

Polar flutterwumpers

Summary: build a machine that moves physical objects to precise positions relative to each other using only circular motions, thus avoiding the need for precise slides, low-backlash gearboxes, lubrication, and other mechanical pitfalls of linear motion. Initial math at http://lists.canonical.org/pipermail/kragen-tol/2012-April/000956.html. **Benefit**: printers, including 3-D printers, with much simpler hardware, if it works.

AOWTBITF: 512 hours. **SNSTWBU**: a 3-D simulation in JS on a web page, 8 hours.

Backtracking HTML templating

Summary: a new way to write low-logic HTML templates that actually simplifies the task of HTML generation beyond what we had in 1994 with server-side includes. Description at http://lists.canonical.org/pipermail/kragen-tol/2012-April/000951.html. **Benefit**: slightly easier templating, especially of HTML, for extra-difficult cases. Something cool to talk about in job interviews.

AOWTBITF: 32 hours. **SNSTWBU**: a minimal, demoable implementation, 8 hours.

Parser generator with code reuse

Summary: a parser generator that comes with a large host-language-independent library of syntactic features that you can mix and match to describe the language you want to parse, which is impossible with commonly-used parsing algorithms like LR and LL. Some notes at http://lists.canonical.org/pipermail/kragen-tol/2012-April/000953.html. **Benefit**: developing new language syntaxes and parsers would be really easy, due to the parser generator already having most of them in its library, as would developing new interpreters and compilers (in new languages) for existing languages that have existing parsers using this system. World fame. Unless nobody uses it.

AOWTBITF: 4096 hours. **SNSTWBU**: extend peg-bootstrap or one of Darius's libraries to produce reasonably efficient parsers, and abstract out the semantic actions, 32 hours.

Radix-sorting rational numbers

Summary: write up my algorithm for representing rational numbers so they can be radix-sorted, described at http://lists.canonical.org/pipermail/kragen-tol/2011-October/000942.html, such that it could be published in a peer-reviewed venue. **Benefit**: a potential academic publication of little importance.

AOWTBITF: 64 hours. **SNSTWBU**: write an implementation and characterize its behavior, 8 hours.

Very-low-bandwidth speech-formant codec

Summary: encode sampled speech in real time by estimating formant center frequencies and widths and transmitting those, entropy-coded to reduce bandwidth below 1kbps, perhaps as low as 500bps. Speculated on in http://lists.canonical.org/pipermail/kragen-tol/2010-March/000911.html. **Benefit**: if it works, which is unlikely, I could push real-time speech transmission into 500-bit-per-second channels, an unprecedented feat. However, this is basically only useful if you have a low-latency communications channel between 500 and 1000 bits per second; there are already codecs that work at 1000 bits per second. It would be pretty awesome to be able to say I was the guy that achieved this.

AOWTBITF: 1024 hours. **SNSTWBU**: learn enough about DSP to write working code to find the centers of formants (and plot and play them), 32 hours.

Fix one of the JS Markdown libraries

Summary: there are two widely-used libraries in JS for rendering Markdown to HTML. Both are buggy, and neither of them supports inline HTML. Fixing one of them to handle safe HTML tags is a must. **Benefit**: every Markdown-using JS site in the world would be able to handle safe HTML tags. I'd become famous, and in any future job interview, I have a good chance of telling them their site is already running code I wrote.

AOWTBITF: 32 hours. **SNSTWBU**: write a simple test case and make it fail, 2 hours.

Update yamemex

It needs to be fixed to work with current versions of its dependencies, it needs to publish and sync, and it needs a Firefox version. **Benefit**: my bookmarks from the last several years will be shared, people will be able to see what I'm doing, and I'll have all the bookmarks; and maybe other people will start using it too.

AOWTBITF: 32 hours. **SNSTWBU**: update dependencies, 8 hours.

Topics

Query evaluation with interval-annotated trees over sequences

Kragen Javier Sitaker, 2019-08-30 (updated 2019-09-03)
(30 minutes)

Consider queries of the following forms:

```
select * from foo where bar between 80 and 101;
select * from foo order by bar limit 10;
select * from foo order by abs(bar - 13) desc limit 10;
select * from foo where bar between 80 and 101 and baz between 2 and 3;
select * from foo order by (bar-10)*(bar-10) + (baz-2)*(baz-2) desc limit 6;
```

Conventional SQL indices permit answering the first three of these with good efficiency, but not the others; conventional SQL query optimizers do not produce a reasonable query plan for the third given an index on foo.bar, although it is in theory feasible. Usually queries like these are answered with K-d trees or, in the last case in case of high dimensionality, ball trees, but these structures occupy significant space, and constructing and updating them is time-consuming.

Such queries are unfortunately common in applications like geographical maps, as described for example in Fast geographical maps on Android (p. 455).

An alternative that is somewhat more flexible is to construct a tree of substrings over an existing file of records, annotated with intervals describing the data in each substring.

The interval-annotated-tree data structure

(There is a standard data structure called an "interval tree" which is not the same thing, thus the awkward name.)

The underlying data to be queried is a sequence or "file" of records, which have some attributes. The tree is a sort of index for some of these attributes; each node of the tree pertains to some contiguous substring of the overall record sequence, and is annotated with the minimum and maximum value for each indexed attribute that occurs within the subsequence. The substrings all begin and end on record boundaries. Leaf nodes contain pointers into the sequence of records, indicating the start and end of the substring to which they pertain; internal nodes, instead, pertain to the union of their child nodes, which are disjoint and consecutive. The root node of the tree pertains to the entire sequence.

Different branching factors may be appropriate in different situations; if memory is random access, the structure is in theory fastest with a branching factor of 2, but with a memory hierarchy, a larger branching factor is probably better.

If there is some kind of reasonable locality within the existing sequence, this structure can perform as well as a K-d tree, at a much

lower space cost (since its leaves contain merely pointers into the sequence of records). If there is not, it should perform about as well as brute force.

Interval query

To find all the records within a given multidimensional interval — a criterion of the form attr0 between min0 and max0 and attr1 between min1 and max1 and attr2 between min2 and max2... — we do a tree search from the root, pruning a node from the search when the interval annotation on that node can show that none of the records in its substring of the sequence can match the query. In the case where the query is on a single attribute and the records are sorted by that attribute, this will take O(lg N) time. So, too, if the records are sorted in the order of a traversal of a K-d tree, and the query is on a subset of the attributes indexed by that K-d tree, even if the branching factor is not the same.

In some cases it may make sense to also halt the traversal when it can be determined that *all* the records within a substring match the query interval, for example if only a count of the matching records is required.

Altering a record in the sequence requires updating as many as all of its ancestor nodes. Appending a record to the end requires creating or updating as many as all of its ancestor nodes, and, as with a B-tree, potentially deepening the tree and adding a new ancestor node when the root node overflows.

This generalizes, of course, to the case where leaf nodes contain intervals rather than point-values of attributes, but I think a standard interval tree (with its sorted lists of intervals crossing each node's partition) will perform better in that case, perhaps much better.

A simple example

Suppose we have a branching factor of 8, a leaf size of 32 records, a total of 16,777,216 records of 64 bytes each, two perfectly-uniformly-distributed 32-bit attributes indexed, and the data arranged according to the traversal of a 2-D K-d tree on those two attributes. The data amounts to one gibibyte. There are 524,288 leaf nodes and 599,187 tree nodes in total distributed across 8 tree levels, including a root node with only two children. The pointers in the tree nodes can be all entirely implicit as long as we don't have to insert and delete in the middle of the sequence, even if we mutate or append records to the end of the sequence, so each tree node can be stored in 16 bytes (4 bytes for the minimum X, 4 bytes for the minimum Y, etc.)

So the tree occupies only 9,586,992 bytes, under 0.9% of the data size. It can be built in time linear in the data size — in a single sequential input pass outputting tree nodes to 8 sequential buffers, or in 8 sequential passes reading from a single sequential input stream and writing to a single sequential output stream.

A tree traversal for a two-dimensional interval that includes only a single record will prune all nodes but one at each recursion level, so it will visit the root node, its two children, and 8 nodes at each of the other six levels, for a total of 51 tree nodes, then the 32 records belonging to the leaf. If we consider 4096-byte memory pages, the top four levels can be in a single memory page, and then each of the

other four tree levels will require reading a single page; the 32 records of the leaf node will also be in a single page. This puts the total read cost of the query at six pages.

Suppose the query is instead for an interval similarly tightly bounded in one attribute but open in the other. This interval will only prune 6 of the 8 children of any given internal node, so it will visit, say, the root node, its two children, and then at the other six levels, 4, 8, 16, 32, 64, and 128 nodes, respectively, and then 256 leafnodes, which will be in 256 separate pages. The top four levels are again in a single page, while we can suppose that all the other nodes are in different pages, so we have 1 + 16 + 32 + 64 + 128 + 256 pages: 497 in total, a total of just over 2 megabytes out of the 1.075 gigabytes of data. Still, to answer this query from 8-ms spinning rust, you would need almost four seconds.

Actually, though, that's overly pessimistic: the groups of 8 nodes we are examining at each level are siblings, and so they are presumably contiguous. So in fact it's 1 + 2 + 4 + 8 + 16 + 256 pages. Adding another level that includes the actual values of the attributes for the records would cut that final 256, and thus the query time by a factor of 8, but bloat the index from 0.9% to 25% of the size of the original data.

At the other extreme, suppose we do the traversal with an interval that includes all the nodes. This is simply a tree traversal that never prunes, so it just has to traverse an extra 0.9% amount of index data as it sequentially examines the gigabyte of records.

As explained above, updating or appending a record just involves updating or creating up to its 8 ancestor nodes; however, if the update results in the record being out of order, it may degrade the index's query performance thereafter. 8 levels of tree is sufficient up to four gigabytes, at which point the tree deepens and it becomes 9.

A tiny sketch of building such an index with numpy

I hacked this out quickly just in order to get some kind of performance measurement; it could obviously be cleaned up. I kept the data size super small because I was plotting things and, although numpy can deal with tens of millions of items with no problem, matplotlib falls over. Obviously in real life you would need some provision for non-power-of-two sizes, and probably appending records as well, and you wouldn't want a special-case array for the leafnode annotations.

```
x = rand(2**17)
x.sort()

leafnodes = x.reshape((len(x)//32, 32))
lmin = leafnodes.min(axis=1)                    # leafnode min/max
lmax = leafnodes.max(axis=1)

imin, imax = zeros(lmin.shape), zeros(lmax.shape)
size = len(lmin) // 2

imin[:size] = lmin.reshape((size, 2)).min(axis=1)
imax[:size] = lmax.reshape((size, 2)).max(axis=1)
```

```
pos = size

while size > 1:
    size //= 2
    imin[pos:pos+size] = imin[pos-2*size:pos].reshape(size, 2).min(axis=1)
    imax[pos:pos+size] = imax[pos-2*size:pos].reshape(size, 2).max(axis=1)
    pos += size
```

Building the index this way on my laptop takes about 3–5 milliseconds. Taking precautions to limit the load on matplotlib, increasing `x` to 2^{24} items (128 mebibytes) slows it to 227 ms one time, 194 ms another time, 182 ms a third time. Increasing it to 2^{26} items, I accidentally crashed my laptop trying to re-evaluate the IPython cell (thus resulting in two half-gibibyte arrays in memory at once); sorting it took 10.8 seconds; but building the index tree only took 892 ms.

Ordering and reindexing flexibility

The interesting thing here is how the indexing data structure is valid entirely independent of the record ordering. For some record orderings, it reduces to brute force, but it doesn't give incorrect results, and the extra cost is mild.

The above case is precisely equivalent to the K-d tree whose traversal order we suppose the records are encountered in, with some of its internal nodes purely implicit. But with some mild degradation, amounting I think to a small constant factor, the same index data structure would accelerate such queries on data that is ordered according to a K-d tree whose node boundaries were not perfectly aligned with the node boundaries of the index tree.

If the tree traversal is done in a Hilbert-curve order, so that adjacent nodes in the serialization are also adjacent in the K-d space, you gain about a factor of 2 in the case of such misalignments. I'm not sure if this is at every tree level or as a constant factor of about 2 overall.

Other traversal orders may provide better performance for some data distributions; for example, an approximate TSP solution might be better at keeping the intervals smaller, and thus permit better pruning. Some applications might even be able to get away with doing the simplest thing — leaving the records purely in insertion order --- because the insertion order already had enough locality in the relevant attributes to make queries adequately efficient. For example, in a zooming user interface, most queries for drawable objects are restricted to a particular (x, y, size) interval, and generally when you create new objects, you create them in a small area in a small range of sizes.

If the records are instead sorted in a lexicographic order by some sequence of attributes, such as (x, y), then the performance characteristics are equivalent to a normal sorted SQL index: criteria `x=x0` or `X=x0 and Y=y0` are evaluated in logarithmic time, while a bare criterion `Y=y0` is nearly as slow as sequential search. Again, this happens without changing the definition of the index tree, just rebuilding it for the new sort order.

Suppose you instead append some new random records to the end of the record sequence. As discussed above, this means that most queries will have to traverse not only their normal path but also much of the path down to these new update records, in case one of them contains a relevant record; the standard index update takes care of this automatically. But until you have a whole leafnode full of extra records, the number of extra nodes each query needs to visit is just a single linear path — 8 nodes in the above example. Each additional leafnode normally adds just a single additional node, for a total of 9 nodes in the above example, or occasionally 10 or even more rarely 11.

This is recognizable as the standard database approach of having an "update file" that every query must examine sequentially in addition to its usual index traversal, but it is in some sense simpler — the index structure handles it automatically. In a sense this is just an extension of the earlier-described property, that the same index structure is valid for different sorting orders — in this case, we have different sorting orders within the same file, one part of the file sorted in a useful way, while another part is unsorted and requires a sequential search.

We can extend this to a whole "LSM-tree" or "log-structured merge tree" approach like the one used by LevelDB or Lucene: if you have 8,388,608 sorted records, followed by 4,194,304 separately sorted records, followed by 2,097,152 sorted records, then most queries will do no pruning in the first few levels of the tree search, then doing a search within each segment in the usual efficient fashion. At any time if we we atomically replace any substring of the file with a reordered version of itself, as well as the relevant index nodes, the system remains valid, but potentially handles queries more efficiently.

It's often advantageous to handle record updates as well as insertions through such an update file, because it avoids spreading the updates all over the file and frustrating early pruning during query evaluation. To strictly maintain the validity of the index in such a case, you need to somehow null out the outdated versions of the records, replacing them with some kind of N/A or NaN or broken-heart value, one which participates in the min and max computations by just being left out. This might require also maintaining explicit rather than implicit record counts in tree nodes. Upon nulling out a record in this way, you can either update its ancestor nodes in the index — preserving the index's precise nature — or you can leave the index untouched, so that the index is only a conservative approximation of the actual data.

The flip side of this ordering-indexing orthogonality is that you can rebuild the index with different parameters — a different leaf-node size, branching factor, or indexed set of attributes, for example — without sorting the records again.

Conservative approximation of queries

To answer a query such as

```
select * from foo where (x-10)*(x-10) + (y-2)*(y-2) < 100;
```

we begin by constructing a conservative approximation

```
where x between 0 and 20 and y between -8 and 12;
```

which can potentially be answered efficiently by this index structure. An alternative conservative approximation with better precision would be

```
where x between 0 and 20 and y between -6 and 10
or x between 2 and 18 and y between -8 and 12;
```

or

```
where x between 0 and 2 and y between -6 and 10
or x between 2 and 18 and y between -8 and 12
or x between 18 and 20 and y between -6 and 10;
```

Records returned by the conservative approximation must then be tested against the precise criterion.

Alternatively, instead of testing the multidimensional interval covered by each index node against such a multidimensional interval, you might be able to do better by testing it against the original precise criterion, using interval or affine arithmetic.

Evaluating nearest-K queries

Consider again my introductory example

```
select * from foo order by abs(bar - 13) desc limit 10;
```

The standard way of evaluating such a query, lacking a functional index on abs(bar - 13) — which won't work for abs(bar - 2), for example — is to compute a sort or partial sort over all the records, which involves examining all the records' bar attribute at least once.

You could imagine doing this query with a series of queries like the following:

```
select abs(bar - 13) as k, * from foo where k < 1 order by k desc limit 10;
select abs(bar - 13) as k, * from foo where k < 2 order by k desc limit 10;
select abs(bar - 13) as k, * from foo where k < 4 order by k desc limit 10;
select abs(bar - 13) as k, * from foo where k < 8 order by k desc limit 10;
```

and stop once you got enough records from one of the queries. Each of these queries can be efficiently evaluated by the index structure described above, repeating the same tree traversal as the previous query, but pruning perhaps fewer nodes.

A more efficient approach is to do the traversal once, storing all the pruned nodes in a priority queue according to *by how much* they were excluded. This allows you to revisit enough of them to ensure that you have found all the points within the radius at which you found the Nth-best point you've found so far (N=10 above). That point

won't necessarily be in one of the nodes whose intervals are nearest to the query point, either in upper bound or lower bound.

Evaluating minimum, maximum, and top-N queries

Consider these queries:

```
select min(x) from foo;
select min(x) from foo where y = 41;
select min(x) from foo where y between 38 and 42;
select * from foo where y between 38 and 42 order by x desc limit 5;
```

If you have an interval-annotated tree built for attribute `x`, and you're updating it precisely rather than conservatively, the first query can be answered instantly — it's stored in the root node of the tree. The second and third queries are not instant, but if there are contiguous ranges of records all matching the selection criterion, it is unnecessary to examine each record individually; you can use the min(x) annotation from their tree nodes instead.

The fourth query is somewhat more interesting, because once we have done enough traversal to define the set of records matching the selection criterion (as a smaller set of tree nodes covering disjoint parts of the file), we would like to return them starting from the largest *x*. Well, we can look at the max(x) annotations to figure out beneath which node in the set the largest x (say 112) is to be found, then recursively trace down through the tree from that node to find some record with x = 112. *Then we remove it from the set*, by removing the tree node we had found it from, adding that node's children, if any, and then recursively repeating that process for the child node by which we found the record; this leaves us in a position to repeat the process four more times. By maintaining the nodes in a priority queue ordered by max(x), we can do this update quite efficiently.

Column-oriented index storage

Above I said that for an index covering two 32-bit attributes, each node will occupy 16 bytes, and in my analyses of locality I assumed they were contiguous in memory. But in the case where a query only needs to examine one of the attributes, it would be better to store the trees for the two attributes separately — the query will need to transfer half as many cache lines from RAM, and perhaps an extra level near the root of the tree will fit into the first page.

Pointer files

In cases where the records are sorted in an inconvenient order, you can do the usual database-index thing of making a sorted file of pointers to the records, perhaps augmented with the sort keys, and build the index tree on that. In the case where you use a simple lexicographical sort, this is just a normal database index, but you also have the possibility of using a different order.

Some explorations of a generalization of

the index structure, and a surprising connection with mathematical morphology

The tree search pruning merely depends on being able to efficiently compute the bounds of the search key in certain substrings of the file; you could imagine using some other structure to accelerate that computation, rather than just a tree for a given set of substrings.

An alternative structure is the preprocessing for the Urbach-Wilkinson erosion algorithm described in Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216): given some string of pixels, you can compute summary strings of maxima of all substrings of 2, 4, 8, 16, 32, etc., pixels, starting at all the possible offsets. Then the maximum of an arbitrary-length substring can be computed in constant time by finding two potentially-overlapping substrings, typically within the same level of summary, whose union is the you want to find the maximum of.

More briefly, say $S_{0,j}$ is pixel j of the string of pixels, and define

$$S_{i,j} = S_{i-1,\,j} \vee S_{i-1,\,j+2^{i-1}}$$

whenever all of the subscripts are within bounds, and $x \vee y$ gives the maximum of x and y. This gives us the N-ary maxima $\vee_j$ of all the size-2^i substrings

$$\vee_{j \in [m,\, m+2^i)} S_{0,\,j} = S_{i,\,m}$$

and from that we can compute the maximum of an arbitrary range of pixels as

$$\vee_{j \in [m,\, n]} S_{0,\,j} = S_{a,\,j} \vee S_{a,\,j+n-m-2^a}$$

where a is chosen such that $n - m + 1 \in [2^a, 2^{a+1})$ to prevent a gap from occurring between the two size-2^a substrings.

If we precompute all the S_i, of which there are a logarithmic number, we can do this in constant time.

Of course all of the above goes *mutatis mutandis* for minima rather than maxima.

An interesting property of this structure is that, without any further metadata, we need only logarithmic time to trace any range maximum thus computed back to one of the pixels where it originated; for example, $S_{4,\,3}$ is either equal to $S_{3,\,3}$ or it's equal to $S_{3,\,11}$, and $S_{3,\,11}$ is either equal to $S_{2,\,11}$ or to $S_{2,\,15}$. By following the trail this way, we can figure out where in the range to find the maximum pixel value. In particular, this allows us to find the runner-up value in logarithmic time: if it turns out that the maximum value came from pixel 12, we can split the range into two halves that exclude pixel 12 and calculate their maxima, one of which is the runner-up.

This is suggestively similar to the query algorithms described above, and as it turns out, we can view the interval-annotated-tree structure we started with as a sort of decimated version of this layer-cake structure, where we only keep $S_{i,\,j}$ if j is divisible by 2^{i+k}, where 2^k gives the number of records in a leaf node, and perhaps if i is divisible by some number — above I used 3. This makes for a much smaller index, but it also makes even the simplest queries take logarithmic time rather than constant time, and it adds a constant factor to logarithmic-time queries. Intermediate points on this time-space tradeoff spectrum may be worth considering. For

example, you could have nodes further up the tree overlap somewhat, without storing every possible offset.

Annotations using lattices and semilattices other than total orderings

The above data structures generalize to general lattices and semilattices. As an example closer to normal databases, you could consider indexing some attribute with small Bloom filters, using bitwise OR between the Bloom filters rather than min or max. Suppose you use 256 bits (16 bytes) and a single hash function:

Records indexed	Expected fullness	False positive probability
1	0.4%	0.4%
32	12%	12%
128	39%	39%
256	63%	63%

So you could skip over about a third of your 256-record chunks after just checking a bit in their Bloom filters, and almost 90% of your 32-record chunks (although there's a conditional probability there that I'm not calculating — if you're looking at a 32-record leafnode at all, it might be because it's, say, one of eight children of a 256-record that had the right bit set, and at least one of those eight children needs to have that bit, and so on average almost two of them will.)

This suggests that aggregating Bloom filters in this way is only useful over a couple of orders of magnitude, and also that you should use a low branching factor like 2 in this case.

I think this gets worse rather than better with more hash functions: false positive probability goes down for single records and small groups, and up for large groups. Consider the case with three hash functions:

Records indexed	Expected fullness	False positive probability
1	1.2%	0.0002%
32	31%	3.1%
128	78%	47%
256	95%	86%

The problem is that Bloom filters are most efficient when they're about half full, but trying to calculate the parent bitvector from the child bitvectors means that they have to be the same size. You could consider giving that up — maybe you use three hash functions, and for 16-record leafnodes, you use 64-bit (8-byte) bitvectors, giving you a 14.9% false-positive probability; while for their 32-record parents, you use 128-bit (16-byte) bitvectors with 14.8% false positives; and for their 64-record parents, 256 bits (32 bytes) and 14.7%; and their 128-record parents, 512 bits and 14.7%; and so on. But then if you delete a record or update it in place, you either have to let the Bloom filters decay a bit, or you have to rehash all 128 of them. At some point you have to stop expanding the filters or deleting a single key ultimately requires rehashing the whole file.

Bloom filters can do some operations that go beyond simple membership testing. For example, if two Bloom filters are built compatibly, you can efficiently compute not only their union (as above) but also the Bloom filter of their intersection. This is potentially useful for joins somehow.

There are also cases where you have honest-to-goodness bitvectors

stored in a database, perhaps in a SAT solver or a database of combinational-logic circuits, and it might be useful to query not only for a particular bit pattern but for things that have supersets or subsets of it set.

Affine-arithmetic nodes

Suppose that instead of annotating each node with just the (min, max) values of each indexed attribute, you annotate it with a coefficient which gives a *predicted* attribute value when multiplied by its offset within that node, and a (min, max) value pair for the *difference* between the predicted value and the real value. If the coefficient is 0, then the (min, max) values are the same as before, but if there's some correlation between the attribute value and the record position (at least locally), this may allow much tighter bounds, and in particular much faster range narrowing — interpolation search on steroids.

(You might want some kind of extended-precision data type to handle the necessary arithmetic on string keys.)

This is quite close to a totally trippy paper from Google Research on training neural nets as database index nodes, whose authors and title I forget.

Topics

Phase change unplugged oven

Kragen Javier Sitaker, 2019-12-15 (0 minutes)

If you want a solar oven, or an electric oven powered by photovoltaic panels, to work at night, you need an energy store. The best approach is probably a phase-change heat reservoir, perhaps with a phase-change temperature somewhere around 200° or 250°. Heat can be moved in and out of the heat reservoir with fan-forced convection.

Topics

Index set inference or domain inference for programming with indexed families

Kragen Javier Sitaker, 2007 to 2009 (updated 2019-05-05)
(27 minutes)

If you infer parametric types for variables used to index arrays (and then for functions that have them as arguments, and functions that call them, and so on), I think you can substantially simplify code that deals with arrays by defining new arrays in terms of pointwise transforms of old ones, and additionally reduce the distinction between functions and arrays. If this information is available reflectively at run-time, you can get some additional power.

Although the idea looks like it might be a bit clumsier than I was thinking at first, I still think it may have some merit.

(Further development of the idea is in A principled rethinking of array languages like APL (p. 1995), from 2015 to 2018.)

Major Problems

I think this needs static domain (i.e. dependent type) inference to work at all (see in the "Efficiency" section) but maybe it's an interesting idea anyway.

I sort of took for granted that you would want C-style curried arrays, where if you have a two-dimensional array a[i][j] you can meaningfully say a[i] and get an array b[j]. But this seems to have led to some extra complexity in the definition of asymmetric-union, in the kinds of domain inference that can plausibly be done in different situations, and even in the notation to define a two-dimensional matrix "narrowed" from some underlying function.

Backus's Matrix Multiply Program

So I've been reading John Backus's 1977 Turing Award lecture paper. (See Why John Backus Was on the Wrong Track (p. 2722) for more commentary.) In it, among other things, he gives this definition in his FP language for matrix multiplication:

```
Def MM = (alpha alpha IP) o (alpha distl) o distr o [1, trans o 2]
```

where (less rigorously than in the paper)

```
: represents function application
< > enclose data vectors
(alpha fn):<z[1], z[2], ...z[n]> = <fn:z[1], fn:z[2], ...z[n]>
(f1 o f2):z = f1:(f2:z)
1:<z[1], z[2], ...z[n]> = z[1]
2:<z[1], z[2], ...z[n]> = z[2]
distr:<<z[1], z[2], ...z[n]>, y> = <<z[1], y>, <z[2], y>, ...<z[n], y>>
distl:<y, <z[1], z[2], ...z[n]>> = <<y, z[1]>, <y, z[2]>, ...<y, z[n]>>
trans:<<z[1][1], z[1][2], ...z[1][n]>,
       <z[2][1], z[2][2], ...z[2][n]>, ...
```

```
      <z[m][1], z[m][2], ...z[m][n]>> =
                                  <<z[1][1], z[2][1], ... z[m][1]>,
                                   <z[1][2], z[2][2], ... z[m][2]>, ...
                                   <z[1][n], z[2][n], ... z[m][n]>>
x is multiplication
[fn1, fn2, ...fnx]:z = <fn1:z, fn2:z, ... fnx:z>
```

IP is his previously given program for the dot product of two vectors (contained in a vector of two items), which reads as follows:

```
Def IP = (/+) o (alpha x) o trans
```

where /fn: = fn:...>>

If you put it all together and squeeze, you get

```
Def MM = (a a((/+)o(a x)otrans))o(a distl)o distr o[1,trans o 2]
```

A Simpler Matrix Multiply Program

So I couldn't help but think, leaving aside all Backus's points about algebraic manipulations of programs and how his unreadable point-free style is so cool — let's go to the other extreme! Wouldn't it be easier if you could instead write this?

```
MM[m][n][i][j] = sum(k) { m[k][j] * n[i][k] }
```

Or, if you use space for procedure application, and curry in the usual Haskell/OCaml way:

```
MM m n i j = sum(k) { m k j * n i k }
```

Or in a Lispy syntax, but still assuming that same kind of currying, so that ((((MM m) n) i) j) is the same as (MM m n i j):

```
(define (MM m n i j) (sum k (* (m k j) (n i k))))
```

Unifying Arrays With Functions

I've thought for some time that it would be interesting to have a programming language where arrays and dictionaries are treated as kinds of functions — they have the distinction that their domains are finite, but that is the case for many functions.

"sum" in the program above is intended to range its dummy variable across all possible valid indices. Evaluating it for some particular i, j pair requires it to interrogate m for its domain — and, more interestingly, (n i) for that i. (What it should do in the case where the two domains are different, I'm not sure; probably it should signal an error by default.)

Given that n has some associated thunk that yields its domain for

such purposes, it's clear to see that the compiler could construct a similar domain thunk for the result of (MM m n), which would merely call n's domain thunk; and for, say, (MM m n 2), given that this reduces to (lambda (j) (sum k (* m k j) (n 2 k))), it can infer that j can range over the domain of (m k j) for some k. (Again, it's not clear what to do in the case that the domains are different for different values of k; if the domain of (MM m n 2) is never requested, the issue doesn't arise, but if it is, it might be a good idea to signal an error by default rather than taking their intersection.)

Domain Inference as Type Inference

This "domain inference" is really just a kind of type inference, so it ought to be possible to do it at compile time. Maybe we could statically infer that MM has a type something like

```
('a:'b -> 'c:'d -> 't) -> ('e:'f -> 'a:'b -> 't)
    -> 'e:'f -> 'c:'d -> 't
```

(I'm assuming here that multiplication has the type 't -> 't -> 't, i.e. it can only multiply two things of the same type. There are other alternatives, but I think that for static inference to work, the compiler at least has to know whether or not multiplication is going to yield something a function that has a domain of its own, and if so, what that range is going to be.)

Doing it statically is a little more trouble than doing it dynamically because you end up with complicated relationships between implicitly-universally-quantified variables, like the above, instead of just some numbers.

Some kinds of control constructs can allow this domain inference to flow through them as well:

```
(define (zeroextend vec zero i)
  (cond ((= i (- (domainbegin vec) 1)) zero)
        ((= i (domainend vec)) zero)
        (t (vec i))))
```

This is a function which, given a function such as a one-dimensional vector, and a zero value to extend it with, makes it into a one-dimensional vector with a domain of indices greater by 1 in each direction. I'm using made-up functions "domainbegin" and "domainend" to return the minimum and maximum+1 values of the domain, which presumably therefore must consist of a single contiguous sequence of integers. This is a useful abstraction to have when, say, implementing numerical simulations with fixed border conditions.

It seems like making the above conditional transparent to domain inference would be pretty hard; you could do partial evaluation and get, say:

```
(lambda (zero i)
  (cond ((= i -1) zero)
        ((= i 4) zero)
```

```
      (t (vec i))))
```

at which point you could deduce that the domain of i was the union of -1:0 (i.e. {-1}), 4:5 ({4}), and 0:4 (the domain of vec), which union comes out to -1:5. Is there an easier way, say, one that could be done at compile time? Even one that required restructuring "zeroextend"? I don't think macros per se particularly help here. Maybe you could say:

```
(define (zeroextend vec zero i)
  (cond ((valid? (vec i)) (vec i))
        ((valid? (vec (+ i 1))) zero)
        ((valid? (vec (- i 1))) zero)
        ; otherwise error:
        (t (vec i))))
```

Termination

I'm not claiming that you can infer which inputs will lead to an infinite loop. That "Total Functional Programming" approach is not something I'm considering here, although it does kind of look interesting.

Narrowing

To make such a system useful, the programmer needs to be able to artificially impose boundaries on functions with much larger underlying domains. For example, you could define the identity matrix in the abstract as follows:

```
(define (identity i j) (if (= i j) 1 0))
```

But, if the matrix multiply routine from earlier is going to insist that its arguments be conformable, you need to be able to wrap it in something to restrict its domain, like this:

```
(narrow (domainbegin m) (domainend m) identity)
```

or even

```
(narrow (domainbegin m) (domainend m)
        (lambda (i) (narrow (domainbegin (m i)) (domainend (m i))
                            (identity i))))
```

or even (in cases where you already have a function handy with the desired range)

```
(narrow m (lambda i (narrow (m i) (identity i))))
```

Inference Rules for Primitives

So if we want to infer the domain for some parameter, and it's being used somewhere deep inside of a context that has some known domain, we can often solve it. Here is a non-exhaustive list for cases of contiguous integer ranges; p is the parameter whose domain we want to infer, and k is a constant.

```
(+ p k) as n:m => p as n-k:m-k
(+ k p) as n:m => p as n-k:m-k
(- p k) as n:m => p as n+k:m+k
(- k p) as n:m => p as k-m+1:k-n+1

(* p k) as n:m => p as n/k:m/k      for k positive  (how do we adjust this when k
o/n is noninteger?)
(* p k) as n:m => p as m/k+1:n/k+1  for k negative  (who cares about the 0 case?)
/, >>, << can be treated as cases of *, I think, although / raises
funny issues about approximate results (which way are they
rounded?)
(& p bitmask) as n:m => p can be any int if the bitmask ensures that
                        the result is in n:m; complicated otherwise
(& bitmask p) likewise
(remainder p k) likewise
(min p k) as n:m => p as n:inf if k < m; complicated otherwise
(max p k) as n:m => p as -inf:m if k >= n; complicated otherwise
(abs p) as n:m => p as -m+1:m if n <= 0 and m > 0; complicated otherwise
```

My hypothesis here is that these rules don't have to successfully analyze every possible program; it just has to be possible to write programs that they can analyze that do what you want, at least most of the time.

Difficulties in Domain Inference

A parameter that is used exactly once in one of the following ways poses no difficulties for domain inference:

- to index a concrete array
- to pass to a primitive function with a fixed domain
- to pass to some other user-defined function as a parameter with a known domain

Parameters that are passed to certain primitive functions may have more restricted domains inferred for them; for example, the domain of (lambda (i) (m (+ i 1)) is the same size as the domain of m, but offset by 1; and (lambda (i) (m (/ i 2))) is twice the size of the domain of m. (And depending on what kind of division that is and whether m's range consists only of integers, it might have twice as many elements.)

Parameters that are not used at all therefore could have any value, which is a problem, in a way. Parameters that are used more than once require deciding what to do in the case of overlapping, but not equal, domains. (Maybe there should be a default, and maybe the default should be to intersect the domains rather than to raise an error.)

This is reminiscent of the restrictions placed by Girard's "linear

logic" and Henry Baker's "Linear Lisp", in which each variable must be used exactly once in a particular path of execution. In this case, some paths of execution (those that depend on the value of the variable itself) may not count, but that's perhaps a detail.

Perhaps the solution, then, is to require or allow explicit duplication of parameters that are used for things with more than one domain, specifying how to resolve differences between the two. For example, you could define a "deltas" function as follows:

```
(define (deltas vec i)
  (- (vec (+ i 1)) (vec (intersecting i))))
```

to specify that the domain inferred for the second "i" is to be intersected with the other inferred domain, rather than raising an error. (I'm not sure that this shouldn't be the default.)

Domain Inference In Conditionals

So if we have (if cond result alt), what do we do?

If the cond tests a parameter we're trying to infer information about, we may be able to infer something about the values that parameter can take in the branch taken. If we have (if (< i 2) (foo i) (bar i)), then we know that in the i<2 case, we're interested in foo's domain, and in the i>=2 case, we're interested in bar's domain. So if foo's domain is 0:10 and bar's is -10:5, the correct domain to infer is 0:5:

- first we construct -inf:2 and 2:inf domains;
- we intersect -inf:2 with 0:10 and get 0:2;
- we intersect 2:inf with 0:5 and get 2:5;
- we union 0:2 and 2:5 and get 0:5.

(This gets tougher if we have ((if (< i 2) foo bar) i).)

Suppose the conditional doesn't test the parameter of interest, though. Now the domain is harder to figure out. Should we intersect the domains computed for the two branches, as if they both got executed (a conservative approximation to the domain)? Should we complain if the domains computed are different (when in doubt, refuse the temptation to guess)? Or should we take the union of those domains (a liberal approximation)?

Asymmetric Union

I recently wrote this C function:

```
// returns an approximation of 256 * sqrt(sqr)
int inline interp_sqrt(int sqr) {
  if (sqr < max_sqrtx256) return sqrtx256[sqr] >> 4;
  int high = sqr >> 8; // we hope high < max_sqrtx256-1
  int below = sqrtx256[high];
  int above = sqrtx256[high+1];
  int spread = above - below;
  int correction = ((sqr & 0xff) * spread) >> 8;
  return correction + below;
}
```

The table sqrtx256[ii] contains (int)(0.5 + 256 * 16 * sqrt(ii)) for values of ii up to max_sqrtx256; that's the rounded value of 256 * sqrt(ii * 256), or sqrt(ii << 8) << 8. For largish values of sqr, it linearly interpolates sqrt(sqr) between two adjacent values from that table, but I found that was too inaccurate for small values of sqr, so I used the table directly for them. (sqrt(ii * 256) is 16 * sqrt(ii).)

(This is all integer math because floating-point is absurdly slow on my CPU.)

So to look at it another way, I constructed two approximations of sqrt:

```
(define (approx1 sqr) (>> (sqrtx256 sqr) 4))
(define (approx2 sqr)
  (let* ((high (>> sqr 8))
         (below (sqrtx256 high))
         (above (sqrtx256 (+ (intersecting high) 1)))
         (spread (- above below))
         (correction (>> (* (& (intersecting sqr) 0xff) spread) 8)))
    (+ correction below)))
```

Now, if sqrtx256 has a known domain, it should be straightforward to infer the domains for approx1 and approx2, given appropriate rules for (>> x constant) and (+ x constant). (In my case, it happened that approx1's domain was a subset of approx2's.) So it would be nice to construct a combined function that used approx1 for arguments within its range, and approx2 when they were outside of approx1's range but within approx2's — a sort of asymmetric union —

```
(define interp_sqrt (asymmetric-union approx1 approx2
                                      (lambda (x) (* (sqrt x) 256))))
```

You can define asymmetric-union, for two functions and a single argument, as follows, in terms of the valid? predicate from the earlier "zeroextend" function:

```
(define (asymmetric-union x y i) (if (valid? (x i)) (x i) (y i)))
```

I'm not sure to what extent you can write an asymmetric-union that works for an arbitrary number of arguments.

Space-Time Tradeoffs as Annotation

In general it's desirable when you can have the actual code to solve a problem in one place, while the optimizations that make it possible to execute the code with adequate speed are somewhere else. SQL databases are probably the most-widely-known example of this; indices make it possible to execute queries at a reasonable speed, but most SQL statements are guaranteed to return the same results regardless of what indices exist, at least if they ever finish. (The ability to replace a table with a view, and then possibly materialize that view, is a more extreme example of the same thing.)

Any kind of separation of interface and implementation gives you this ability to some extent, in the sense that the code that calls the interface will only run fast if the implementation (which is somewhere else) is adequately efficient. But it is often the case that our implementations mix together their essential semantics with efficiency-driven choices of representation and so on, which makes it difficult to determine whether the code is actually correct.

Another common way to separate the optimizations from the semantics is to put the optimizations in the compiler, or even in the CPU. While this is very economically efficient — it allows all programs compiled by the compiler or run on the CPU to get the benefit — it has a certain incentive problem, namely that the guy who thinks his code runs too slowly usually isn't in a position to modify the compiler or the CPU to speed it up.

So recently I was computing, in Numerical Python, a bunch of animation frames of expanding haloes for a game-like program I was writing. (I say "game-like" because it doesn't have a score or a goal; it's a simple sound sample sequencer.) The NumPy code that renders a frame looks more or less like this, although I'm leaving out the stuff that computes the RGBA palette (indexed by 'take' in the last line), interacts with SDL, and handles varying object lifetimes.

```
# inputs are size, max_age, fuzz, age
shape = (size*2, size*2)
cx = cy = size                  # x and y at center
xs, ys = Numeric.indices(shape) # x and y coords of each pixel
(dx, dy) = (xs - cx, ys - cy)   # distances from center for each pixel
rsq = dx*dx + dy*dy        # squared distance from center

max_level = size**2/2 * (1 - (1 - age*2)**2)
density = Numeric.clip(Numeric.absolute(rsq - max_level),
                       0, fuzz**2).astype(Numeric.Int)
colors = Numeric.take(palette, density)
```

Originally I was computing each frame like this on the fly, but I was taking a hit on my frame rate. So I made an object that stored up everything except age, max_level, density, colors, and palette, which involved quite a bit of change to the above code. That worked OK but was still slow, so I precomputed 48 frames the first time any of them was requested, which took a second or two, and then just fetched the precomputed frames from then on.

That was fine, but then I used a larger halo somewhere else and the pause when it was computed was rather dramatic. So I changed the strategy again to compute, then store, each frame as it was asked for; this improved the situation to a slightly perceptible slowdown the first few times the larger halo was used (since only a few of the frames would be requested each time), but introduced a couple of visible levels of indirection into the code. Now the NumericHaloMovie object has a `__getitem__` that renders frames on KeyError, and there's a get_halo_movie function that stores references to the existing NumericHaloMovies and creates new ones when needed. This extra code complicates the task of someone who wants to see where the images are coming from.

In a system where the contents of arrays, the contents of dictionaries, and the return values of functions are defined the same way and accessed the same way, the decision about whether and how a particular function is memoized would not need to be part of that function's actual code. You could simply say somewhere:

```
(declare-concrete rsq density colors)
```

to tell the system to eagerly materialize those functions as concrete arrays. And possibly

```
(memoize frames)
```

to tell the system to materialize items of frames when they are needed, but then remember them. (Maybe. It will require some cleverness to actually make that possible.)

(Of course, now that I write this, I think maybe I should have just written a DictionaryMemoizer and ArrayMemoizer in Python, which, in a way, is just the approach of merely separating interface from implementation. Oh well. Maybe I will.)

My Halo Animation Expressed This Way

Compare to Python code earlier.

```
(define (sq x) (* x x))
(define (colors palette size max_age fuzz age x y)
  (let* ((rsq (let ((dx (- x size)) (dy (- y size)))
                (+ (sq dx) (sq dy))))
         (max_level (* (/ (sq size) 2) (- 1 (sq (- 1 (* age 2))))))
         (density
          (int (clip 0 (sq fuzz) (abs (- rsq max_level))))))
    (palette density)))
(define (discrete-colors palette size max_age fuzz age)
  (narrow-to-int-range 0 (* 2 size)
    (lambda (x)
      (narrow-to-int-range 0 (* 2 size)
                           (colors palette size max_age fuzz age x)))))
; already defined in Numeric, but here it is:
(define (clip min max val)
  (cond ((< val min) min) ((> val max) max) (t val)))
```

The Python code is clearer, I think. Infix syntax helps for numerical things! Also, the two-layer-deep narrow-to-int-range thing is a mess.

However, this looks a lot like what I would write in C, and consequently isn't all that novel; none of the variables (other than discrete-colors) have what you'd think of as a matrix value.

I tried writing a version in which the variables x and y were passed explicitly to each thing that needed them, making it clear that, say, rsq and density varied by pixel, while max_level and palette didn't. So they were arrays, in a sense. But it was far more wordy than this version.

Windows and Sprites

Asymmetric union is exactly what you need for drawing windows on the screen: consider the window to be a function from coordinates to pixel values that is only defined at some pixels. If your system additionally supports domains that are more complicated than simple contiguous ranges of integers, you get the ability to composite sprites and shaped windows.

Efficiency

Of course if you're checking (dynamically or statically) that the parameters you're passing to a function are inside a domain where that function won't violate any array bounds, you can leave off checking for that inside the function. This subsumes a lot of the kind of type checking needed for safety and dynamic dispatch as well.

So suppose you use the dynamic strategy I suggested with the matrix multiply program I had at the beginning:

```
(define (MM m n i j) (sum k (* (m i k) (n k j))))
```

When you are evaluating, say, (MM m n 3 4), you check upon entry to MM that 3 is within the relevant domain, given m and n. Then, in the inner loop, you don't need to do much bounds checking:

- you know 3 is within the domain of m, because that's where the range of (MM m n) came from;
- you know k is within the domains of (m 3) and n, because that's where its values came from.

However, you still need to verify that (n k j) is valid, each time through the loop, because in the dynamic approach, you don't know if each of the (n k) is a different length. I think that this is a serious problem because it means you can't actually find out what the domain of (MM m n 3) is at all.

If you are calling MM from inside some larger loop, you may be able to push the bounds-checking of i and j out even further, using the same approach. For example, if for some reason you were to write

```
(sum i (sum j (MM m n i j)))
```

then, each time through the inner loop, you would know that i and j were within the bounds of (MM m n) because that's where the sum operators are getting them, so you wouldn't need to do any bounds-checking inside at all.

Other Related Stuff

Neel Krishnaswami said, in a thread on Fortress, "Look at Hongwei Xi's work on Dependent ML http://www.cs.bu.edu/~hwxi/DML/DML.html. Checking matrix sizes was the first thing I thought of when I read his work." So far, I haven't.

From

http://www.haskell.org//pipermail/haskell/2004-August/014397.html

Our example is `bsearch', taken from the famous paper "Eliminating Array Bound Checking Through Dependent Types" by Hongwei Xi and Frank Pfenning (PLDI'98). Hongwei Xi's code was written in SML extended with a restricted form of dependent types. Here is the original code of the example (taken from Figure 3 of that paper, see also http://www-2.cs.cmu.edu/~hwxi/DML/examples/)

```
] datatype 'a answer = NONE | SOME of int * 'a
]
] assert sub <| {n:nat, i:nat | i < n } 'a array(n) * int(i) -> 'a
] assert length  <| {n:nat} 'a array(n) -> int(n)
]
] fun('a){size:nat}
] bsearch cmp (key, arr) =
] let
]     fun look(lo, hi) =
]         if hi >= lo then
]             let
]                 val m = (hi + lo) div 2
]                 val x = sub(arr, m)
]             in
]                 case cmp(key, x) of
]                     LESS => look(lo, m-1)
]                   | EQUAL => (SOME(m, x))
]                   | GREATER => look(m+1, hi)
]             end
]         else NONE
]     where look <|
]     {l:nat, h:int | 0 <= l <= size /\ 0 <= h+1 <= size } int(l) * int(h)
]             -> 'a answer
] in
]     look (0, length arr - 1)
] end
] where bsearch <| ('a * 'a -> order) -> 'a * 'a array(size) -> 'a answer
```

The text after `<|' are dependent type annotations. They *must* be specified by the programmer -- even for internal functions such as `look'.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Graphics (p. 3483) (91 notes)
- Programming languages (p. 3656) (47 notes)
- Python (p. 3671) (27 notes)
- Graphical user interfaces (p. 3489) (23 notes)
- Arrays (p. 3326) (17 notes)
- APL (p. 3320) (9 notes)

Enumerating binary trees and their elements

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

Raphael Finkel's book says, "Enumerating binary trees (see Chapter 2) is quite difficult in most languages, but quite easy in CLU." Of course I am not enthusiastic about the idea that CLU has any merits not shared by my own favorite languages, and so I am curious how hard it would be in, say, Python or Scheme.

So, I thought, enumerating the binary trees of a particular size should be fairly straightforward in Python:

```
# With a generator.
def enum_bin_tree(n):
    if n == 0:
        yield None
    for leftsize in range(n):
        for left_tree in enum_bin_tree(leftsize):
            for right_tree in enum_bin_tree(n - leftsize - 1):
                yield left_tree, right_tree

# With a multi-for list comprehension.
def enum_bin_tree_eager(n):
    if n == 0: return [None]
    return [(left_tree, right_tree)
            for leftsize in range(n)
            for left_tree in enum_bin_tree_eager(leftsize)
            for right_tree in enum_bin_tree_eager(n - leftsize - 1)]

# With a simple nested loop.

def enum_bin_tree_simple(n):
    if n == 0: return [None]
    rv = []
    for leftsize in range(n):
        for left_tree in enum_bin_tree_simple(leftsize):
            for right_tree in enum_bin_tree_simple(n - leftsize - 1):
                rv.append((left_tree, right_tree))
    return rv
```

Or in Scheme:

```
(define (enum-bin-tree n)
  (if (= n 0) '(())
      (mapappend (lambda (left-size)
                   (mapappend (lambda (left-tree)
                                (map (lambda (right-tree)
                                       (list left-tree right-tree))
                                     (enum-bin-tree (- (- n left-size) 1))))
                              (enum-bin-tree left-size)))
                 (iota n))))
```

```
(define (iota n) (iotaplus (- n 1) '()))
(define (iotaplus n rest)
  (if (< n 0) rest (iotaplus (- n 1) (cons n rest))))
(define (mapappend fn lst)
  (if (null? lst) '()
      (append (fn (car lst)) (mapappend fn (cdr lst)))))
```

Then I looked at Finkel's pseudo-CLU version; it is 24 lines, compared to 14 for the Scheme version and 6–8 for the Python versions. However, it happens to be very similar to the first (7-line) Python version above; the only differences in the algorithm are the position of the -1 and its use of side effects in place of constructing new tree nodes.

A variant of the approach above can be used to enumerate the sentences of a given length in at least some context-free languages; the tricky part is making sure that the recursion terminates. I think it will work for grammars with no epsilon-productions and no cycles in which a nonterminal can expand to itself A -> A. I'm not quite sure if it can be expanded to include those languages; I'm pretty sure allowing the cycles don't add any power (you can rewrite the grammar to an equivalent grammar without them) but I'm not sure about the epsilon-productions.

Enumerating binary search tree keys

Another example Finkel's book gives, which is perhaps more to the point, is comparing two binary trees to see if their nodes have the same value in inorder traversal. This is very similar to the samefringe problem, in that the recursive formulation of inorder traversal is very straightforward:

```
def inorder_traverse(fn, bintree):
    if type(bintree) is type(()):
        left, middle, right = bintree
        inorder_traverse(fn, left)
        fn(middle)
        inorder_traverse(fn, right)
```

A nonrecursive procedural formulation, on the other hand, is considerably more opaque.

```
def inorder_traverse_nr(fn, bintree):
    stack = [("node", bintree)]
    while stack:
        action, arg = stack.pop()
        if action == "node":
            if type(arg) is type(()):
                left, middle, right = arg
                stack.push(("node", right))
                stack.push(("visit", middle))
                stack.push(("node", left))
        else:                           # action == "visit"
            fn(arg)
```

And if you want to be able to get those items on demand, for example so that you can compare them with the items being

produced from another such traversal, you end up structuring it into some objects:

```
class Inorder_Iterator:
    def __init__(self, bintree): self.stack = [Node(bintree)]
    def next(self):
        if self.stack: return self.stack.pop().next(self)
        raise StopIteration
    def push(self, other): self.stack.push(other)
    def __iter__(self): return self
class Node:
    def __init__(self, bintree): self.node = bintree
    def next(self, stack):
        if type(self.node) is type(()):
            left, middle, right = self.node
            stack.push(Node(right))
            stack.push(Visit(middle))
            stack.push(Node(left))
        return stack.next()
class Visit:
    def __init__(self, val): self.val = val
    def next(self, stack): return self.val
```

By contrast, Python generators let you write this:

```
def inorder_traverse(bintree):
    if type(bintree) is type(()):
        left, middle, right = bintree
        for item in inorder_traverse(left): yield item
        yield middle
        for item in inorder_traverse(right): yield item
```

That's 6 lines of code instead of the 19 of the explicit object version. Both are noticeably shorter, however, than the 25-line pseudo-Simula version with coroutines that Finkel presents (in chapter 2, subsection 2, p.33, figure 2.8).

Topics

- Programming (p. 3658) (286 notes)
- Syntax (p. 3738) (28 notes)
- Python (p. 3671) (27 notes)
- Lisp (p. 3552) (9 notes)
- Concurrency (p. 3386) (9 notes)
- Scheme (p. 3694) (8 notes)
- The CLU programming language

Opacity holograms

Kragen Javier Sitaker, 2016-08-16 (8 minutes)

We present an algorithm and example images demonstrating a new form of full-color "holography" printable on a standard laser printer that supports full parallax, rather than just one-dimensional parallax, like lenticular 3D and unlike traditional full-color holograms and Beaty's "scratch holograms". Like Beaty's design, these "opacity holograms" do not take advantage of the wave nature of light, but like interference holograms, measures of their quality degrade only proportional to the square root of the number of encoded frames, rather than linearly as with lenticular 3D.

Consider an image obscured by a grille, of the kind used for cryptography in the past. The grille is an opaque sheet with holes in it. When laid atop a ciphertext, a viewer sees portions of the ciphertext where the holes are, and the color of the grille elsewhere, which we will take to be black.

Naor and Shamir (1994) demonstrated how to use the grille approach to optically perform a one-time pad encryption of a bilevel image. The present author extended this algorithm to grayscale images, without loss of security, in a non-peer-reviewed publication in 2007.

The present work demonstrates an extension in a different direction: discarding security entirely as an objective, it uses N random grilles, or equivalently N spatial shifts of the same random grille, to optically decode approximations of N unrelated plaintext images from a single encoded image, with the addition of Gaussian noise with a standard deviation of $O(\sqrt{N})$ added, resulting in an $O(1)$ loss of intensity and contrast, and an $O(\sqrt{N})$ loss of effective resolution. XXX If the grille is physically mounted parallel to and a short distance away from the encoded image — for example, by printing them on opposite sides of transparency film or float glass — these spatial shifts can be achieved merely by viewing the assemblage from slightly different angles.

From here on, we will speak of N grilles, one for each input image, and we will speak of the image as grayscale, with possible values ranging from -1.0 black to +1.0 white; but everything generalizes to the cases of RGB images and N spatial shifts of a random grille. For the time being, we will take the grille's distribution to be 25% open holes, a number we will call L, and 75% opaque black.

The encoded image is a pixelwise sum of N different encoded subimages, each generated from a single plaintext image. A given pixel in the encoded image from among those visible through a hole in a particular grille consists of the corresponding pixel of the input image, plus Gaussian noise from the other subimages. If the noise has a uniform distribution with a mean at 0 across the entire image, then it will merely decrease the effective resolution of the image.

How can we get the sum of the other subimages to have a spatially uniform zero-centered Gaussian distribution? Consider each pixel in each subimage as a random variable, which with some probability shows through a hole in that subimage's grille and therefore must be equal to the original image's pixel, and otherwise can be chosen to have any value. By the Law of Large Numbers, the sum of the corresponding pixels in each subimage will tend to be Gaussian with a

variance that is the sum of the variances of the pixels in each subimage. So we are faced with the task of ensuring that this sum of N-1 subimages has mean 0 at each pixel and has a spatially uniform variance, even though with some significant probability the values of the pixels in particular subimages are fully determined by the corresponding input image pixel.

So, a given pixel in a given subimage might have some plaintext value, such as -.21, with probability L = 0.25, and be a free choice otherwise, while some other pixel in that same subimage might have a different plaintext value, such as .99, with probability 0.25, and be a free choice otherwise. The problem is making the free choice such that the mean of both distributions is zero and the variance is equal.

Getting the mean to be zero is easy: the naïve approach would be to always pick, for example, .07 for the first pixel and -.33 for the second one, when the choice is available. But an encoded subimage generated using this naïve algorithm will have widely varying variances, and worse, they will tend to vary in a spatially coherent way — where the plaintext image is close to medium gray, it will add almost no noise to the other subimages, while where it is close to black or white, it will add a great deal of noise. This will result in crosstalk visible in the decoded images.

The simplest way of fixing this defect of the naïve algorithm is simply to compute the variance for each pixel, and add an additional N+1th subimage purely of random noise with a compensating amount of variance for each pixel. (You could use a Gaussian distribution, or you could try to construct a distribution that will bring the distribution of the sum as close as possible to Gaussian.) Another possible way is to use more complicated distributions in each of the subimages, perhaps shaped to be as close to perfectly Gaussian as possible, so that the Law of Large Numbers kicks in immediately. I don't know how much the higher moments of the noise distribution will be visible, though. And I don't know how much this choice matters.

The final result image will, in general, have pixels outside the [-1.0, 1.0] range, and so will require some amount of clipping. We can scale and translate its range somewhat (diminishing contrast) to reduce the damage from clipping. Translating values is more harmful near -1.0, where it reduces contrast, than near +1.0, where it merely reduces brightness. Probably the most pragmatic approach is to do a linear scaling such that, say, the 1st and 95th percentile values are mapped to -1.0 and 1.0. XXX calculate how much damage this does and how it changes with the number of images.

However, a bilevel output device like a laser printer cannot reproduce continuous grayscale; it can only produce black pixels and white pixels. We would like the probability of a pixel being white to be proportional to its ideal whiteness, which we could do by XXX the usual dithering approaches, no? But it would be nice if we could avoid adding still more noise, by like just thresholding at 0 or something, but that would probably result in undesirably nonlinear performance.

The grille's load factor L (25% in the above) is independent of the number of images, but it affects the effective resolution and the noise levels. Higher load factors L will result in proportionally more noise (i.e. noise with a proportionally higher standard deviation), but also

proportionally more resolution. But the noise also affects the effective resolution: twice the noise requires four times the resolution to compensate. It would seem, then, that the lower the load factor, the better, which of course is absurd, because it implies that the optimum is L=0, a fully opaque grid. This is because the noise isn't actually proportional to L; it's proportional to L/(1-L), which, hmm, that doesn't fix the absurd conclusion, so maybe that's not it. XXX

Thinking further about L. To maximize effective resolution, we want to maximize $L/\sqrt{(L/(1-L))}$.

Topics

Fermat primes

Kragen Javier Sitaker, 2019-07-07 (4 minutes)

Fermat conjectured that all Fermat numbers, numbers of the form $2^n + 1$ where $n = 2^i$ for some $i \in \mathbb{N}$, were prime. In fact, as far as we know, only the first five ($i \in [0, 4]$) are prime, namely 3, 5, 17, 257, and 65537; Euler showed in 1732 that when $i = 5$ the number you get, 4'294'967'297, is composite. Fermat's original conjecture was in 1650, so it's sort of puzzling that it took 82 years to disprove it.

In fact, you can disprove it using Fermat's Little Theorem, that $a^n \equiv a \pmod{n}$ if n is prime. Nonprime numbers n that pass this primality test for every a are called "Carmichael numbers" or "Fermat pseudoprimes", and they do exist, but 4'294'967'297 happens not to be one of them. So, for example, when $n = 4'294'967'297$, $3^n \% n = 497'143'886$, which is not 3. (It does pass the test with $a = 2$.) This is usually a very much easier way to show that a large number is composite than by finding its factors.

You could reasonably argue that Fermat didn't have a gigaflops netbook in his lap when he made his conjecture, and so he could hardly be expected to go around raising numbers to such powers, but actually the calculation is something you could do with pen and paper in a day or two, particularly if you had a table of squares to speed you up. You proceed by successive squaring, dropping out a 4'294'967'297 when necessary; each number here is the square mod n of the previous one:

$3^{1} \% 4294967297 = 3$
$3^{2} \% 4294967297 = 9$
$3^{4} \% 4294967297 = 81$
$3^{8} \% 4294967297 = 6561$
$3^{16} \% 4294967297 = 43046721$
$3^{32} \% 4294967297 = 3793201458$
$3^{64} \% 4294967297 = 1461798105$
$3^{128} \% 4294967297 = 852385491$
$3^{256} \% 4294967297 = 547249794$
$3^{512} \% 4294967297 = 1194573931$
$3^{1024} \% 4294967297 = 2171923848$
$3^{2048} \% 4294967297 = 3995994998$
$3^{4096} \% 4294967297 = 2840704206$
$3^{8192} \% 4294967297 = 1980848889$
$3^{16384} \% 4294967297 = 2331116839$
$3^{32768} \% 4294967297 = 2121054614$
$3^{65536} \% 4294967297 = 2259349256$
$3^{131072} \% 4294967297 = 1861782498$
$3^{262144} \% 4294967297 = 1513400831$
$3^{524288} \% 4294967297 = 2897320357$
$3^{1048576} \% 4294967297 = 367100590$
$3^{2097152} \% 4294967297 = 2192730157$
$3^{4194304} \% 4294967297 = 2050943431$
$3^{8388608} \% 4294967297 = 2206192234$
$3^{16777216} \% 4294967297 = 2861695674$
$3^{33554432} \% 4294967297 = 2995335231$
$3^{67108864} \% 4294967297 = 3422723814$

$3^{13421772^8} \% 4294967297 = 3416557920$
$3^{26843545^6} \% 4294967297 = 3938027619$
$3^{53687091^{12}} \% 4294967297 = 2357699199$
$3^{10737418^{24}} \% 4294967297 = 1676826986$
$3^{21474836^{48}} \% 4294967297 = 10324303$
$3^{42949672^{96}} \% 4294967297 = 3029026160$

This final result at the end would have needed to be 1; if you multiply it by 3 mod 4'294'967'297, you get the number I gave earlier, 497'143'886, which is manifestly not 3.

Squaring a ten-digit number like 3'422'723'814 by hand is a significant amount of work, though you can reduce it substantially by looking up the first and last digits in a table of squares. But of course poor old Fermat couldn't just run off the 60-page table of the first ten thousand squares with a one-line command:

```
perl -le 'print "$_^2 = ", $_*$_ for (1..10000)' | pr -3 | lpr
```

He might have had such a table available — people were printing books with mathematical tables of this size around this time — but I'm uncertain as to whether he did. If he did, it probably contained many errors. Wikipedia suggests no such table was available until Antoine Voisin published a table of 1000 squares in 1817, although multiplication algorithms using them have been known since ancient Babylonia.

And he didn't even have Karatsuba multiplication (see Karatsuba (p. 2090)) to help him along.

Topics

- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Algebra (p. 3309) (11 notes)

Interactive calculator 0

Kragen Javier Sitaker, 2015-09-17 (2 minutes)

I want a calculator that lets me do instant calculation with low precision and then do the same calculation to higher precision, and that is instantly responsive.

I want to be able to mix values that are functions of time with simple scalars.

One aspect of this is using interval arithmetic or some similar kind of approximation to ensure that the rendering is responsive and doesn't block user input.

Another aspect is that I want to see a time-series dangling from each non-scalar-valued operator and variable in my expression, although maybe not all of equal prominence; maybe mouseovers to focus on the active ones. And in some cases, I'd like to see it updating in real time as I'm editing the expression, like if it's a sound signal I'm generating.

I want the time axis to be vertical, rather than the usual horizontal, because the expression is traditionally represented horizontally, and trying to run the time axis horizontally too would mean that you can't see much of any variable.

Another is that the RPN aspect of rpn-edit works really well: each keystroke has an immediate effect, and there are no unbalanced-parentheses errors. So I want to stick with that. If I'm playing music, I probably want the leftmost top-level value to be the one that's played by default, rather than the ones I'm building up to the right.

Another thing, though, is that I want to be able to move back and forth between continuous and discrete time-series without changing the formula, and I want to be able to apply the same function to different input time-series. This requires some kind of functional abstraction, at least giving names to things.

For trading analysis, I need more powerful time-series manipulation than what I've implemented so far. I need to be able to control, and in particular subset, time. I need to be able to conform time-series acquired with different (and irregular) time bases. I need operations on selections like "extend ten minutes into the future". I need to be able to apply smoothing filters.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Interval and affine arithmetic (p. 3528) (24 notes)
- Latency (p. 3542) (19 notes)
- Calculators (p. 3362) (11 notes)
- Time series (p. 3750) (6 notes)

Caustic simulation

Kragen Javier Sitaker, 2018-09-10 (updated 2018-11-04) (2 minutes)

I'd like to do simulation and optimization of optical caustics. The basic phenomenon is that caustics can have a very high dynamic range and project a reasonable distance, and in particular can produce divergence-limited-brightness lines in the projected image.

Fundamentally the simplest idea is that if you take a single pencil of a collimated beam impinging on a nearly-flat mirrorlike reflector surface and look at the pattern it makes on a target — whether an interface between optical media or a metallic surface — the point where the pencil lands on the target is a continuous function of its position on the reflector, and can be computed using Heron's law of reflection. However, the pencil may be deformed to be convergent, divergent, or both, by the curvature of the reflector at that point. If it is converged to a point or a line, then there is a singularity of infinite brightness in the caustic.

The derivative of the point to which the pencil is directed with respect to the point where it's reflected may go through zero at these points, in X, Y, or both. The product of the two derivatives gives, in some sense, the size of the spot.

Probably the most sensible thing to do is actually to compute a bunch of reflected rays from different points, then calculate some kind of spline joining them. The area of each square of the spline grid provides a local brightness, but the grid may fold back on itself through the aforementioned lines of singularity. This is close to the approach taken by Evan Wallace in WebGL-Water.

(Of course the bright lines are limited in brightness both by the divergence of the incident light and diffraction over the focusing zone.)

See also Gauzy shit (p. 2985).

Topics

Data archival on gold leaf or Mylar with DVD-writer lasers or sparks

Kragen Javier Sitaker, 2018-04-27 (5 minutes)

I was doing some calculations yesterday about blowing holes in things with lasers from DVD writers. My estimate is that you can blow a micron-diameter hole in just about anything with those 400-milliwatt red lasers if you get them properly focused, in about a microsecond.

That would store a few gigabits in a fairly permanent way into a 3.5" square of material, in a few gigamicroseconds, so a few kiloseconds. This is the first long-term archival technology that I think is feasible to do on a shoestring; a DVD-writer is a lot cheaper than a focused-ion-beam etching machine. One of the most appealing media to use for this purpose would actually be old floppy discs, because the Mylar medium is extremely chemically stable, and the oxide coating is nice and optically absorbent, but glass or aluminum seems like it would work fine.

An even simpler alternative that would work with metal foils or even metal surfaces: blow holes in them using sparks from a graphite point (of a radius on the order of a micron, about as sharp as a new scalpel, a bit sharper than the sharpness of a razor blade) driven by a capacitor. However, most metals are not very stable in Earth's atmosphere.

Gold leaf as a medium

Gold is an exception; it's stable in Earth's atmosphere. I can buy 100 sheets of 140mm square gold leaf for AR$350 (US$20), which works out to about 2 million square millimeters, or 100 000 square millimeters per US$.

The data density we're talking about here is about a megabit per square millimeter, so 2 million square millimeters is about 250 gigabytes. Hard disks are a bit cheaper at this point, by a factor of maybe 4, but they weigh more and won't last as long.

Gold and spark properties

Gold boils at 2970°, occupies 19.30 g/cc, and consumes 12.55 kJ/mol of latent heat of melting and 342 kJ/mol latent heat of vaporization; its molar heat capacity is some 25.42 J/mol/K. At the 650-nm wavelength of red lasers, it's about 98% reflective, which makes red lasers kind of a shitty way to boil it, but an arc in air deposits about 90% of its energy at the negative electrode (the one that's releasing electrons and being bombarded with positive ions). Gold leaf can be as thin as about 100 nanometers thick or a bit more, but 200 to 400 nanometers and thicker is also sold. Its molar mass is 197.0 g/mol.

Gold leaf calculations

The per-mole quantities above work out to 63.7 kJ/kg latent heat

of melting, 1.74 MJ/kg latent heat of vaporization, and 129 J/kg/K of specific heat. So boiling gold from something like room temperature should take, roughly, 383 kJ/kg of heating plus the latent heats, or 2.19 MJ/kg, or 2.19 J/mg. A cubic micron of gold weighs 19.3 picograms, so boiling it requires about 42.3 nJ. But our gold leaf is a fraction of a micron thick, perhaps 200 nanometers, so you only need a fifth of that; if you get the polarity right and the pulse quick enough, you need maybe 10 nJ per spark to blow holes in a sheet of gold foil, or about 500 nJ per laser pulse, which would be 1¼ microseconds of 400 mW — I hope that's fast enough to keep the heat from conducting away.

The 250 gigabytes on 100 pages of 140-mm gold leaf would occupy, hypothetically, some 20 microns of thickness, if you laid them atop one another, for a volumetric density of about 10 exabits per cubic meter or 10 gigabits per cubic millimeter. But you can't do that because they would cold-weld together and then you wouldn't be able to separate them in order to read a particular page.

Gold is kind of a worst-case material in some ways — you lose a factor of 25 or 50 compared to any substance that is reasonably emissive in the relevant wavelength range, and it has very high thermal conductivity.

Nearly any other material will be a factor of 2 or more less dense, boil at a factor of 2 lower temperature, and have a factor of 10 or higher emissivity, have similar or lower heat of vaporization, and have worse thermal conductivity, so will be easier to blow holes in. Even aluminum has a third lower thermal conductivity of gold, though it has higher heat of vaporization.

Topics

- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Thermodynamics (p. 3747) (49 notes)
- Archival (p. 3322) (34 notes)
- Microprint (p. 3582) (8 notes)
- Sparks (p. 3724) (4 notes)
- Lasers (p. 3541) (3 notes)

Bench trash power supply

Kragen Javier Sitaker, 2018-04-27 (9 minutes)

So I was thinking that an AVR-driven lab power supply from trash (except for the microcontroller) would be a good idea. You need a lab power supply for a lot of electronics tinkering.

Lots of discarded electronics have power supplies in them. They usually have a few problems we would need to fix for them to be decent lab power supplies:

• They're noisy switching power supplies, not quiet linear ones.
• Their output voltage is fixed, not adjustable.
• They don't have output current limiting, so it's easy to blow them up by shorting them out.
• They have no readout to tell you what the actual current (and voltage) is.
• A lot of them are only 5 volts.

Problem #1 is really tough to solve, so I think I'll ignore it. I think I can solve the others pretty reasonably, though, with a generic microcontroller and a relatively simple circuit plugged into the output of some generic salvaged power supply, like an ATX power supply, say.

The basic idea is that you drive a buck-boost converter (a diode, an inductor, a capacitor, and a MOSFET) using a microcontroller's digital output. To its output you hook, in series, an inductor, a sense resistor, and the load, with a capacitor and megohm resistor across the load; then you add some diode protection. You hook each end of the sense resistor to a separate high-impedance voltage divider to scale down the voltage to a range adequate to the microcontroller's analog inputs. Calibration trimpots form part of these voltage dividers.

Two additional potentiometers are attached to other analog inputs of the microcontroller for use as user input, and an old Nokia serial LCD is attached to five digital I/O pins of the microcontroller.

Actually, I think maybe the inductor and capacitor on the load end aren't necessary. The AVR can respond to changes in load in about 50 µs. I was thinking that the inductor would keep the short-circuit current from rising too quickly for the Arduino to respond, but in fact the buck-boost converter can't respond instantly either — if its output impedance suddenly approaches zero, it won't dump an unlimited amount of energy into it because it doesn't contain an unlimited amount of energy in the first place! It isn't until the next cycle that it has the opportunity to charge up, but of course the microcontroller then has the opportunity to *not* charge it up.

ATX power supplies have ±12V lines, but they are very limited in power. The 5V lines are where the real power is at, typically hundreds of watts of it. A pretty minimal bench power supply should be able to produce up to 30 volts and at least an amp or so, say 1.5 amps, which works out to 45 watts. (You can run four to eight of these off a single ATX power supply.)

Consider first the simple inverting topology, even though that isn't the right thing for a bench power supply.

If we're running the buck-boost converter at 31.25 kHz (the highest

frequency an Arduino running an AVR at 16MHz supports with its native PWM with full 8-bit resolution) then each cycle is 32 microseconds and involves converting up to 1.44 millijoules from the input voltage of 5 V up to the output voltage of 30 V.

At this case, the duty cycle is 17%: the inductor (on the 5V side of the freewheel diode) is getting its current ramped up with 5V for 83% of the time, then getting its current ramped down with 30V for the other 17% of the time when the transistor is turned off. During that time it dumps 83% of its 1.44 millijoules into the capacitor and 17% of it into the load, I guess, which means that the capacitor actually needs to be able to hold several times those 1.44 millijoules in order to keep ripple reasonable.

In particular, say our output ripple is 3%, one volt. That means that the difference between the capacitor energy at, say, 29V, and 30V, is that 1.44 millijoules. $C(30^2-29^2)V^2/2 = 1.44\text{mJ}$, so $C = 2\cdot1440\mu\text{F}/(30^2-29^2) = 49\ \mu\text{F}$. This means that the total energy in the capacitor is $CE^2/2 = 22$ mJ. If you were to short it out with a ¼Ω short circuit (about the internal resistance of a D cell), you would briefly get 120 A and 3.6 kW — but with a time constant of 12 microseconds.

Suppose we use a 10Ω sense resistor to measure the current, though. Now at 1.5 A, we drop 15 V across the resistor, and at 30 V, it's impossible to draw more than 3 A, no matter how we short the output. This, however, requires that the capacitor charge up to 45 V, in order to get 30 V to such a hefty load. So we could draw as much as 4.5 A actually, and a rather alarming 45 W on the sense resistor very briefly, coming down to 15 W after the control circuit kicks in. Gonna need a hefty heatsink on that sucker.

So now the duty cycle is 11%: 5 V to spin up the inductor for 89% of the time, then 45 V as it charges up the capacitor the other 11% of the time. And instead of 1.44 millijoules, because of the extra power in the sense resistor pushing us up to 60 W, it's 1.92 millijoules, so our capacitor goes up to 65 μF, and the total stored energy is 29 mJ. And if you short out the output, the time constant is 290 milliseconds.

This is sort of alarming because it means you can't get a reasonable slew rate on the output voltage, whether in response to changing load or in response to user input. I think the solution is a combination of a higher switching rate and a somewhat smaller sense resistor, basically in order to keep the energy stored in the capacitor down.

(No, wait, all of the above is wrong. The time constant was 650 microseconds.)

It's useful to have some kind of scale for how much energy we're talking about here. 2.3 kJ is enough to boil a gram of water, a cubic centimeter. 2.3 J is enough to boil a milligram of water, a cubic millimeter. 2.3 mJ is eough to boil a microgram of water, a cube 100 microns in diameter. This would still be enough to vaporize the bond wire of a chip if it could somehow be released there, but the bond wire will never be more than 0.1Ω, so even with a sense resistor of 1Ω we can ensure that the majority of the short-circuit energy is dissipated in the sense resistor and not the bond wire. So tens of millijoules should be fine, but not more.

Let's suppose we can manage a higher PWM frequency — say, 125 kHz, maybe by using 6-bit resolution, since we're only shooting for 3% ripple anyway. And a 4.7Ω sense resistor, so at 1.5 amps we only

dissipate 10.6 watts in its 7 V, for a total of 55.5 W, which means 0.4 millijoules per cycle, and 15 microfarads is enough to keep the ripple acceptable, and our time constant when shorted is 70 µs. Our duty cycle then is 5V/37V = 13.5%.

You might also want some kind of minimum load there, in parallel with the real load, to keep the time constant reasonable when you're trying to adjust the voltage under open-circuit conditions. 50 ohms or something.

Okay, enough about the capacitor and sense resistor. What about the inductor? It needs to spin up to some current Imax at 5 volts in 6.9 microseconds, then spin back down at 37 volts in 1.1 microseconds, delivering 0.4 millijoules. 0.4 millijoules in 1.1 microseconds at 37 volts is an average of 9.8 amps, so the max should be 19.6 amps. Let's say 20 amps. That means the inductor needs to be 1.7 microhenries.

Such inductors exist; Digi-Key lists a 12.8 × 12.8 × 7 mm item that costs US$1.20.

What happens if the inductor is some other size? Like, will it work better if the inductance is bigger or if it's smaller?

Topics

Copper plating furniture

Kragen Javier Sitaker, 2017-07-19 (updated 2017-09-01) (4 minutes)

Copper plating is an effective technique for preventing microbial contamination of surfaces; it can also prevent corrosion and contact dermatitis from other metals. US pennies are zinc electroplated with 20 microns of copper; this is sufficient to prevent them from wearing visibly even over the course of decades. The Royal Mint claims that their 25-micron coating is sufficient to give circulating currency a life of 25 to 30 years.

Materials cost

How much would it cost to buy enough copper to copper-plate furniture, plates, doorknobs, silverware, walls, etc.? Let's suppose that 10 microns is adequate, since we can replate when it wears through after a few years, so each m^2 of surface is 10 $\mu(m^3)$ = 10 $m\ell$. Copper's density is 8.96g/$m\ell$, so that's 0.0896kg/m^2. At the moment, the price of NYMEX HG contracts — in some sense the wholesale price of copper — is US$2.26 to US$2.30 per pound. That works out to about US$5 per kg, which works out to a copper cost of about US$0.45/$m^2$ for this 10-micron plating.

So copper-plating your entire house is going to be dominated by the cost of the plating process, not of the copper itself. The copper itself is a quite affordable cost.

Dumpster-diving

In fact, you might be able to dumpster-dive enough copper to copper-plate every surface in your house. An AWG20 copper wire, suitable for carrying 5 amps, is 0.812 mm in diameter, according to Induction kiln (p. 2352); a meter-long power cord cut off a small discarded appliance might contain two of them, contains just over one cubic centimeter of copper, which is to say 8.96 g — enough to copper-plate a thousand square centimeters, which is quite a bit of silverware and door handles.

Around here, larger chunks of copper and copper alloys are rapidly dumpster-dived by *cartoneros*, who even break the yokes off discarded CRTs to recover the copper, typically within an hour or two. So finding free copper by the kilogram is not common; nevertheless, I've picked up a discarded microwave oven or two.

Electroplating chemistry

As reported in Immersion plating of copper on iron with blue vitriol (p. 1099), it's feasible to electroplate copper using vinegar and salt, though professional metal finishers are not enthusiastic about "hardware-store chemistry". A nickel flash (also platable with vinegar and salt) is recommended to get an adherent copper coating on steel, while for chromium-based stainless steel, people on #electronics have told me that copper will adhere directly, while *nickel*-plating it requires a *copper* flash. I haven't tried it yet, but I bought some Argentine 1941 50-centavo pieces made of pure nickel.

Energy cost

The 2001 edition of the ASM Specialty Handbook *Copper and Copper Alloys* says, on p.133, Figure 4, in the "Copper and Copper Alloy Coatings" chapter, that it is practical to achieve copper deposit thickness of 3 to 3.5 μm per minute with 8–10 amperes per square decimeter with periodic current reversal, at I think 6 volts; as its source it cites "*Electroplating Engineering Handbook*, Reinhold, 1971, p. 748, 750". This would provide the desired 10μm thickness in about three minutes at a cost of, say, 9 A/dm² · 100 dm²/m² · 3 minutes · 6 V = 972 kJ/m² ≈ 1MJ/m². That's about US$0.027/m² for the energy at a retail cost of US$0.10/kWh, which is measurable, but more than an order of magnitude lower than the cost of the copper itself.

Topics

- Materials (p. 3560) (112 notes)
- Pricing (p. 3646) (89 notes)
- Household management and home economics (p. 3504) (44 notes)
- Chemistry (p. 3373) (20 notes)
- Plating (p. 3637) (4 notes)
- Metallurgy (p. 3576) (4 notes)
- Copper plating (p. 3394) (4 notes)
- Copper (p. 3395) (4 notes)

Waterfryer

Kragen Javier Sitaker, 2019-04-20 (1 minute)

An "airfryer" is a sort of convection oven that rapidly achieves frying-like results by blowing hot air over the food at very high speed, browning the surface. It occurred to me that you could, similarly, use high-speed pumped water circulation to rapidly heat food to boiling, or with some pressure, a bit higher than boiling. You can't brown the surface but you can cook the food.

This may be less of a difference from regular cooking (than an airfryer) because the bubbles rising in a boiling pot already produce substantial water circulation, far more than the air convection in a conventional oven; also, the water's thermal conductivity and specific heat are much higher and close to the food's, so perhaps in a boiling pot heat is already being transferred to the food so rapidly that most of the thermal resistance is in the food, not the heat transfer medium.

Topics

Wang tile font

Kragen Javier Sitaker, 2018-08-16 (5 minutes)

Old computer display systems used a "character generator" to drive the raster display (see Gradient pixels (p. 2202) for more detail), giving rise to the traditional "terminal screen" look and feel, with its single shitty typewriter font, blaring primary colors, block cursors, and readability-slaughtering background-color transitions on character-cell boundaries. The major advantage of this system is that it saves a lot of memory; in Gradient pixels (p. 2202), I gave the example of a color 80×25 terminal with 8×16 pixels per character, which requires somewhere between 2000 bytes and 8096 bytes for a 1-bit-deep font, and perhaps 20 kilobytes or so for a font with antialiasing or color. But a framebuffer for the 256000 pixels would require probably 128–768 kilobytes.

An interesting alternative that occurred to me today is to drive the pixel generator from a complete set of Wang tiles. The pixel generator still uses a rectangular array of data directly corresponding to physical screen positions, but the "font" is potentially larger, because the pixels on the screen are a function not only of the glyph index at the underlying location, but also of some pixel generator state. Specifically, you have, say, two bits of "bottom-edge-color state" from the corresponding cell in the previous line (that is, the cell above this one), and, say, two bits of "left-edge-color state" from the corresponding cell in the previous column (that is, the cell to the left). And the font entry indexed by these four bits and the bits of the glyph index contains not only pixels but also the new bottom-edge-color state and left-edge-color state.

In the simplest case, where the rectangular array has no bits per character position, the screen contents are entirely determined by the font; at each position, the four bits of edge state uniquely determine the pixels to display at that position, from the 16 available glyphs in the font, and also determine the new edge state. With a single bit of color per edge, and thus only 4 glyphs, this is sufficient in itself to generate some simple fractals and count in binary, but nothing more interesting, but with two bits of edge state, all kinds of interesting programs are possible. (There are $16^{16} = 2^{64}$ possible programs that can be encoded in the font edge state bits.)

When the rectangular array actually has some data in it, we can think of the bits in it as specifying which of the valid alternative tiles, as constrained by the environment, is to occupy the space; this, in turn, affects the environment of the cells below and to the right.

To take a slightly more interesting case, you might use 5 bits per character position to encode which letter is to appear there, with four variants of the space character — one to set the right-edge state to any of four alternatives, which we could gloss as FIGURES, TITLECASE, UPPERCASE, and LOWERCASE. The TITLECASE glyphs might be identical to the UPPERCASE glyphs in appearance, but the right edge of a TITLECASE glyph leaves the left-edge state set to LOWERCASE, while the right edge of an UPPERCASE glyph remains in UPPERCASE. This is a somewhat suboptimal way to handle case-shifting, since it means that each

case-shift leaves a blank space on the screen — undesirable if you're talking about Mohamed ElBaradei, or example — but it shows that there are some interesting possibilities.

Accommodating descenders from previous lines, or using special word-initial forms of letters, are other possible uses at the character-cell level.

But what if instead we use Wang tiles at a slightly finer granularity than per character cell? After all, if we try to have 256 possible tiles for any given environment, we need 4096 tiles if we have two bits of color on each edge, which implies an uncomfortably large "font ROM". What if each tile only covers *part* of a letter? Maybe "ɑ", "d", and "q" could share a common code for the common part of the letter, while "d" and perhaps "l" could share a common ascender glyph. Perhaps we could even have a smaller font ROM rather than a larger one.

For circuit schematics, we could use edge colors to note whether a wire is or is not present at each edge of a tile, eliminating some duplication. However, without backtracking, the potential promise for this to propagate changes across your whole schematic without further work is limited — they can only propagate down and to the right.

Topics

Deep freeze

Kragen Javier Sitaker, 2017-08-22 (updated 2019-01-22) (7 minutes)

A person needs about 2000 kcal of food energy per day, which, according to conventional nutritional thinking, should ideally be broken down as about 40% carbohydrate, 30% protein, and 30% fat: 800 kcal of carbohydrate (weighing 160 g), 600 kcal of protein (weighing 120 g), and 600 kcal of fat (weighing 67 g), for a total weight of 350 g of macronutrients.

But foods do not consist entirely of macronutrients, even in storage. To take an example food, soybeans are normally required to have ≤11% moisture content for food use; 172 g of cooked soybeans contains (according to one source which cites USDA SR-21) 17 g carbohydrate, 15 g fat, 29 g protein, and 1.6 g of micronutrients, totaling 63 g, but 10 g of the carbohydrate is indigestible fiber, about 16% of the total macronutrient mass. So only about 75% of dry soybeans is made of digestible macronutrients, a number which I will take as typical for other dried foods.

This means you need about 470 g of stored dried foods per day to get to 2000 kcal. This is an amount that is reasonably tractable to store; 64 years' worth is just 11 tonnes. (If we take soybeans as a typical bulk food, they cost about US$10 per bushel, which is 60 pounds; 11 tonnes is 404 bushels or US$4040, US$370 per tonne.) But without further work, stored foods will deteriorate in nutritional value during 64 years, and may even rancidify, producing free radicals and other poisons.

The only way to prevent this nutritional deterioration is to keep the food cold in a freezer, which generally requires three components: insulation, refrigeration, and passive thermal storage (either thermal mass or phase-change materials).

In habitable regions of Earth, the ambient temperature is too high for long-term food storage, so insulation is needed to prevent heat exchange between the food and the environment. You can measure the thermal resistance around a thing in K/W, kelvins per watt — generally the heat flow is proportional to the temperature difference. This thermal resistance is proportional to the thickness of the insulation, inversely proportional to the surface area through which the heat is flowing, and proportional to the insulance of the material. Insulances range from 1000 W/m/K for diamond down to 20 mW/m/K for silica aerogel. Water's insulance is 580 mW/m/K, though it rises to 2200 mW/m/K when frozen, and commonly-used low-temperature insulating materials range from straw at 90 mW/m/K to styrofoam at 33 mW/m/K. (Refractory insulators are trickier; firebrick is normally around 500 mW/m/K). Minerals are less insulating; quartz is 3 W/m/K, sandstone is 1.7, graphite is a metal-like outlier at 168, copper is 400. Water-saturated dirt is about 2–4 W/m/K, while dry sand is about 150–250 mW/m/K, an order of magnitude better.

Because the heat leakage is proportional to the surface area, larger freezers are more efficient — they have less surface area per unit volume, so they lose less heat per unit of food stored, if we hold constant the insulation thickness and material. A cubic-meter sphere

has about 4.8 m² of surface area; a thousand-cubic-meter sphere has 480 m² of surface area, only 0.48 m² per m³. So it will warm up ten times more slowly, and it will need only a hundred times as much refrigeration — one-tenth as much per unit of food stored.

Refrigeration is needed both to initially cool the food down and to compensate for heat lost through the insulation. In the steady state, the refrigeration needed is proportional to the temperature difference (to ambient) and inversely proportional to the thermal resistance (to ambient).

Passive thermal storage is needed to keep the food cold when the refrigeration is turned off, either because the thermostat has turned it off or because energy is not available to power the refrigeration. (A solar-powered refrigeration system might only work on sunny days, for example, and a grid-tied system won't work during blackouts.) As with refrigeration, the passive thermal storage needed is inversely proportional to the thermal resistance of the insulation. The food itself has a certain thermal mass, typically about 2kJ/kg/K, and it can be supplemented with a phase-change material such as water ice, which has an enthalpy of fusion of some 79 kJ/kg. XXX XXX I think this is wrong and the correct figure is 333 kJ/kg. Adding phase-change materials increases the volume that must be insulated, which increases the surface area, requiring a greater depth of insulation over a larger area. Consequently, there is a tradeoff between resiliency to power outages and energy efficiency.

Water-saturated dirt is not a very good insulator, with insulances of around 3 W/m/K, about 30 times worse than straw and 90 times worse than styrofoam. But you can get 30 meters of water-saturated dirt insulation just by drilling a 30-meter well, while the equivalent meter of straw insulation requires you to buy and then move around cubic meters of straw. Better still, dry sand is only a factor of 3 worse than straw, so it's easy to get an enormous thermal resistance in dry areas. Straw is by far the cheapest decent insulation material; it costs at a minimum about US$25 per tonne just to replace its fertilization value and bale it; in places the cost can be two or three times that, depending on competing demand for livestock feed.

If you were to try to build a mound of dry sand aboveground in order to get dry-sand thermal insulances in an otherwise swampy area, you'd need the sand to cost less than about US$10 per tonne to make it cheaper than straw. Construction sand for concrete costs about US$20–40 per tonne delivered here in Buenos Aires, but if you can instead just move earth around locally (producing a pit and a mound), you could conceivably improve on that. However, excavation costs are typically close to that same US$20–40 per tonne using construction-scale hydraulic earthmoving equipment.

So probably straw insulation is the

Topics

What can you build out of 256-byte ROMs?

Kragen Javier Sitaker, 2018-12-02 (1 minute)

Suppose we have, as a fundamental unit, a simple finite state machine: an asynchronous ROM with some address inputs fed from its own registered output and other address inputs coming from elsewhere. To be concrete, consider the case where the ROM contains 256 8-bit words, 4 address lines attached to its own registered output, and 4 address lines coming in from elsewhere. The ROM contains 2048 bits; if it is in fact programmable (for example by way of a serial bitstream) it amounts to a programmable lookup table; it can plainly run quite fast. What can we do with such a machine?

It's straightforward to use it to hold 4 bits, or one decimal digit, of an up/down counter; you can also use it as a 3-bit accumulator slice, adding its input to its contents on each clock cycle, with carry in and carry out.

However, upon further thought, I think this is probably not a useful design; it necessarily contains upward of 4000 transistors, enough for an entire 8-bit CPU. The 256×8 configuration memory is surely better off as 128 16×1 memories, as in a normal FPGA.

Topics

Augmenting a slow precise ADC with a sloppy fast high-pass filtered parallel ADC

Kragen Javier Sitaker, 2017-03-20 (2 minutes)

A 16MHz AVR like the one in an Arduino can read an 8-bit port fairly frequently, in theory every 4 cycles or so (4MHz), though I think in a real program you'd be lucky to get every 15 cycles (1MHz). It occurred to me that you could maybe do a quasi-equivalent of an R-2R DAC to get some kind of sample at that frequency: by running different voltage dividers from a voltage source across the different pins, the digital level thresholding can maybe give you some approximate idea of how high a signal is: a 3-bit idea of it, to be exact.

This sounds kind of useless, but I was thinking you could use it in combination with the ADC, which on the ATMega328 runs at up to 76.9 ksps or up to 15 ksps at maximum resolution. Each conversion takes 13 ADC clock cycles, where the ADC clock runs at between 1/128 and 1/2 the system clock, thus 125kHz to 8MHz at a 16MHz system clock speed; this gives us 9.6 ksps to 615 ksps, although I'm guessing that running the ADC clock at over 1MHz won't produce useful results.

I was thinking that you could run the normal ADC at its 15 ksps speed, thus providing an absolute 10-bit measurement of the input voltage, but then sample a high-pass-filtered version of the input voltage at about 1MHz and only three bits per sample. This would give you about 64 3-bit relative samples for each 10-bit absolute sample, gathering information about frequency components up to 500kHz. If there is significant overlap in the frequency responses of the two parts of the system, you can use the slow, absolute, high-precision ADC to calibrate the fast, relative, low-precision ADC.

You probably want to buffer the input voltage on its way into the high-pass filter and voltage dividers in some way — at least an emitter follower or source follower, maybe a whole op-amp buffer or something.

This seems like it might be capable of converting an ATMega328 into a 500kHz-bandwidth "oscilloscope", which is like 2½% of a real oscilloscope.

Topics

Hash gossip exchange

Kragen Javier Sitaker, 2015-11-19 (4 minutes)

I saw an article on HN the other day with a title like "the opposite of a Bloom filter", making the observation that if you make a hash table where you resolve collisions by simply discarding one of the colliding keys, then you get a probabilistic-set data structure that may have false negatives but never false positives, which is in some sense the reverse of a Bloom filter. It's very similar in a sense to a CPU memory cache (which also tracks the contents of memory as well as the data present there).

This comes at a fairly heavy space cost compared to typical Bloom filters, since you have to actually store the keys in order to ensure that you aren't getting false positives.

It occurred to me that you could use this approach to implement an efficient stateless gossip protocol for large datasets, but then it occurred to me that actually you can do better with Bloom filters.

Bloom filter approach

When you make contact with a new peer, generate a high-false-positive-probability Bloom filter of your current set of keys, using HMAC as the hash algorithm, keyed with a random nonce. Send the filter and the nonce to the peer. The false-positive probability can reasonably be as high as 90%, which should allow you to use a single hash function and much less than one bit per key.

When you receive a filter and a nonce, hash all your keys with the nonce and probe to see which are in the filter. If any are not in the filter, send them in your reply.

This sends a randomly selected 10% of the missing data to the peer. This should be sufficient to provide efficient information distribution if encounters are frequent but happen at intervals relatively large compared to the communication latency. In particular, it should be possible to keep the filters several orders of magnitude smaller than the dataset being synchronized, and each node will almost always receive each data item only once, even if it's simultaneously syncing with multiple peers.

With a fixed false-positive probability, the filter will grow linearly with the dataset. This problem can be avoided by increasing the false-positive probability over time, but of course that makes the network gradually slower as information accretes. Suppose that, instead, we partition the dataset into epochs of, say, one day each, in such a way that each node can determine independently which epoch a particular key belongs to, and we compute and transmit a separate filter for each epoch. Then we can choose a different convergence rate for each epoch: perhaps I want today's epoch to have a 90% false positive rate, but yesterday's 95%, 97% for the day before, and so on. In this way it is possible to continue very slow convergence even for very old epochs.

(Rathe than choosing different filter sizes, you could merge different numbers of epochs into each filter, producing different load factors; this is probably actually a better idea. So you might have one filter for the current minute, one filter for the previous three minutes,

one filter for the previous nine minutes, and so on.)

In addition to being more efficient, this approach is somewhat more privacy-protecting than the standard approach used by e.g. BitTorrent, in which you send a full list of all the keys (pieces, in the BitTorrent case) you have. That list of keys is almost certainly enough to identify you uniquely for a period of time. By contrast, it should be somewhat more difficult to identify the sender of such a filter: infeasible if you have no idea what keys might have been hashed into it, but even if you have some candidate keys, each one only gives you a small, probabilistic amount of information.

Topics

Analogies between spring–mass–dashpot systems, electrical systems, and fluidic systems

Kragen Javier Sitaker, 2016-10-30 (4 minutes)

The direct stiffness method of the finite element method turns a system of springs into a system of linear equations, imposes boundary conditions by removing some of the variables to make the system nonsingular, and then solves the system.

If we restrict our attention to systems of Hookean springs in one dimension connected at nodes, we have essentially two relationships:

• The net force from the springs on any node, except for those held fixed by boundary conditions, is zero.
• The force through a spring is its rigidity times the difference in displacement between its endpoints.

This immediately suggests comparison to electric circuits. For example:

• The net current from the resistors into any node is zero, except for those connected to current sources.
• The current through a resistor is its conductance times the difference in voltage between its endpoints.

And the resulting series, parallel, and bridge laws are the same for springs and for resistors.

This suggests extending the analogy {force = current, rigidity = conductance, displacement = voltage, springs = resistors} to other components. In this analogy, a dashpot, whose force is its "dashpoticity" times the rate of change in the displacement between its endpoints, or equivalently the difference in velocity between its endpoints, is analogous to a capacitor, whose current is its capacitance times the rate of change in voltage between its endpoints. This extends the analogy to the following:

Mechanical	Electrical		
force	current	F = kd	I = GV
stiffness/rigidity	conductance	k = F/d	G = I/V
spring	resistor		
displacement	voltage	d = F/k	V = I/G
dashpot	capacitor	F = x dd/dt	I = C dV/dt
dashpot-stiffness	capacitance		C = Q/V = ∫I dt/V

For reasons of conservation of energy, an inductor has no passive mechanical equivalent in this analogy; it would need to have a displacement proportional to the rate of change of force.

But that's not the only possible analogy, since other linear circuit elements (inductors and capacitors) combine in series and parallel in

the same way. For example, making capacitors instead of resistors analogous to springs:

• The net charge at any node is zero, except for those connected to current sources.
• The charge across a capacitor is its capacitance times the difference in voltage between its endpoints.

That is:

Mechanical	Electrical		
spring	capacitor	???	???
compliance? stiffness?	capacitance? elastance?		
velocity			
force			

In this analogy (force = charge, displacement = voltage, rigidity = capacitance) if we differentiate with respect to time, we find that velocity is rate of change of voltage and FUCK I HAVE NO FUCKING CLUE. It seems like inductors should be masses, what? Since they're 180° away from springs? On a mass, the derivative of velocity is force divided by mass. On an inductor, the derivative of current is voltage divided by inductance. So does that mean {velocity = current, force = voltage, inductance = mass}?

Presumably you can make the analogy equally valid in exactly the opposite direction, with springs representing inductors, masses representing capacitors, and dashpots representing resistors. In this analogy, I think charge represents displacement, velocity represents current, and force represents voltage: the rate of change of displacement of a mass is its velocity, and the rate of change of velocity of a mass is the force divided by the mass, thus {force = voltage, velocity = current, capacitance = mass}. This is exactly contradictory to what I came up with in the previous paragraph, so clearly I am smoking crack.

This analogy has:

force	voltage				
velocity	current	v = dd/dt	I = dQ/dt		
displacement	charge				
spring	inductor				
dashpot	resistor				
mass	capacitor				
stiffness	inductance?	k = F/(dv/dt)	L = V/(dI/dt)	!?!?!?	

Fuck, I have no idea.

Topics

• Physics (p. 3632) (119 notes)
• Math (p. 3564) (78 notes)
• Facepalm (p. 3450) (24 notes)

Constructing error-correcting codes using Hadamard transforms

Kragen Javier Sitaker, 2013-05-17 (updated 2013-05-20) (22 minutes)

(I think this was published previously on kragen-tol.)

Last night, I encoded some messages in ASCII by hand, including error-correction coding: like the printer dots Seth Schoen decoded, matrices of 8 columns of 8 bits each, with an extra column and row for parity. (I used even parity, but odd would have been a better choice.) Matrix parity lets you correct single-bit errors: a single-bit error will show up as a parity error in both the row parity and the column parity. And it's simple enough and spatial enough that you can both detect and correct errors by hand.

This got me to thinking about what kind of more powerful error-correction coding might be feasible to perform by hand, and I think I came up with one, which at first glance appears to be related to the Hadamard transform, but in the end seems to be something simpler.

Existing Hadamard-code work is quite different

For many years, Hadamard matrices have been used for error-correction coding by the simple expedient of using a row of the matrix for each codeword; since the rows are perfectly uncorrelated, they have a large Hamming distance, and the Fast Hadamard Transform of the received (possibly corrupted) codeword gives you the Hamming distance from all the N codewords in 2 N lg N additions and subtractions. (This is now apparently obsoleted by better codes.)

A problem with this for human use is that it's not just some checksum symbols you glom onto your data; it makes your data totally unrecognizable.

I want to discuss a different application of Hadamard matrices for error-correction codes. As far as I can tell, this is not related to "Hadamard-Craigen error correcting codes" (Craigen 2002), and in the unlikely event that these codes become popular, I hope that nobody decides to call them "Hadamard-Kragen codes".

The Hadamard transform and its relevance briefly explained

The Hadamard transform of a vector is a "holographic representation" of the vector in the sense that the vector can be recovered from it (by repeating the Hadamard transform and dividing by the determinant) and that every value in the Hadamard transform is affected by every element in the original vector. This means that any local change in the original vector results in a global change in the Hadamard-transformed vector --- in the case of the Hadamard transform, *every* element changes.

(For simplicity, I'm going to consider only dimension-`2**n` Hadamard matrices produced by the Sylvester method here, although

others exist.)

The Hadamard transform is particularly interesting among such holographic representations because the Fast Hadamard Transform enables computing the HT very efficiently.

The fast Hadamard transform of a vector or characters, illustrated

So suppose you have a vector of characters, say, 'Hadamard'. You compute the Fast Hadamard Transform of this vector:

```
def hadamard(vec):
    if len(vec) == 1: return vec
    pairs = zip(hadamard(vec[:len(vec)/2]),
                hadamard(vec[len(vec)/2:]))
    return ([lv + rv for lv, rv in pairs] +
            [lv - rv for lv, rv in pairs])

>>> hadamard(map(ord, 'Hadamard'))
[786, 4, -36, -30, -54, -48, -20, -26]
```

Note that a local change in the input produces a linear global change in the output:

```
>>> hadamard(map(ord, 'Hadamarc'))
[785, 5, -35, -31, -53, -49, -21, -25]
>>> hadamard(map(ord, 'Hacamarc'))
[784, 4, -34, -30, -54, -50, -20, -24]
```

And you can reconstruct the original input vector from the Hadamard-transformed version:

```
>>> ''.join(chr(c/8) for c in hadamard(hadamard(map(ord, 'Hadamard'))))
'Hadamard'
```

And, naturally, local errors in the Hadamard-transformed version result in linear global errors in the result:

```
>>> ''.join(chr(c/8) for c in hadamard([786,4,-36,-30,-54,-48,-20,-26]))
'Hadamard'
>>> ''.join(chr(c/8) for c in hadamard([778,4,-36,-38,-54,-48,-20,-26]))
'Fad_karb'
>>> ''.join(chr(c/8) for c in hadamard([700,4,-36,-38,-54,-48,-20,-26]))
'<WZUaWhX'
```

How to use this for a human-friendly error-correction code

So suppose you want to be able to correct errors. You can pick some subset of the Hadamard transform to transmit along with your raw data. If there's a single-character error, it will show up in every transformed value you transmit, but with varying signs; each sign after the first gives you one bit of information about where the error is --- *exactly* one bit, I thought, since the rows of the Hadamard matrix are *perfectly* uncorrelated; so you should be able to localize the

error with lg N bits. (That turns out to be false, as demonstrated below; it matters a lot which rows you pick.) In fact, since the whole transform is linear, you should be able to just run the inverse transform on your error signal and get the location of the error.

Using the inverse transform on arbitrary outputs to localize errors works poorly

Let's try:

```
>>> orig = hadamard(map(ord, 'Hadamard'))[:3]
>>> computed = hadamard(map(ord, 'Hagamard'))[:3]
>>> [computed_i - orig_i for computed_i, orig_i in zip(computed, orig)]
[3, 3, -3]
>>> hadamard([computed_i - orig_i
              for computed_i, orig_i
              in zip(computed, orig)] + [0] * 5)
[3, -3, 9, 3, 3, -3, 9, 3]
```

Not quite. We got a nice peak in the transformed error signal at the location of the actual error, it's true, but we got an equally big one elsewhere. It doesn't work with four values either:

```
>>> orig = hadamard(map(ord, 'Hadamard'))[:4]
>>> computed = hadamard(map(ord, 'Hagamard'))[:4]
>>> hadamard([computed_i - orig_i
              for computed_i, orig_i
              in zip(computed, orig)] + [0] * 4)
[0, 0, 12, 0, 0, 0, 12, 0]
```

It does work with five:

```
>>> computed = hadamard(map(ord, 'Hagamard'))[:5]
>>> orig = hadamard(map(ord, 'Hadamard'))[:5]
>>> hadamard([computed_i - orig_i
              for computed_i, orig_i
              in zip(computed, orig)] + [0] * 3)
[3, 3, 15, 3, -3, -3, 9, -3]
```

And of course the first value tells us the sign and magnitude of the actual error:

```
>>> computed[0] - orig[0]
3
```

What if we pick different values, like the ones from the end? We still need five of them to locate the error:

```
>>> orig = hadamard(map(ord, 'Hadamard'))[-3:]
>>> computed = hadamard(map(ord, 'Hagamard'))[-3:]
>>> computed
[-45, -23, -29]
>>> hadamard([0] * 5 + [computed_i - orig_i
                        for computed_i, orig_i
                        in zip(computed, orig)])
[-3, -3, 9, -3, 3, 3, -9, 3]
```

```
>>> orig = hadamard(map(ord, 'Hadamard'))[-4:]
>>> computed = hadamard(map(ord, 'Hagamard'))[-4:]
>>> hadamard([0] * 4 + [computed_i - orig_i
                        for computed_i, orig_i
                        in zip(computed, orig)])
[0, 0, 12, 0, 0, 0, -12, 0]
>>> orig = hadamard(map(ord, 'Hadamard'))[-5:]
>>> computed = hadamard(map(ord, 'Hagamard'))[-5:]
>>> hadamard([0] * 3 + [computed_i - orig_i
                        for computed_i, orig_i
                        in zip(computed, orig)])
[-3, 3, 15, -3, -3, 3, -9, -3]
```

How about a different error?

```
>>> computed = hadamard(map(ord, 'Hadamerd'))[-5:]
>>> hadamard([0] * 3 + [computed_i - orig_i
                        for computed_i, orig_i
                        in zip(computed, orig)])
[-4, -12, 4, -4, -4, 20, 4, -4]
```

How about an error in the transformed values? That produces a distinctly different pattern:

```
>>> computed = [0] + orig[1:]
>>> hadamard([0] * 3 + [computed_i - orig_i
                        for computed_i, orig_i
                        in zip(computed, orig)])
[30, -30, -30, 30, 30, -30, -30, 30]
```

It seems like it ought to be possible to correct multiple errors.

How about a longer text? Correcting one error in eight characters by appending five error-check symbols is not at all impressive. Can we correct an error in a 64-character text with 11 Hadamard results? (Not the way I've been doing it, but see later sections for successfully doing it with 7.)

```
>>> text='correct an error in a 64-character text with 11 Hadamard results'
>>> len(text)
64
>>> orig = hadamard(map(ord, text))[:11]
>>> errtext = text[:20] + 'x' + text[21:]
>>> errtext
'correct an error in x 64-character text with 11 Hadamard results'
>>> computed = hadamard(map(ord, errtext))[:11]
>>> orig
[5806, -74, 100, -304, 68, -576, -166, -78, -170, 310, 212]
>>> computed
[5829, -51, 123, -281, 45, -599, -189, -101, -147, 333, 235]
>>> hadamard([computed_i - orig_i
              for computed_i, orig_i
              in zip(computed, orig)] + [0] * 53)
[69, 23, 23, -23, 253, 23, 23, -23, -69, -23, -23, 23, 115, -23,
-23, 23, 69, 23, 23, -23, 253, 23, 23, -23, -69, -23, -23, 23,
115, -23, -23, 23, 69, 23, 23, -23, 253, 23, 23, -23, -69, -23,
```

```
-23, 23, 115, -23, -23, 23, 69, 23, 23, -23, 253, 23, 23, -23,
-69, -23, -23, 23, 115, -23, -23, 23]
```

No, it identifies four equally likely locations for the error. 13?

```
>>> orig = hadamard(map(ord, text))[:13]
>>> computed = hadamard(map(ord, errtext))[:13]
>>> hadamard([computed_i - orig_i
              for computed_i, orig_i
              in zip(computed, orig)] + [0] * 51)
[69, -23, -23, -23, 299, 23, 23, 23, -69, 23, 23, 23, 69, -23,
-23, -23, 69, -23, -23, -23, 299, 23, 23, 23, -69, 23, 23, 23, 69,
-23, -23, -23, 69, -23, -23, -23, 299, 23, 23, 23, -69, 23, 23,
23, 69, -23, -23, -23, 69, -23, -23, -23, 299, 23, 23, 23, -69,
23, 23, 23, 69, -23, -23, -23]
```

No, mysteriously the new Hadamard results didn't give us any new information about the location of the error; we still have four equally likely locations. It's as if the relevant Hadamard rows are somehow correlated with the basis comprised of the rows we have already, though not any of those rows individually.

It turns out that better choices are available:

```
>>> orig_full = hadamard(map(ord, text))
>>> computed_full = hadamard(map(ord, errtext))
>>> orig = orig_full[:11] + [0] * 51 + orig_full[-2:]
>>> computed = computed_full[:11] + [0] * 51 + computed_full[-2:]
>>> hadamard([computed_i - orig_i
              for computed_i, orig_i
              in zip(computed, orig)])
[115, 23, -23, -23, 207, 23, 69, -23, -115, -23, 23, 23, 161, -23,
-69, 23, 23, 23, 69, -23, 299, 23, -23, -23, -23, -23, -69, 23,
69, -23, 23, 23, 23, 23, 69, -23, 299, 23, -23, -23, -23, -23,
-69, 23, 69, -23, 23, 23, 115, 23, -23, -23, 207, 23, 69, -23,
-115, -23, 23, 23, 161, -23, -69, 23]
```

That reduces the choices to two. What if we choose by a different approach?

```
>>> orig = orig_full[::5]
>>> len(orig)
13
>>> computed = computed_full[::5]
>>> orig_expanded = [0] * 64
>>> orig_expanded[::5] = orig
>>> orig_expanded
[5806, 0, 0, 0, 0, -576, 0, 0, 0, 0, 212, 0, 0, 0, 0, 86, 0, 0, 0,
0, 438, 0, 0, 0, 0, -108, 0, 0, 0, 0, -220, 0, 0, 0, 0, 4, 0, 0,
0, 0, -358, 0, 0, 0, 0, 96, 0, 0, 0, 0, -102, 0, 0, 0, 0, 120, 0,
0, 0, 0, 406, 0, 0, 0]
>>> computed_expanded = [0] * 64
>>> computed_expanded[::5] = computed
>>> hadamard([computed_i - orig_i
              for computed_i, orig_i
              in zip(computed_expanded, orig_expanded)])
```

```
[69, 161, -23, 161, 23, -69, 23, 23, 23, -161, 23, 115, -23, -23,
69, -23, -23, 69, -23, -23, 299, 23, 23, 23, 23, 23, -69, 23, -23,
-23, -23, 69, -23, 161, -23, -115, 23, 23, -69, 23, 23, 115, 115,
115, 69, -23, 69, -115, -23, -23, 69, -23, 23, 23, 23, -69, -69,
23, -69, 115, -23, 69, 253, 69]
```

That has a max, 299, at the right spot, with "only" 13 Hadamard-transformed values to give us six bits of error-location information.

So, although this can be made to work, it doesn't perform as well as I had hoped, and I didn't really understand why. I did find out how to stop it; see later sections.

Using the assumption that there's just one error doesn't help

But that approach is, in essence, attempting to find an estimate of an *arbitrary* difference between the two vectors. But we're assuming that the difference here comes from a transmission error corrupting a single letter.

What if we just use the signs of the differences, and try to figure out which single input position could have caused that pattern of signs? That doesn't work either with poor choices of rows:

```
sgn = lambda x: 1 if x > 0 else 0 if x == 0 else -1
matrix = lambda n: [hadamard([i==j for i in range(n)]) for j in range(n)]
def find(prefix, items):
    for ii, item in enumerate(items):
        if item[:len(prefix)] == prefix: yield ii

for i in range(64):
     print i, list(find([sgn(computed_i - orig_i)
                         for computed_i, orig_i
                         in zip(computed_full[:i], orig_full[:i])],
                        matrix(64)))

0 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
   18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
   34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
   50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63]
...
2 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
   34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62]
3 [0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60]
4 [0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60]
5 [4, 12, 20, 28, 36, 44, 52, 60]
...
9 [4, 20, 36, 52]
...
17 [20, 52]
...
33 [20]
...
```

Picking particularly helpful rows of the matrix *does* help

Somehow, almost all of the additional rows we get are adding no new information. Rows 0, 1, 2, 4, 8, 16, and 32 apparently added a bit each. (I say row 0 because we need the information about which direction the change perturbed the input in order to interpret the others.) What if we use just those rows?

```
select = lambda x: [x[i] for i in [0, 1, 2, 4, 8, 16, 32]]
>>> len(set(tuple(select(row)) for row in matrix(64)))
64
```

This looks promising! Compare to:

```
>>> len(set(tuple(row[:5]) for row in matrix(64)))
8
```

And it works, see?

```
>>> list(find([sgn(computed_i - orig_i)
               for computed_i, orig_i
               in zip(select(computed_full), select(orig_full))],
              [select(row) for row in matrix(64)]))
[20]
```

So there we have seven Hadamard values that successfully localized the error using only their signs. (If the change were negative, we'd've had to try `-sgn` as well as `sgn`; you could presumably avoid that by using the pattern of sign *changes* instead of signs). What if we use the reverse-transform approach with these values instead of the arbitrary ones we were using before?

```
def expand(vals):
    rv = [0] * 64
    for val, pos in zip(vals, [0, 1, 2, 4, 8, 16, 32]): rv[pos] = val
    return rv

hadamard(expand(computed_i - orig_i
                for computed_i, orig_i
                in zip(select(computed_full), select(orig_full))))

[69, 23, 23, -23, 115, 69, 69, 23, 23, -23, -23, -69, 69, 23, 23,
-23, 115, 69, 69, 23, 161, 115, 115, 69, 69, 23, 23, -23, 115, 69,
69, 23, 23, -23, -23, -69, 69, 23, 23, -23, -23, -69, -69, -115,
23, -23, -23, -69, 69, 23, 23, -23, 115, 69, 69, 23, 23, -23, -23,
-69, 69, 23, 23, -23]
```

And that does indeed have a clear peak of 161 at position 20, which is where the error is.

What if we have a different error? That seems to work too:

```
>>> errtext = text[:35] + '!' + text[36:]
>>> errtext
'correct an error in a 64-character !ext with 11 Hadamard results'
>>> computed_full = hadamard(map(ord, errtext))
```

```
>>> select(computed_full)
[5723, 9, 183, -15, -253, -55, 5]
>>> select(orig_full)
[5806, -74, 100, 68, -170, 28, -78]

...     hadamard(expand(computed_i - orig_i
...                     for computed_i, orig_i
...                     in zip(select(computed_full), select(orig_full))))

[-83, -249, -249, -415, 83, -83, -83, -249, 83, -83, -83, -249,
249, 83, 83, -83, 83, -83, -83, -249, 249, 83, 83, -83, 249, 83,
83, -83, 415, 249, 249, 83, -249, -415, -415, -581, -83, -249,
-249, -415, -83, -249, -249, -415, 83, -83, -83, -249, -83, -249,
-249, -415, 83, -83, -83, -249, 83, -83, -83, -249, 249, 83, 83,
-83]
```

And indeed there's a negative peak at position 35, where all seven differences of 83 ('t' - '!') line up with the same sign, producing -581. And using the approach of just looking at the signs? That works too, but since it's a negative change, we have to negate the signs:

```
>>> list(find([sgn(computed_i - orig_i)
...                    for computed_i, orig_i
...                    in zip(select(computed_full), select(orig_full))],
...                   [select(row) for row in matrix(64)]))
[]
>>> list(find([-sgn(computed_i - orig_i)
...                    for computed_i, orig_i
...                    in zip(select(computed_full), select(orig_full))],
...                   [select(row) for row in matrix(64)]))
[35]
```

This is very far from a proof that the method works, but it's suggestive. Here are the rows of `select(matrix(64))`:

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

[1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1,
-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1,
1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1,
-1, 1, -1, 1, -1, 1, -1]

[1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1,
-1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1,
-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1,
1, -1, -1, 1, 1, -1, -1]

[1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1,
1, -1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1,
-1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1,
1, 1, 1, -1, -1, -1, -1]

[1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1,
```

```
1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1,
1, 1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, -1,
-1, -1, -1, -1, -1, -1, -1]

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1]

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1]
```

These appear to be precisely the most regular rows, the ones with sequencies 0, 63, 31, 15, 7, 3, and 1. You certainly don't need a Hadamard matrix to compute those! It's closely related to the matrix parity code I started with --- but instead of XOR, you're using addition, and instead of two dimensions, you're in 6, and instead of separately recording the totals modulo whatever of two 5-dimensional hyperplanes, you're recording only the difference between them.

So maybe this is already known under some other name!

Using finite fields instead of the integers?

I did all the above in the ring Z, the integers, which has the problem that you need a potentially unbounded number of bits to represent its members.

All of this also works over finite fields such as Z/7, the integers mod 7, as well as the integers themselves. In fact, it isn't even necessary to have a full field, which is why it's possible to do it in Z at all; you only need to be able to divide by the power of two that is your vector length. Unfortunately, this eliminates the most desirable group for computation, Z/256 or in general `Z/2**n`, since it will lose the upper bits of each character.

One option would be to use Z/257, the integers mod 257. Then you can represent the Hadamard transform results in a byte each, plus a usually-empty list of which transform results are equal to 256; or in 9 bits each.

Another somewhat more computationally expensive option, which might avoid that additional unnecessary message expansion, is to represent the entire message in some other finite field: start by interpreting it as a large number in base 256, then transform it into a sequence of digits in e.g. Z/7 or Z/257, then do the Hadamard computation in that field.

Using a field with fewer members makes the message longer, which improves the ratio lg N/N, which I strongly suspect is proportional to the number of Hadamard results you need to include to localize and correct a single error.

How do you find the peaks in the transformed error signal in Z/p, where big numbers can wrap around to be small ones again, and division can easily produce big numbers? I have no idea. But the sign-sequence-lookup approach should work, even though "sign" itself is not a well-defined concept in Z/p.

Topics

Constant time sets for pixel painting

Kragen Javier Sitaker, 2017-02-07 (2 minutes)

There's a data structure for representing sets of small integers, up to M integers less than N, using a count variable and two arrays, A of size M and B of size N. The membership test is

```
mem(i)  ⇐  B[i] < count  ∧  A[B[i]] == i
```

from which you can derive the constant-time item insertion, item deletion, start-unordered-iteration, iterate-next-item, and set-to-empty-set operations. Additionally the array contents do not need to be initialized, so creating a new such set is a constant-time operation (although not, I suspect, in standard C, since I think reading the uninitialized data in B[i] is undefined behavior).

You can also use this structure as a sparse array, either adding an additional array of data-values parallel to A or by making A an array of (small-int, data-value) tuples. And in particular I was thinking that this could be useful for scanline rendering of overlapping or self-intersecting polygons. The idea is that, to compute a scan line, you maintain a sparse array of pixel value changes (polygon boundaries), and the resulting scan line is the prefix sum of that sparse array.

This requires efficient sequential (rather than unordered) access to the items of the set, but this is readily provided by running a sorting step after you're done storing boundaries into the array, but before you start generating pixels. It's not quite constant-time per drawing operation, but it's pretty close.

(Asymptotically speaking, it would be better to iterate over the B array, since you need to visit every pixel anyway. In practice, this is probably silly.)

By using subpixel coordinates as the indices in B, you should be able to get decent antialiasing in the X dimension at little additional cost.

I don't know if you can extend this to painting translucent or gradient-filled polygons; that seems like it would require a nearest-neighbor kind of test to find out what color a given range currently was.

Topics

High-risk behavior in context

Kragen Javier Sitaker, 2007 to 2009 (5 minutes)

Most people in the US think riding in a car without a seatbelt is reckless, to the point that it should be prohibited by law — not to speak of activities like bicycling without a helmet, skydiving, skateboarding on busy streets, and so on.

The WHO's Burden of Disease Report from December 2004 gives death rates for various causes for various countries. Some selected data:

```
W149 III A Unintentional Injuries:      USA:  36.6; Argentina:  33.8
W150 III A 1 Road traffic accidents:    USA:  15.5; Argentina:  11.5
W157 III B 1 Self-inflicted injuries:   USA:  10.3; Argentina:  10.2
W060 II A Malignant neoplasms (cancer): USA: 191.9; Argentina: 157.6
W107 II G 3 Ischaemic heart disease:    USA: 176.8; Argentina:  90.3
```

These are death rates per 100 000 population for 2002. These numbers are affected somewhat by the age distribution of the countries; some of the causes of death are much more prevalent among the old than among the young, such as ischemic heart disease. So they will tend to be very low in countries where more of the population is young, either because people die young or because the birth rate has recently increased a lot (or infant mortality recently decreased a lot). Other causes of death, such as traffic accidents, affect various age groups more evenly, although there is still some variation.

I think the numbers might be more graspable as MTTFs, "mean time to failure", where "failure" in this case means "death". If you divide 100 000 person-years by the various causes of death, you get life expectancies from a fantasy world where everyone stays the age they currently are until they die, and are divided into groups only affected by a single cause of death:

```
W149 III A Unintentional Injuries:      USA: 2730; Argentina: 2960
W150 III A 1 Road traffic accidents:    USA: 6450; Argentina: 8700
W157 III B 1 Self-inflicted injuries:   USA: 9710; Argentina: 9800
W060 II A Malignant neoplasms (cancer): USA:  521; Argentina:  635
W107 II G 3 Ischaemic heart disease:    USA:  566; Argentina: 1110
```

These numbers are interesting for three reasons.

First, they give a sort of ceiling on the possible progress of medical science: we can probably defeat cancer through medical treatments, but eliminating suicide would require a kind of totalitarian mind control, and therefore is not ethically possible. But it seems plausible that we could raise average human life expectancy to several thousand years, merely by curing aging and other diseases.

Second, they provide a reasonable basis for comparing the two countries, risk-wise. Road traffic is slightly less dangerous in Argentina than in the US, despite the crazy driving; perhaps that's because people take public transit more, or because there's less traffic on the long-distance highways.

Third, they provide a basis for estimating the life-expectancy cost of various risky activities and the life-expectancy benefits of various

prophylactic measures, if we heuristically estimate their effects by adjusting known risks. If something doubles your risk of cancer, then your "cancer life expectancy" falls to about 300 years, modulo age-group differences — which means that it has something very roughly like a one in seven chance of killing you, but your *actual* remaining life expectancy of 40 years or so will only shrink by five years or so. (Less, really, since cancer risk is somewhat weighted toward your older years anyway.) On the other hand, if it multiplies your risk of cancer by 20 — as heavy smoking does for one particular kind of cancer — it shortens your life expectancy by quite a bit. And what about riding motorcycles?

Well, supposing that motorcycles are five times as deadly as cars (to their riders), they might reduce your "traffic life expectancy" from 6500 years to 1300 years. That risk *is* more or less evenly distributed over your lifespan, so if you have 50 years left, your risk of dying from the motorcycle would be about one chance in 26, and your life expectancy decreases by about a year.

I still wear a seatbelt, because I don't really get much benefit from not wearing one. But, as a result of calculations like these, I go ahead and bicycle to places even when I don't have a bike helmet handy.

Topics

Heat exchangers modeled on retia mirabilia might reach 4 TW/m³

Kragen Javier Sitaker, 2014-07-16 (updated 2019-08-21) (14 minutes)

I have this idea for a vastly more effective and efficient heat exchanger.

The basic issue is that heat exchangers have a tradeoff between thermal resistance and fluid resistance. Ideally you want to drive both of these parameters as far as possible toward zero, but thinner tubes or turbulence mean less thermal resistance and more fluid resistance, while thicker tubes or laminar flow mean less fluid resistance and more thermal resistance.

Consider recuperator-type heat exchangers, which use two separate fluids, typically in a countercurrent configuration.

I suspect you may be able to cut this Gordian knot using fractal geometry like that of the lungs. The idea is to perform almost all of the heat exchange along a very convoluted (wrinkly) fractal surface whose Euclidean dimension is 2 but whose Hausdorff-Besicovitch dimension is nearly 3. This surface is full of tiny capillaries, and it separates two volumes from each other, which we can denominate the "arterial" and "venous" volumes. These spaces are not themselves empty; each one contains two separate, non-communicating sets of branching passages which branch down to the capillaries. In one of these sets of passages, one fluid passes from the arterial space to the venous space; in the other, a potentially separate fluid passes from the venous space to the arterial space.

Each of these two spaces looks a lot like a cauliflower, with a branching tree structure (or rather two of them) of passages feeding a very rough surface.

The two sets of passages are separate and do not mix, but they intertwine progressively more intimately until, at the capillary boundary, they are separated only by very thin walls. Nearly all of the heat is transferred during the passage through the capillaries, and nearly all of the fluid resistance is also due to this passage through these thin capillaries; the passages in the arterial and venous spaces are so much wider than the capillaries that they offer relatively little resistance. But, because the surface through which the capillaries pass is so enormously convoluted, the total cross-sectional surface area of the capillaries is immense, allowing the fluids to pass with relatively little resistance.

You could imagine the convoluted membrane filling, say, 80% of a one-liter volume with a 50-µm thickness penetrated by many 10-µm-diameter capillaries some 10 µm apart. Roughly estimating, this gives you 16 m² of membrane pierced by 4 m² of capillary cross-sectional area dived among 40 billion capillaries, 2 m² for each of the two fluids. A flow rate of 500mℓ/s amounts to 250 microns/s through these capillaries; that is, the fluid spends about 200ms in the capillary, during which time it is in somewhat more intimate thermal contact with the other fluid 20µm away through the capillary wall than it is with its own predecessor or successor fluid 50µm away through the capillary.

(Branching to 20 billion capillaries involves some 34 levels of branching if they are binary.)

The cross-section through which the heat must travel by conduction is some twenty times (5× length, 4× four directions if the capillaries are in a checkerboard through the membrane) the cross-section of the capillaries themselves, so 40 m². Aluminum has a thermal conductivity of some 205 W/m/K; multiplying that by 40m²/10μm gives us 820 MW/K, which is a dramatically enormous number.

Some random online pipe pressure drop calculator tells me that, given .000000000025 l/s (25 pℓ/s, 250 microns/s * 10 microns * 10 microns), pipe diameter of 0.010 mm, pipe roughness of 0.001 mm, and pipe length of .000050 m, the flow should be laminar and the pressure drop should be 0.05 millibar if the fluid is water. That's 5 Pa or a column height of 0.5 mm of water. God only knows if it's using some approximation or other that isn't valid at these scales, but it's somewhat reassuring.

These two figures together suggest that you should be able to pump considerably larger amounts of heat and fluids through this heat exchanger. If we consider a 5 kelvin loss acceptable, then maybe we can deal with 4100 MW in our one-liter heat exchanger. Water can hold perhaps 100kcal/kg; that gives us 9800 kg/s, which is 9800 ℓ/s, which would be 4.9 m/s through the 2m² of capillaries, which amounts to 0.49 microliters per second per capillary. Entering this into the same calculator provides me with an answer of 999.83 millibar, or 10.2 meters of water column height, which is to say, one atmosphere.

If we take this seriously, it would seem that we can probably nanofabricate a 4-gigawatt water-based countercurrent heat exchanger with a 5-kelvin temperature drop, with only one atmosphere of pumping pressure, in a single liter. I find these numbers so outlandish that they are hard to take seriously, but I wonder how close we could really get.

(In some cases you might have a much lower heat capacity per unit volume, for example with air, and desire a much lower delta temperature, like 5 millikelvins. I think this is also achievable.)

Previous work

Apparently Lingai Luo wrote a book on this in 2013, "Heat and Mass Transfer Intensification and Shape Optimization"; they proposed doing this in 2001. In 2002, Yongping Chen and Ping Cheng wrote a paper, "Heat transfer and pressure drop in fractal tree-like microchannel nets", surveying the existing work and proposing a two-dimensional coolant duct structure for cooling semiconductor chips. They have 275 citations in Google Scholar, and since then there has been a lot of work in convection in porous and complex structures, including a 2004 book by Bejan, Lorente, and others. It contains this remarkable passage:

> Tree-shaped flows in balanced counterflow are a prevailing flow structure in subskin vascularized tissues (Weinbaum and Jiji, 1985; Bejan, 2001). The purpose of the intimate thermal contact between the streams in counterflow is to minimize the leakage of heat (an enthalpy current) along the counterflow, from the warm end to the cold end. The counterflow provides thermal insulation in the flow direction: this insulation effect has its origin in the minimization of thermal

resistance in the direction perpendicular to the streams (Bejan, 1979b, 1982). This special feature, and the fact that the streamwise leakage of heat vanishes as the thermal contact between streams becomes perfect, is the reason why the balanced counterflow is the best arrangement from the point of view of minimizing heat transfer irreversibilities.

Unfortunately, although he arranges his flows dendritically, Bejan's work does not seem to contemplate distributing the capillaries themselves over a fractal surface, although he alludes to lungs at some point.

He also describes our lungs:

The alveoli act as the primary gas exchange units of the lungs. It is estimated that an adult has approximately 600 million alveoli with a surface area for gas exchange of about 75 m^2, which are perfused by more than 2000 km of capillaries (see Section 6.2). ... In order to optimize its function (see Section 4.11), the airway tree exhibits 23 levels of bifurcations after the trachea (Weibel, 1963).

Note that this 75 m^2 in a few liters is comparable to the exchange area I described above for the heat exchanger.

That topology-optimization guy in Scandinavia has been using topology optimization to design heatsinks using CFD simulation of convection. Understandably, they come out dendritic.

Luo's book is about "process intensification", which she defined in 2001 as "enhancement of the density of flux transferred between two phases through an interface", which covers the heat-exchanger thing above as well as many other possibilities. Its chapter 4 is all about heat exchangers, and it says:

Microchannel heat exchangers usually have a surface area density above about 10,000 $m^2 \cdot m^{-3}$ (Shah 1991). One typical example in nature is the human lungs, as a very high performance ultra compact heat and mass transfer system which have a surface area density of about 17,500 $m^2 \cdot m^{-3}$.

The numbers I postulated above work out to about 80,000, so it isn't too far from what people were already achieving in 1991. A Karlsruhe project got 18000 MW/m^3 by these techniques, with a pressure drop of 4 bar (400 kPa), a 10-kelvin temperature jump, and a residence time of about 2ms, while my calculations above suggested you should be able to get 4 terawatts/m^3 (i.e. 4,000,000 MW/m^3), which suggests that my calculations are perhaps a bit optimistic but not entirely out of the ballpark. The Karlsruhe device, however, did not use a fractal geometry, and it used crossflow rather than counterflow.

Luo also points out that for microchannel heat exchangers you probably don't actually want to use a highly conductive material, because it conducts heat longitudinally in the wrong direction. You'd be better off with a highly insulating material, something silly like polyethylene! And this is far more important for the kinds of geometries I'm considering.

Luo does at some point start using fractal designs for her multi-scale distributors, but never for the capillary-bearing surface itself. She proposes a multi-scale branched tree structure, but don't give performance figures. She does cite da Silva et al. 2004 ("Constructal multi-scalar tree-shaped heat exchangers") and Zimparov et al. 2006 ("Constructal tree-shaped parallel flow heat exchangers"). The designs in Zimparov's and da Silva's papers are basically the same as the Chen and Cheng 2002 paper: essentially planar trees.

Luo's Chapter 7 shows the optimization of the distributor/collector

network from chapter 3 (which is basically the same as Chen and Cheng's) using cellular automata to improve flow, but never leaving two dimensions.

Sun, Huang, and Zhang did a paper in 2015 where they did a CFD analysis of the Chen and Cheng planar fractal heat exchanger.

Poltorak applied for a US patent in 2012 (publication 20120285660 A1, application US 13/106,640) which is more about branching heatsinks fractally to keep them from whistling. He's now formed a patent troll company to extort money with his patent, called Fractal Heatsinks Inc. But this is not really relevant.

As mentioned in Recuperator heat storage (p. 594), Galen discovered heat exchangers with this structure 1800 years ago dissecting animals, and in anatomy they are known as retia mirabilia. They are used both for heat transfer (for example, to prevent heat loss in the legs of sheep) and for mass transfer (for example, in all mammalian kidneys).

Other applications other than recuperator-type heat exchangers

Although heat exchangers themselves are very important — a perfect heat exchanger makes compressed-air energy storage 100% efficient, for example — there are other possible uses of this geometry.

Luo's book mentions a number of applications of "process intensification" beyond just heat exchangers, and Luo's and Bejan's books both go into some detail on mass transfer in lungs and kidneys. But in fact this kind of broccoli-like design can be used for a variety of other things as well:

- If you have just a single fluid, this kind of design provides a more efficient regenerator-type heat exchanger, although the advantage there over the traditional designs is smaller. You can use it with modern microencapsulated phase-change materials to get unprecedented regenerator performance.
- Catalytic chemical processes can, in some cases, benefit from mixing reagents uniformly and intimately as they pass over a catalyst. This design provides that possibility; the only missing piece is to run both fluids in a co-current direction and join their capillaries.
- Non-catalytic chemical processes, too. Consider feeding fuel (e.g. propane) and oxidizer (e.g. oxygen) through the two sets of passages at a controlled rate, mixing and heating evenly at the surface and producing a controlled hot blast of oxidation products on the other side.
- If some substance like water can diffuse through the walls between the capillaries, this design can work for mass transfer as well as heat transfer.
- If the substance you're transferring heat to is water, you can flash-boil it and get a steam explosion. A .22 LR pistol puts about 75 J of kinetic energy into its bullet in about 2 ms, a power of about 40 kW. The case on its cartridge is 15.6 mm long by 5.7 mm diameter, about 400 mm^3. That works out to about 94 GW/m^3 (94 MW/ℓ), so even the Karlsruhe heat exchanger mentioned above, at 18 GW/m^3, is only a factor of five below the power density of a handgun

cartridge, modulo the Carnot limit. This means you can totally make a steam-powered gun that's competitive in size and power to explosive-powered guns.

Topics

Needle binder injection printing

Kragen Javier Sitaker, 2019-08-05 (12 minutes)

Another atypical 3-D printing process I'd like to try out: injecting binder into a powder bed through an array of needles as they are withdrawn from the powder bed. Unlike normal powder-bed processes, this doesn't require depositing the powder layer by layer or smoothing each layer.

Basic single-needle process

You have a build chamber filled with loose powder. In the simplest form, there is a long hollow hypodermic needle sunk all the way through the build chamber. The needle is filled with a fluid binder, like those used in inkjet powder-bed 3-D printing processes. It is withdrawn gradually, during which time the flow of binder through them is alternately forced by pressure and stopped by a valve. This results in selectively depositing binder along the path of the needle, selectively solidifying the powder bed at the locations where binder is being extruded. When the needle is fully withdrawn, it is moved laterally and inserted in the new location, and then the process is repeated.

You need to inject enough binder to affect a radius in the powder bed equal to at least half the hole spacing. Assuming the powder-bed material is isotropic, this limits your resolution in the along-needle axis as well; for example, if the holes are 10 mm apart, you need to inject enough binder to spread 5 mm in all directions from the insertion point, which probably means you can't get much better than 10-mm resolution in the needle-withdrawal direction. You can probably control a small valve at 1 kHz or so (reed relays go up to 30 kHz, and inkjet print head pumps are controlled at some 5–20 kHz), so the limit on printing speed is then the needle withdrawal speed, unless you're withdrawing the needle at 10 m/s or more. This speed is probably limited by needle breakage.

Multi-needle variants

This can be done with many needles in parallel. Inserting sixteen parallel needles, moved as a single unit in a carriage, offers the possibility of printing sixteen times as fast as the single-needle process; additionally, if sixteen pixels in one dimension is adequate, the movement mechanics can support only two orthogonal axes rather than the three needed by the single-needle process.

More generally, if you have enough needles to cover an entire dimension of the build chamber, you can use only two orthogonal axes of movement. For example, with 2-mm needle spacing and 100 100-mm-long needles, you could print a 200 mm by 100 mm by 1000 mm print space with a single fast movement axis of a bit over 100 mm (probably moving the needles) and a single much slower 1000-mm movement axis (probably moving the powder bed).

If you have a larger number of needles, you can print with only a single axis of movement; for example, with 20-mm needle spacing and 144 needles of 120 mm each, you could print a 240 mm by 240 mm by 120 mm print volume in a single movement.

I suspect machines with large numbers of needles will tend to be unreliable, as the larger numbers of needles may experience clogging, breakage, and plastic flexion more often.

Layered hybrid

Ultimately the needle length will limit positioning precision, due to flexions from vibration and random powder-bed anisotropies or nonuniformities during insertion; reliability, due to needle breakage; and extrusion rate, if that isn't limited by valve speed, due to larger fluid friction in longer needles.

A more complex machine that combines some of the advantages of traditional powder-bed 3-D printers with the advantages of this design would deposit a thick layers rather than the usual thin layers, insert the needles only to the bottom of the new layer, and selectively bind that layer in 3-D. This permits the use of shorter needles at nearly the same printing speed. Some designs of machine may be able to use the needle carriage positioning to compact and level the powder bed.

Speed comparison

RepRap-descended FDM printers have a precision of about 100 μm and a deposition rate of about 20 mm/sec (times 500 μm times 300 μm, typically, although this depends on your print settings). If we approximate crudely, this amounts to a "voxel rate" of about 3000 voxels per second, but you only pay for the part of the build volume you're actually building things in.

This needle-binder-injection printing process can produce about 1000 voxels per second per needle, so with 16 needles, it should be about 16000 voxels per second. However, this includes the unused parts of the build volume. So that variant should be similar to RepRap-style printers in its productive capacity; the variants with more needles should be faster.

Inkjet-based powder-bed processes like those pioneered by Open3DP produce much higher voxel rates, on the order of 100k voxels per second.

Needle diameter and packing

Narrower needles are more flexible and, barring some kind of active control, will produce larger positioning errors. They are also more prone to breakage and produce more fluid friction. However, it's desirable for the powder-bed particles to be larger than the needle opening to limit clogging during insertion.

The space occupied by the needles after insertion probably needs to be occupied by void spaces between particles in the powder bed before insertion. This limits the needle diameter to a fraction of the hole spacing. This can be partly circumvented, in the designs that need more than a single insertion to complete a print, by spacing the needles on the carriage much further apart than the holes. For example, printing a 512×512 matrix of 1-mm-apart holes with 16 needles, the needles can be in a line in a 495-mm-long carriage spaced 33 mm apart rather than 1 mm apart, and can be moved laterally by 1 mm in between insertions rather than 16 mm, so that after printing a whole 513-mm-long slab of holes, each needle has made 32 adjacent

holes rather than 32 holes spaced 16 mm apart interspersed with the holes from the other needles.

(Variants: a hexagonal matrix is better than a square matrix, and staggering the needle insertion order somewhat, so that subsequent holes are not spatially adjacent, is probably better than making holes one next to the other.)

Needle design

To ameliorate the clogging tradeoff, perhaps the needle opening could be smaller than the bore through the center of the needle, unlike the usual practice with hypodermic needles. This would allow the needle to have the clog-avoidance capability of a very thin needle, but the stiffness and much of the fluid friction of a much thicker one.

A potential problem is that, once the needle is partly withdrawn, the empty bore behind it may channel binder being injected into what we hoped was higher up, allowing the binder to affect a much larger area than desired. It might be possible to ameliorate this somewhat by locating the binder injection ports on the side of the needle, some distance away from its tip, like the inflation needle for a basketball, rather than its end. Binder could still diffuse back into the channel from the powder bed, but the problem would be less serious.

Materials

Most of the material systems published by Open3DP should work, although the viscosity of the liquid binder may be a smaller consideration than it was for them. Not only might this reduce the need for alcohol, it opens up the potential of injecting more viscous binders such as sodium silicate.

Many of the candidate material systems described in 3-D printing by flux deposition (p. 466) and Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) are also applicable to this form of 3-D printing. Some of them will not work as described because there is no practical solvent to dissolve the binder in, but perhaps some of them can be made to work by depositing a soluble precursor (such as calcium sulfate instead of calcium hydroxide); by including all the reagents and a low-temperature organic binder such as carboxymethylcellulose in the powder bed, then injecting simply a solvent such as water, then removing unbound powder before firing; or by including all the reagents, and injecting a suitable solvent and/or catalyst to allow them to react.

However, as I've envisioned it here, this device can probably finish a print faster (in minutes), so faster-hardening binder systems may be worthwhile. And the needles have the possibility of injecting gases as well as liquids; two very interesting gases in this context are CO_2 and steam. CO_2 and heat applied with steam might be able to accelerate the hardening of slaked lime, which normally takes hours. And CO_2 injection is well-known as a way to harden sand that is stuck together with unhardened sodium silicate, acting within a few seconds. (Afterwards, you can wash off the unhardened sand and sodium silicate with water.) As I suggested in Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196), possible alternative sources of CO_2 include $CaCO_3$ and $NaHCO_3$.

Steam can deliver a great deal of heat very quickly to a fairly precisely located position, but of course it heats up the needle to 100°,

or more at greater-than-atmospheric presure. This means that the needle needs to be withdrawn quickly enough so that conduction from the needle surface is very small compared to the heat delivered by the condensing steam.

Particular cements that might be amenable to fast steam hardening include plaster of Paris, hygroscopic salts (such as NaCl, $CaCl_2$, sodium acetate, sodium silicate, or magnesium oxide), and organic gums and other organic binders (such as carboxymethylcellulose, gelatin, agar, guar gum, and gum arabic). Under pressure, steam condensation can reach high enough temperatures to melt many organic thermoplastics, but most of them are vulnerable to hydrolysis and must usually be dried before softening with heat.

Needle injection of nonpolar organic solvents like acetone, methyl ethyl ketone, ethyl acetate, or alcohol, either in liquid or vapor form, might be a reasonable way to cement a powder bed containing soluble organic plastics in powder form as a binder. It might also be an effective way to selectively deposit heat within the powder volume without subjecting fragile plastic molecules to aggressively-hydrolyzing water.

Either reactive gases like H_2S, Cl_2, or SO_2 or solvents might also be usable to provoke either hardening chemical reactions, or, in a reversal, the destruction of a previously solid binder and the selective conversion of remaining inert fillers into powder. This does, however, require that the solid material be sufficiently yielding and/or porous to get the needles into it in the first place. Selectively dissolving styrofoam with acetone, ethyl acetate, or gasoline is one example, though lacking in filler; EVA foam is another candidate material.

Topics

Material merits

Kragen Javier Sitaker, 2016-05-08 (6 minutes)

I was thinking about the relative costs of materials, and it occurred to me that there are three principal figures of merit on which you might consider using a material for mechanical engineering:

- Cost per tensile support: dollars per newton-meter before it breaks
- Cost per tensile stiffness: ???
- Cost per impact energy: dollars per joule

Of course, for non-mechanical and even manufacturing purposes, many other properties are relevant: transparency, color, smell, machinability, magnetic permeability, conductivity, dielectric constant, thermal conductivity, dielectric strength, melting point, electronegativity, heat capacity, chemical resistance, UV resistance, viscosity when molten, resistance to galling, lubricity, fatigue resistance, and so on. And even for mechanical purposes, ratios of weight or volume to tensile support, stiffness, and impact energy are of secondary importance; and there are other kinds of stiffness and strength other than tensile: shear and compressive, normally. Additionally, these fundamental properties give rise to emergent properties like Vickers hardness.

But mostly those three are the ones we care about.

Cost is an extensive quantity — twice as many grams of the same alloy costs twice as much — so, to take these ratios, we need to convert tensile strength, stiffness (Young's modulus), and impact resistance into extensive quantities too.

Cost per tensile support

Consider ASTM A36 structural steel, with a yield stress of 250 MPa. A square-millimeter fiber of it can thus support 250 N, about 25 kg weight. Suppose it costs US$0.60 per pound ($1.32/kg) and has a density of 7.8 g/cc; then it costs about 10.3 μ$ per cubic millimeter, which means that it costs 10.3 μ$ per millimeter to support 250 N.

Normalizing, this works out to about 41 μ$ per newton-meter. If you need to hang 1000 N one meter below a support, or 10kN 100 mm below, this steel will do it for you for 4.1¢.

This is just reinterpreting 250 MPa from 250 N / m² to 250 N m / m³, then dividing by the density to get 32 kN m / kg. Dividing the cost by that gets you to the 41 μ$ / (N m):

```
$ 0.60 / pound / (250 MPa / (7.8 g/cc))
```

Cost per tensile stiffness

The Young's modulus E has, like tensile strength, units of pressure; it's the stress that would stretch a cable to twice its natural length if it stayed Hookean over that range, which nothing does. A less fanciful, but more arbitrary, way of looking at it is that it's a million times the stress per microstrain.

If we consider the square-millimeter cable of A36 steel again, whose Young's modulus is about 200 GPa, it will elongate by a

microstrain under a force of 0.2 N. But we have a somewhat different situation. While the force needed to break the cable is independent of its length, that microstrain is a micron if it's a meter long, or ten microns if it's ten meters long. But if we make the cable ten times thicker instead of ten times longer, it becomes ten times stiffer rather than ten times less stiff.

How do we derive an extensive quantity from this, so that we can divide the cost per pound by it? I have no idea. I have to think about this.

Cost per impact energy

When our aforementioned square-millimeter cable of A36 steel is stretched, it stores energy in its elastic deformation. If deformed too much, beyond its tensile strength, it will deform plastically or break (sometimes the difference between these is important, but I'll gloss over it here). The ratio between its tensile strength (yield stress S in this case) and its Young's modulus is its elongation at break (or at yield), about 0.125%†, which is an intensive quantity; but in itself that doesn't tell us much.

† this differs by more than two orders of magnitude from the elongation at break in MatWeb, and ultimate tensile strength isn't that much higher than yield stress, so I'm probably wrong.

It turns out, though, that the energy it can store is also an intensive quantity — which ought not to be surprising in retrospect. If we integrate the force over the deformation distance up to the yield stress times the wire length, we get the energy that can be stored in the cable before breaking. This energy is proportional to the force, which is proportional to the cross-section of the cable and doesn't change with its length; it's also proportional to the distance, which is proportional to the length of the cable and doesn't change with its cross-section. The force $F = EAx/L$ is also proportional to the Young's modulus E: it's the cross-section A times the Young's modulus E times the distance x, divided by the length to get the strain. So we have $\int EAx/L\ dx$ from $x=0$ to $x=LS/E$, so the total energy is $\frac{1}{2}E\ A/L\ (LS/E)^2 = \frac{1}{2}ALS^2/E$. The AL part of this is just the volume of the wire, so $\frac{1}{2}S^2/E$ is the energy that can be stored per unit volume.

In the case of A36 steel, this gives us an impact energy (before plastic deformation) of about 156kJ/m³, or 20 J/kg, which works out to 6.6¢/J of elastic deformation at the US$0.60/pound price above:

```
$ 0.60 / pound / (1/(7.8 g/cc) * (250 MPa)^2/200GPa/2)
```

Topics

Use crit-bit trees as the fundamental string-set data structure

Kragen Javier Sitaker, 2013-05-17 (3 minutes)

DJB makes a good argument for using crit-bit trees as the fundamental string-set data structure: you get to use two words of memory per string, or three if you store the strings as pointers instead of inline in the nodes.

The trouble is, how do you generalize it? A hash table can store pointers to arbitrary objects as long as they support .hash() and .equals() methods. But crit-bit trees can apparently store only strings.

For a small set of strings, you could perhaps do even better. For up to 128 strings, say, you could use parallel arrays:

```
uchar left[128],  // left child pointer (as array index)
      right[128], // right child pointer
      nbits[128], // number of bits to skip from parent node bit offset
      start[128], // string #3 is contents[start[3]:start[3]+length[3]]
      length[128],
      contents[256];
```

These definitions support strings of up to 256 bytes total, with no shared prefix of 256 bits, 32 bytes, or more; the high bit being set in a pointer indicates a leaf node. Increasing `start` to 16 bits per entry allows your strings to be up to 255 bytes each, removing the aggregate limit.

An empty dict in CPython costs about 180 bytes. With a structure like

```
typedef struct { uchar left[N], right[N], nbits[N]; object *out[N]; } tinydict;
```

you have some 7 bytes per potential entry in your dict; that gives you Python's degree of overhead at N=25, so N=16 is probably a reasonable choice, using up 112 bytes per tinydict. (At which point you could save a little further space by using nibbles, but don't.)

An amusing thing about these parallel-array structures: like PHP's arrays, they have an implicit sequence to them, the sequence of array indices. You could imagine a routine that shuffles the order of treenodes around without changing the shape of the tree, thus providing an additional traversal order.

A larger `tinydict`:

```
typedef struct { u16 left[N2], right[N2], nbits[N2]; object *out[N2]; } smalldicto
o;
```

This can support up to 64Ki items.

What does the protocol for an `object` look like? Presumably you have a `.key()` method which returns a sequence of bytes — perhaps a lazy sequence, one or a few words at a time.

As an amusing exercise, perhaps many objects could be normally stored as strings, with a small cache (a few kilobytes) of objects built in a pointer-filled manner.

Topics

• Programming (p. 3658) (286 notes)
• Performance (p. 3621) (149 notes)
• Algorithms (p. 3310) (123 notes)

Burst computation

Kragen Javier Sitaker, 2017-03-20 (13 minutes)

Timesharing was originally intended to make computing power more accessible by allowing you to pay for your average usage rather than your peak usage, also promoting economies of scale through centralized computing facilities, in the same way you could publish classified ads in the newspaper from time to time without buying a web-fed offset press. In the day of the US$58 (in Argentina! retail!) 4-CPU-core 4-GPU-core 1200MHz Raspberry Pi 3, this might seem like an outdated concept. But there are still tasks that benefit from more computation than the Pi can provide.

(At some point in the 1970–1990 time frame, the economies of scale moved from the individual computers to the mass-production facilities for their integrated circuits, so the cheapest way to provide a lot of computing power is currently to buy a large number of some computing device that is produced in high volumes. Rather than amortizing the costs of building a powerful computer over many users, we amortize the cost of designing and making the masks for a powerful computer over many computers.)

Roy Longbottom benchmarked the Pi 3; he got 711 million Whetstone operations per second, in the neighborhood of 300 Whetstone megaflops (single precision), 2500 VAX MIPS, 180 Linpack double-precision megaflops, 486 Linpack single-precision megaflops with NEON, and 200 double-precision megaflops when memory-bandwidth limited (520 in 64-bit mode). Linux measures it at 38 BogoMIPS.

By comparison, the NVIDIA Pascal GP100 on a mezzanine P100 GPU Accelerator card, introduced in 2016, provides 5 double-precision teraflops, about the same as 25000 Raspberry Pi 3s. It isn't available in Argentina, but Amazon has it for about US$1500, which is about US$0.06 per Pi, showing that there still are *some* economies of scale in computation.

Moreover, I commonly run intensive computations where latency is important, but the duty cycle is low — my peak usage would ideally be much higher than my average usage. With the advent of per-minute Openstack vendors like Orange's Cloudwatt and per-request services like AWS Lambda, it seems like it might be quite feasible to spin up a temporary virtual supercomputer on demand.

Consider, hypothetically, that I am doing some interactive computations, like in IPython or whatever, and I would like to keep my interactive response time under a second; maybe I am doing 10 such computations in a minute, 10 such minutes in an hour, and 10 such hours in a day. And let's say that these computations are embarrassingly parallel, and each such computation involves about two minutes of calculation time on a GP100: 600 trillion floating-point operations.

It would in theory be possible to fulfill this wish by buying three million Raspberry Pi 3s for US$174 million, plus the networking and electrical equipment to harness them together. If we lower our aims to being able to complete 100 such computations per hour, we only need 83,000 Pis, costing only US$4.8 million.

A much more practical approach would be to buy NVIDIA GPUs and ATX PCs to plug them into, maybe two PCs per GPU; if this costs US$1750 per GPU, then we need two GPUs (US$3500) to provide the 36-second response provided by the five-million-dollar Pi cluster, or 72 GPUs (US$126,000) to provide the wished-for one-second response.

Cloudwatt charges €0.0102 per hour for its t1.cw.tiny-1 instances, comparable to DigitalOcean's smallest "droplet", and €0.0558 per hour for n1.cw.standard-1 instances, each with one virtual CPU; an "instance de type haute performance" n1.cw.highcpu-2 costs €0.0870 per hour and has two virtual CPUs. Supposedly these virtual CPUs are equivalent to one Xeon processor thread, but they don't seem to say what clock speed or generation of Xeon.

Suppose we're talking about the Xeon E5-2687W v3 Haswell-EP system Donald Kinghorn benchmarked at 788 gigaflops (Linpack, double precision) at 3.1 GHz in 2014. That's across two CPUs with 10 cores each. Then we should expect about 39 gigaflops per core, and maybe 20 gigaflops per core thread (assuming "hyperthreading" gives two threads per core). Then our 600 trillion floating-point operations work out to about 30000 vCPU seconds. This means that to get the computation results in a second, we need to spin up about 15000 n1.cw.highcpu-2 instances, then shut them down after they're idle for a minute, so we get billed for two minutes for each of them, 30000 minutes or 500 hours in all, €43.50.

Recall that this is €43.50 for one minute of computation (with a 17% duty cycle within that minute), and we're doing such minutes 100 times a day, so it works out to €4350 per day. This comes out to the US$126,000 (€117,000) cost of the 72-GPU cluster in only 27 days.

If we instead accept a 36-second response time, we only need about 420 n1.cpu.highcpu-2 instances, but we need them for ten hours a day, which works out to 420·10·€0.0870 = €365 per day, which works out to 320 days to add up to the cost of the cluster.

An instructive comparison would be the cost of a personal cluster of Xeons, since they might be a less efficient way to run your problem, and a more direct comparison to Cloudwatt's pricing.

I think a currently typical price for a 3.06GHz six-core Xeon X5675 CPU is currently US$215 including shipping. BOINC measured the E5-2687W mentioned above at 3.34 GFlops/core and the X5675 at 3.13 GFlops per core, and SETI@Home has a similar list , so the X5675 cores are probably more or less comparable to the ones Kinghorn was benchmarking. The X5675 is for an LGA-1366 socket, and a currently typical price for a motherboard like the SuperMicro X8DTN+ with two LGA-1366 sockets might be US$430, and it has on-board Ethernet. (I'm not 100% sure this motherboard will work for this processor, but a similar one should.) 4GiB of DDR3-1333 memory costs US$40 now, and 16GiB might be a reasonable amount to include in such a machine, so US$160.

So a Xeon machine with 12 Xeon cores on two CPUs with 16GiB of RAM might cost US$1020, not counting power supply, case, and labor. A cluster equivalent to 420 n1.cpu.highcpu-2 instances would have 210 cores and thus 18 machines, US$18,360 (comparable to the US$3500 GPU cluster); the 15000 instances would be 7500 cores and thus 625 machines, US$637,500 (comparable to the US$126,000

GPU cluster).

So, in effect, Cloudwatt lets you use a US$18k cluster for 10 hours a day for €365 a day (US$392/day), which pays for the cluster in 46 days, or a US$637,500 cluster for 100 or 200 minutes per day for €4350 a day (US$4650/day), which pays for the cluster in 137 days. The higher "efficiency" in the second case is because, in our scenario, the duty cycle is lower for the more powerful cluster — you only have instances spun up for 10 or 20 minutes out of each of the 10 hours you're using the thing.

Nevertheless, Cloudwatt's pricing is so high that they are only a good choice if you are experimenting with computation for a few days, even in an apples-to-apples comparison with CPU hardware bought outright. If you continue to do computation, you would be better off buying your own computer; the payback time is only one to four months, and the useful life of the computer is probably 18 months or more.

AWS Lambda charges per 100ms multiplied by RAM usage, specifically $0.00001667 per gigabyte-second, which I guess is 16.67 microdollars per gigabyte-second or 16.67 femtodollars per byte-second, plus $0.0000002 (0.2 microdollars) per request.

100ms is three orders of magnitude finer granularity than the per-minute billing of Cloudwatt, so you could imagine this would result in substantially improved costs.

Let's suppose you could farm out these one-second computations into 30000 AWS Lambda requests (maybe through four levels of request tree, each tree request farming out to 16 subrequests, or something) which each take one second and use 256MB of RAM during that time. That's 30000 seconds at 256MB, which works out to US$0.128 for the time for the computation, plus US$0.006 for the requests. At 10 computations per minute, 10 minutes per hour, 10 hours per day, that's US$128 per day. This is considerably better than Cloudwatt's US$4350 per day; it doesn't become more expensive than buying the US$126,000 72-GPU rig for almost 1000 days, which is probably longer than the depreciation time for the hardware, even in the current post-Moore era.

(There's the potential problem that even AWS Lambda functions in Java might suffer a serious performance penalty over native code.)

However, there's still a large efficient region in between Amazon's pricing and what it's economically feasible to provide. The 72-GPU rig in our scenario could support multiple users. If we expand it to a 144-GPU rig (US$252,000), then three users need to submit a computation during the same second for the response time to exceed the usual second.

It isn't obvious to me how to calculate the load statistics in closed form, but whipping up a quick numerical simulation, it seems like with repeated simulations with 10 users that all use the cluster during the same hour, typically about 30 of their 1000 requests (10 per user; 3%) will be submitted during a second which has two or more other requests. This lowers the cost per user to US$25,200. So you could probably charge each user US$50 per day (US$5 per hour) and still come out ahead, assuming answering only 97% of requests in under 1½ seconds (answering most requests in 500ms) is acceptable. Scaling up the cluster further should allow lower costs per user with lower numbers of slow requests.

Note that the cluster could produce other value streams as well; the 10,000 requests per day still leave it 88% idle, so it could run lower-priority batch computations at the same time.

At a smaller scale, you could imagine 32 users using US$58 Raspberry Pi 3s (US$1900 total) submitting 250-millisecond tasks to a shared dual-GP100 rackmount box (US$3500), which has the number-crunching power of 50000 Pis. At the same 100-task-per-hour pace, nearly all of the tasks will be completed in under 500ms, though the Pi would need four hours to complete one of them — in effect, at this low utilization, each of the 32 users are getting nearly the full benefit of a GP100 or two, for only US$170 each.

(The costs for the rackmount box may be a bit low. The US$430 motherboard I linked earlier has 18 memory slots and supports up to 144 GiB of RAM; for another US$720 you could give it 72GiB of RAM, which would work out to 2¼ GiB per user, which might be a bit low. The motherboard, a single US$215 CPU, and 72GiB of RAM work out to US$1365, bringing the total cost of the shared computer to US$4365, adding an extra US$42 per user, US$136 of shared computer per user and US$194 per user in total.)

This is one reason Google (and, presumably, Facebook) is so effective at beating competition: it has been standard practice for more than a decade that everyday engineers can fire off a 10,000-CPU computation and see results within a few minutes — though not, at the time, less than a second.

Topics

Micro pubsub

Kragen Javier Sitaker, 2017-06-15 (8 minutes)

HTTP has an "ETag" attribute, short for "entity tag", to identify the current state of a resource, which can be usefully used in four ways:

• In the response to a GET or HEAD request, to inform the client what the current ETag of the resource is, making the other three uses possible;
• In the If-None-Match header of a GET request, to request that the server not send (a representation of) the resource if it hasn't changed (validating the cache to ensure it hasn't gotten too out of date);
• In the If-Match header of a PUT request, to execute a kind of atomic compare-and-swap for safe lock-free concurrency, preventing the request from executing if the resource has changed since it was last fetched;
• In the If-Match header of a byte-range GET request to ensure that the underlying file hasn't changed in a way that would make previous byte-range requests invalid.

These three kinds of concurrency control, however, are limited by the request-response nature of the REST architectural style, which can only propagate invalidation notifications by polling; thus, it faces an unavoidable painful tradeoff between expected notification latency and polling load, as described in Khare's dissertation. Various approaches to this have been suggested, including Khare's proposed WATCH method for HTTP and the RFC 7641 Observe option for CoAP, which are very similar to one another, but in practice the most common solutions today are HTTP long polling and WebSockets, which are effectively ways to tunnel arbitrary application-layer protocols on top of HTTP.

There are a number of potential problems with the straightforward implementation of the Observer pattern in a distributed system, modified only by a timeout, as proposed by Khare's dissertation and RFC 7641:

• There is no flow control to the stream of update messages. If the client has a 10kbps connection, while the server has a 100Mbps connection, then under circumstances of constant updates, a 500-byte subscription can result in the client's network connection being overwhelmed by four orders of magnitude more traffic than it can handle, until the subscription expires.
• One of the great benefits of REST is its stateless-server constraint; by storing all session state on the client, it enables easy horizontal scalability of servers (including by serving the same resource from many physical servers), allows servers to operate reliably and correctly even with extremely limited resources, simplifies failure recovery, and permits extremely large client-to-server ratios. These event-notification proposals, however, obligate servers to maintain unbounded amounts of session state on behalf of clients.
• Thus, the subscription itself becomes a long-lived resource on the server; to support subscription inspection and cancellation, we begin

to desire defined protocols and content-types to manage the subscription, not to mention authentication and access control lists. This adds undesirable complexity not only to the server but also to the protocol suite.

• A typical factor we examine for network deployments nowadays is their DoS potential, as measured by their amplification factor — if an attacker forges a request from a victim, how much traffic can they direct to the victim? These proposals have potentially very large DoS potential.

I would like to suggest a minimal extension to HTTP and similar protocols which covers most publish-subscribe use cases and largely solves the above problems.

The solution

The client includes a new ETag-Change-To header which includes a webhook URL. The server is free to ignore this header or to add the webhook URL to a set of observers it maintains for the current ETag of the resource being requested. When the resource's state changes, it sends the URL of the resource in an HTTP request to each webhook in the observer set for the old ETag, then discards that observer set. This allows the client, if they are still interested, to immediately retrieve the new state of the resource (possibly resubscribing), while bounding the maximum possible traffic to one in-flight message at a time, and the maximum possible cost imposed on the server to sending a single such message.

By atomically adding the webhook to the set associated with the ETag associated with the representation that was actually retrieved, we ensure that no updates happen after the retrieval but before the subscription.

In contexts like CoAP, protocols other than HTTP might be more appropriate for delivering these cache invalidation notifications. For example, an additional CoAP Response message — as with the RFC 7641 Observe option, but lacking the payload — may be a perfectly adequate solution, since CoAP imposes no constraints on how far in the future that message can be sent; or a CoAP Request message analogous to the HTTP request to a webhook may be more appropriate.

This approach is not better under all possible circumstances. At times, as mentioned in Khare's dissertation, it's desirable to have many update messages in flight at the same time; a stock-price logging application might prefer to see all the intermediate market prices of a stock, even when they are milliseconds apart and it is separated from the data source by hundreds of milliseconds of latency. However, there is a very broad range of applications for which the drawbacks of such streaming data are greater than their advantages.

Furthermore, in many cases, the appropriate response to such a cache invalidation message is not to update the cache, but rather merely to purge the cache, possibly generating further cache invalidation messages.

Since the server is not required to store the ETag-Change-To webhook, the server may suffer a hardware failure, and in any case the invalidation message may be lost, the client must still poll the resource at intervals determined by its maximum tolerable staleness time.

As an alternative that doesn't depart from the strict request-reply

discipline, we could instead use a GET request with a new When-None-Match header, which delays the response to the request until the resource's ETag no longer matches the supplied ETag. That is, if the ETag is different, the server will respond immediately, as if with If-None-Match; but if the ETag is the same, the server will simply acknowledge that the request has been received, and perhaps send a response at some later time.

Derived resources

Mediators like Shodouka, Google Translate, CritLink, and the Jupyter Notebook Viewer host virtual resources derived from other resources whose URLs are passed as query parameters. These resources could, in principle, be cacheable — they only change when the underlying derivation code changes, or when the source resources do. (In practice, the Jupyter Notebook Viewer is rather annoyingly aggressively cached.)

If such resources wanted to participate in this kind of push cache invalidation, they would probably be best served by propagating cache invalidations downstream when they received them, rather than refetching upstream resources to repeat their mediating transformation, perhaps many times, in order to keep up-to-date a resource that will perhaps never again be used.

Topics

• REpresentational State Transfer (p. 3684) (8 notes)
• Pubsub (p. 3670) (7 notes)
• HTTP (p. 3509) (4 notes)
• CoAP (p. 3380) (4 notes)

Achieving smooth curves in scanline image generation

Kragen Javier Sitaker, 2013-05-17 (1 minute)

I was thinking about chasing-the-beam-style raster-generation software in low memory, and how to achieve smooth curves.

Suppose you want to rasterize some kind of shape. You can approximate the border of the shape in a number of different ways: a polygon, a Bézier spline, etc. One particularly interesting way, to me, is as a tilewise polynomial.

Suppose you have divided your screen up into tiles of 256×256 pixels. The screen I'm looking at at the moment is 1024×600, so it would need four such tiles horizontally and three vertically, for a total of twelve. But coordinates within each tile are only 8 bits per coordinate, which means that you can do parallel computations over 16 tiles using SSE instructions.

Suppose you decide to represent the shape's border within a given tile as a polynomial, either for the horizontal axis in terms of the vertical or vice versa, along with limits. What degree of polynomial do you need?

Clearly if the shape border is of arbitrary complexity you need a degree-256 polynomial. But if you just want to hit three points in the tile, you only need a degree-2 polynomial. And if you need two points with specific derivatives — say, the top and bottom of the tile — you can do that with a degree-3 polynomial.

Topics

• Graphics (p. 3483) (91 notes)
• Math (p. 3564) (78 notes)
• Gradients (p. 3481) (8 notes)
• Splines (p. 3727) (6 notes)

Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation

Kragen Javier Sitaker, 2013-05-17 (updated 2013-05-20) (21 minutes)

(I think this was published previously on kragen-tol.)

Suppose you want to be able to execute the Visitor pattern as quickly as possible on some tree structure. You could compile your tree structure into executable code, each node a subroutine which invokes a method of the visitor object — traditionally each node type invokes a different method — and then passes the visitor object to each child node.

Using the "quaject" approach described in Henry Massalin's "Synthesis" thesis, and passing the pointer to the visitor object itself in %ebx, the code to invoke the appropriate method on the visitor might look like this:

```
    mov %ebx, %edx
    add $20, %edx
    call *%edx
    jc 1f
    ret
1:
```

Here we are checking the carry flag to see if the visitor object is requesting that we skip child nodes; if so, we return immediately.

The visitor "method" in this case is the code at offset 20 from the visitor base pointer; if its code is very short, it might be entirely inline at that point, but in many cases it will be longer, and only a call instruction will be present there. The called code will receive a pointer to the visitor itself in `%ebx`.

However, we probably want to pass some additional information to the visitor method; for example, if we're a variable node in a program AST, we might want to pass the name of the variable, or if we're an element node in a DOM, we might want to pass the tag name and attributes. So the entire call might look like this:

```
    mov $0xfe38d080, %eax
    mov %ebx, %edx
    add $20, %edx
    call *%edx
    jc 1f
    ret
1:
```

Following the call to the visitor method, we pass the visitor to the children. In the standard i386 GCC calling convention, `%ebx`, `%esi`, `%edi`, and `%ebp` are callee-saved registers, so if the visitor method follows this convention, we still have `%ebx` pointing to the visitor after it returns. So we can simply call our children one by one, implicitly passing them the visitor, then return:

```
    call 0xfe381000
    call 0xfe38d844
    call 0xfe391daa
    ret
```

For many applications, though, the visitor needs to take some action at the end of each node as well as the beginning. Perhaps it could inspect the child count at the beginning, maintaining its own stack of remaining child counts, and invoke the appropriate code when a child count reaches zero; but it's probably simpler to just notify it explicitly, by invoking another method on it before returning. This suggests putting the node pointer in a callee-saved register too, such as %ebp. With this additional modification, our example node looks like this:

```
  0:   55                   push   %ebp
  1:   bd 80 d0 38 fe       mov    $0xfe38d080,%ebp
  6:   89 da                mov    %ebx,%edx
  8:   83 c2 14             add    $0x14,%edx
  b:   ff d2                call   *%edx
  d:   73 0f                jae    0x1e
  f:   e8 fc 0f 38 fe       call   0xfe381010
 14:   e8 40 d8 38 fe       call   0xfe38d859
 19:   e8 a6 1d 39 fe       call   0xfe391dc4
 1e:   89 da                mov    %ebx,%edx
 20:   83 c2 1c             add    $0x1c,%edx
 23:   ff d2                call   *%edx
 25:   5d                   pop    %ebp
 26:   c3                   ret
```

(Note that this version has acquired an unfortunate limit of 127 bytes of child nodes, which is to say, 25 of them.)

That's 39 bytes for an executable representation of the data pointer 0xfe38d080 plus a variable-length list of child pointers that happens to contain three pointers; the most straightforward way to represent this

```
struct node {
  enum { document_node, element_node, text_node } nodetype;
  struct nodedata *data;
  int n_children;
  struct node *child[0];
};
```

would have needed 24 bytes, so making the data structure executable has cost us less than a factor of 2 in space.

(In the case where the children are sufficiently small, we can inline them, eliminating another 6 bytes (16%) of call and return.)

We can guarantee the visitor code a stronger calling convention than the usual; an executable tree built in the above fashion behaves in a very stereotyped way: of the caller-saved registers, it uses only %edx, plus %eax as an argument, so the visitor code is free to use %ecx as a private global variable, unless it calls some other code, in which case it must save it as usual. We can free up %edx too for the visitor's use as follows:

```
   0:   55                      push   %ebp
   1:   53                      push   %ebx
   2:   bd 80 d0 38 fe          mov    $0xfe38d080,%ebp
   7:   83 c3 14                add    $0x14,%ebx
   a:   ff d3                   call   *%ebx
   c:   73 0f                   jae    0x1d
   e:   e8 fc 0f 38 fe          call   0xfe38100f
  13:   e8 40 d8 38 fe          call   0xfe38d858
  18:   e8 a6 1d 39 fe          call   0xfe391dc3
  1d:   83 c3 08                add    $0x8,%ebx
  20:   ff d3                   call   *%ebx
  22:   5b                      pop    %ebx
  23:   5d                      pop    %ebp
  24:   c3                      ret
```

This has the side advantage of saving us two bytes (5%), but it also means the visitor method is invoked with a pointer to the visitor method rather than the visitor object; it is likely to need to subtract the method offset and add it back later. This seems like a fair tradeoff for giving it two private registers instead of one.

Additionally, the return value of the visitor method (that is, whatever it leaves in %eax) is either passed to the next visitor method called or is the return value of the entire node traversal.

For whatever it matters in today's world, this also means that the per-node overhead for tree traversal is 14 instructions, which seems pretty small. It may blow up your icache, though, and its single conditional jump might be mispredicted more often than the several in a more traditional implementation.

Consider this C implementation, listing generated as follows:

```
gcc -g -c -Wall -O4 -fomit-frame-pointer -Wa,-adhlns=nonquajectdom.lst nonquajectdom.c
```

```
   1                            .file   "nonquajectdom.c"
   2                            .text
   3                    .Ltext0:
   4                            .p2align 4,,15
   6                    _visit_node:
   7                    .LFB1:
   8                            .file 1 "nonquajectdom.c"
   1:nonquajectdom.c **** struct node {
   2:nonquajectdom.c ****   enum { document_node, element_node, text_node } nodetype;
   3:nonquajectdom.c ****   struct nodedata *data;
   4:nonquajectdom.c ****   int n_children;
   5:nonquajectdom.c ****   struct node *child[0];
   6:nonquajectdom.c **** };
   7:nonquajectdom.c ****
   8:nonquajectdom.c **** typedef int bool;
   9:nonquajectdom.c ****
  10:nonquajectdom.c **** struct visitor {
  11:nonquajectdom.c ****   void *visitor_data;
```

```
  12:nonquajectdom.c ****   bool (*document_callback)(struct visitor *self, struco
ot nodedata*);

  13:nonquajectdom.c ****   void (*document_end_callback)(struct visitor *self, so
otruct nodedata*);

  14:nonquajectdom.c ****   bool (*element_callback)(struct visitor *self, structo
o nodedata*);

  15:nonquajectdom.c ****   void (*element_end_callback)(struct visitor *self, sto
oruct nodedata*);

  16:nonquajectdom.c ****   void (*text_callback)(struct visitor *self, struct noo
odedata*);
  17:nonquajectdom.c **** };
  18:nonquajectdom.c ****
  19:nonquajectdom.c **** void visit_node(struct node *n, struct visitor *v);
  20:nonquajectdom.c ****

  21:nonquajectdom.c **** static inline void visit_children(struct node *n, struco
ot visitor *v)
  22:nonquajectdom.c **** {
  23:nonquajectdom.c ****   int ii;
  24:nonquajectdom.c ****
  25:nonquajectdom.c ****   for (ii = 0; ii < n->n_children; ii++) {
  26:nonquajectdom.c ****     visit_node(n->child[ii], v);
  27:nonquajectdom.c ****   }
  28:nonquajectdom.c **** }
  29:nonquajectdom.c ****

  30:nonquajectdom.c **** static void _visit_node(struct node *n, struct visitor o
o*v)
  31:nonquajectdom.c **** {
   9                            .loc 1 31 0
  10                            .cfi_startproc
  11                    .LVL0:
  12 0000 83EC1C                subl    $28, %esp
  13                    .LCFI0:
  14                            .cfi_def_cfa_offset 32
  15 0003 895C2410              movl    %ebx, 16(%esp)
  16 0007 89C3                  movl    %eax, %ebx
  17                            .cfi_offset 3, -16
  32:nonquajectdom.c ****   switch (n->nodetype) {
  18                            .loc 1 32 0
  19 0009 8B00                  movl    (%eax), %eax
  20                    .LVL1:
  31:nonquajectdom.c **** {
  21                            .loc 1 31 0
  22 000b 89742414              movl    %esi, 20(%esp)
  23 000f 89D6                  movl    %edx, %esi
  24                            .cfi_offset 6, -12
  25 0011 897C2418              movl    %edi, 24(%esp)
  26                            .loc 1 32 0
  27 0015 83F801                cmpl    $1, %eax
  28 0018 7476                  je      .L4
```

```
  29                            .cfi_offset 7, -8
  30 001a 7224                  jb      .L3
  31 001c 83F802                cmpl    $2, %eax
  32 001f 750D                  jne     .L1
  33:nonquajectdom.c ****   case element_node:

  34:nonquajectdom.c ****     if (v->element_callback(v, n->data)) visit_children
oo(n, v);
  35:nonquajectdom.c ****     v->element_end_callback(v, n->data);
  36:nonquajectdom.c ****     break;
  37:nonquajectdom.c ****   case document_node:

  38:nonquajectdom.c ****     if (v->document_callback(v, n->data)) visit_childre
oon(n, v);
  39:nonquajectdom.c ****     v->document_end_callback(v, n->data);
  40:nonquajectdom.c ****     break;
  41:nonquajectdom.c ****   case text_node:
  42:nonquajectdom.c ****     v->text_callback(v, n->data);
  33                            .loc 1 42 0
  34 0021 8B4304                movl    4(%ebx), %eax
  35 0024 891424                movl    %edx, (%esp)
  36 0027 89442404              movl    %eax, 4(%esp)
  37 002b FF5214                call    *20(%edx)
  38                    .LVL2:
  39                    .L1:
  43:nonquajectdom.c ****     break;
  44:nonquajectdom.c ****   }
  45:nonquajectdom.c **** }
  40                            .loc 1 45 0
  41 002e 8B5C2410              movl    16(%esp), %ebx
  42                    .LVL3:
  43 0032 8B742414              movl    20(%esp), %esi
  44                    .LVL4:
  45 0036 8B7C2418              movl    24(%esp), %edi
  46 003a 83C41C                addl    $28, %esp
  47                            .cfi_remember_state
  48                    .LCFI1:
  49                            .cfi_def_cfa_offset 4
  50                            .cfi_restore 7
  51                            .cfi_restore 6
  52                            .cfi_restore 3
  53 003d C3                    ret
  54                    .LVL5:
  55 003e 6690                  .p2align 4,,7
  56                            .p2align 3
  57                    .L3:
  58                    .LCFI2:
  59                            .cfi_restore_state

  38:nonquajectdom.c ****     if (v->document_callback(v, n->data)) visit_childre
oon(n, v);
  60                            .loc 1 38 0
  61 0040 8B4304                movl    4(%ebx), %eax
  62 0043 891424                movl    %edx, (%esp)
  63 0046 89442404              movl    %eax, 4(%esp)
```

```
 64 004a FF5204                 call    *4(%edx)
 65                     .LVL6:
 66 004d 85C0                   testl   %eax, %eax
 67 004f 7422                   je      .L8
 68                     .LVL7:
 69                     .LBB12:
 70                     .LBB13:
 25:nonquajectdom.c ****   for (ii = 0; ii < n->n_children; ii++) {
 71                             .loc 1 25 0
 72 0051 8B4308                 movl    8(%ebx), %eax
 73 0054 85C0                   testl   %eax, %eax
 74 0056 7E1B                   jle     .L8
 75 0058 31FF                   xorl    %edi, %edi
 76                     .LVL8:
 77 005a 8DB60000               .p2align 4,,7
 77      0000
 78                             .p2align 3
 79                     .L9:
 80                     .LBB14:
 81                     .LBB15:
 46:nonquajectdom.c ****
 47:nonquajectdom.c **** void visit_node(struct node *n, struct visitor *v)
 48:nonquajectdom.c **** {
 49:nonquajectdom.c ****   _visit_node(n, v);
 82                             .loc 1 49 0
 83 0060 8B44BB0C               movl    12(%ebx,%edi,4), %eax
 84 0064 89F2                   movl    %esi, %edx
 85                     .LBE15:
 86                     .LBE14:
 25:nonquajectdom.c ****   for (ii = 0; ii < n->n_children; ii++) {
 87                             .loc 1 25 0
 88 0066 83C701                 addl    $1, %edi
 89                     .LVL9:
 90                     .LBB17:
 91                     .LBB16:
 92                             .loc 1 49 0
 93 0069 E892FFFF               call    _visit_node
 93      FF
 94                     .LVL10:
 95                     .LBE16:
 96                     .LBE17:
 25:nonquajectdom.c ****   for (ii = 0; ii < n->n_children; ii++) {
 97                             .loc 1 25 0
 98 006e 3B7B08                 cmpl    8(%ebx), %edi
 99 0071 7CED                   jl      .L9
100                     .LVL11:
101                     .L8:
102                     .LBE13:
103                     .LBE12:
 39:nonquajectdom.c ****     v->document_end_callback(v, n->data);
104                             .loc 1 39 0
105 0073 8B4304                 movl    4(%ebx), %eax
106 0076 893424                 movl    %esi, (%esp)
107 0079 89442404               movl    %eax, 4(%esp)
108 007d FF5608                 call    *8(%esi)
```

```
 45:nonquajectdom.c **** }
 109                            .loc 1 45 0
 110 0080 8B5C2410              movl    16(%esp), %ebx
 111                    .LVL12:
 112 0084 8B742414              movl    20(%esp), %esi
 113                    .LVL13:
 114 0088 8B7C2418              movl    24(%esp), %edi
 115 008c 83C41C                addl    $28, %esp
 116                            .cfi_remember_state
 117                            .cfi_restore 3
 118                            .cfi_restore 6
 119                            .cfi_restore 7
 120                    .LCFI3:
 121                            .cfi_def_cfa_offset 4
 122 008f C3                    ret
 123                    .LVL14:
 124                            .p2align 4,,7
 125                            .p2align 3
 126                    .L4:
 127                    .LCFI4:
 128                            .cfi_restore_state

 34:nonquajectdom.c ****     if (v->element_callback(v, n->data)) visit_children
o
o(n, v);
 129                            .loc 1 34 0
 130 0090 8B4304                movl    4(%ebx), %eax
 131 0093 891424                movl    %edx, (%esp)
 132 0096 89442404              movl    %eax, 4(%esp)
 133 009a FF520C                call    *12(%edx)
 134                    .LVL15:
 135 009d 85C0                  testl   %eax, %eax
 136 009f 7422                  je      .L6
 137                    .LVL16:
 138                    .LBB18:
 139                    .LBB19:
 25:nonquajectdom.c ****   for (ii = 0; ii < n->n_children; ii++) {
 140                            .loc 1 25 0
 141 00a1 8B5308                movl    8(%ebx), %edx
 142 00a4 85D2                  testl   %edx, %edx
 143 00a6 7E1B                  jle     .L6
 144 00a8 31FF                  xorl    %edi, %edi
 145                    .LVL17:
 146 00aa 8DB60000              .p2align 4,,7
 146      0000
 147                            .p2align 3
 148                    .L7:
 149                    .LBB20:
 150                    .LBB21:
 151                            .loc 1 49 0
 152 00b0 8B44BB0C              movl    12(%ebx,%edi,4), %eax
 153 00b4 89F2                  movl    %esi, %edx
 154                    .LBE21:
 155                    .LBE20:
 25:nonquajectdom.c ****   for (ii = 0; ii < n->n_children; ii++) {
 156                            .loc 1 25 0
```

```
157 00b6 83C701                 addl    $1, %edi
158                     .LVL18:
159                     .LBB23:
160                     .LBB22:
161                             .loc 1 49 0
162 00b9 E842FFFF               call    _visit_node
162      FF
163                     .LVL19:
164                     .LBE22:
165                     .LBE23:
 25:nonquajectdom.c ****   for (ii = 0; ii < n->n_children; ii++) {
166                             .loc 1 25 0
167 00be 3B7B08                 cmpl    8(%ebx), %edi
168 00c1 7CED                   jl      .L7
169                     .LVL20:
170                     .L6:
171                     .LBE19:
172                     .LBE18:
 35:nonquajectdom.c ****     v->element_end_callback(v, n->data);
173                             .loc 1 35 0
174 00c3 8B4304                 movl    4(%ebx), %eax
175 00c6 893424                 movl    %esi, (%esp)
176 00c9 89442404               movl    %eax, 4(%esp)
177 00cd FF5610                 call    *16(%esi)
 45:nonquajectdom.c **** }
178                             .loc 1 45 0
179 00d0 8B5C2410               movl    16(%esp), %ebx
180                     .LVL21:
181 00d4 8B742414               movl    20(%esp), %esi
182                     .LVL22:
183 00d8 8B7C2418               movl    24(%esp), %edi
184 00dc 83C41C                 addl    $28, %esp
185                             .cfi_restore 3
186                             .cfi_restore 6
187                             .cfi_restore 7
188                     .LCFI5:
189                             .cfi_def_cfa_offset 4
190 00df C3                     ret
191                             .cfi_endproc
192                     .LFE1:
194                             .p2align 4,,15
195                             .globl  visit_node
197                     visit_node:
198                     .LFB2:
 48:nonquajectdom.c **** {
199                             .loc 1 48 0
200                             .cfi_startproc
201                     .LVL23:
202                             .loc 1 49 0
203 00e0 8B542408               movl    8(%esp), %edx
204 00e4 8B442404               movl    4(%esp), %eax
205 00e8 E913FFFF               jmp     _visit_node
205      FF
206                             .cfi_endproc
207                     .LFE2:
```

```
 209                    .Letext0:
DEFINED SYMBOLS
                           *ABS*:0000000000000000 nonquajectdom.c
     /tmp/cc4F3UeL.s:6     .text:0000000000000000 _visit_node
     /tmp/cc4F3UeL.s:197   .text:00000000000000e0 visit_node

NO UNDEFINED SYMBOLS
```

The above is really hard for me to read, so here's the disassembly from `objdump -d`:

```
/home/default/devel/aspmisc/nonquajectdom.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_visit_node>:
   0:   83 ec 1c                sub    $0x1c,%esp
   3:   89 5c 24 10             mov    %ebx,0x10(%esp)
   7:   89 c3                   mov    %eax,%ebx
   9:   8b 00                   mov    (%eax),%eax
   b:   89 74 24 14             mov    %esi,0x14(%esp)
   f:   89 d6                   mov    %edx,%esi
  11:   89 7c 24 18             mov    %edi,0x18(%esp)
  15:   83 f8 01                cmp    $0x1,%eax
  18:   74 76                   je     90 <_visit_node+0x90>
  1a:   72 24                   jb     40 <_visit_node+0x40>
  1c:   83 f8 02                cmp    $0x2,%eax
  1f:   75 0d                   jne    2e <_visit_node+0x2e>
  21:   8b 43 04                mov    0x4(%ebx),%eax
  24:   89 14 24                mov    %edx,(%esp)
  27:   89 44 24 04             mov    %eax,0x4(%esp)
  2b:   ff 52 14                call   *0x14(%edx)
  2e:   8b 5c 24 10             mov    0x10(%esp),%ebx
  32:   8b 74 24 14             mov    0x14(%esp),%esi
  36:   8b 7c 24 18             mov    0x18(%esp),%edi
  3a:   83 c4 1c                add    $0x1c,%esp
  3d:   c3                      ret
  3e:   66 90                   xchg   %ax,%ax
  40:   8b 43 04                mov    0x4(%ebx),%eax
  43:   89 14 24                mov    %edx,(%esp)
  46:   89 44 24 04             mov    %eax,0x4(%esp)
  4a:   ff 52 04                call   *0x4(%edx)
  4d:   85 c0                   test   %eax,%eax
  4f:   74 22                   je     73 <_visit_node+0x73>
  51:   8b 43 08                mov    0x8(%ebx),%eax
  54:   85 c0                   test   %eax,%eax
  56:   7e 1b                   jle    73 <_visit_node+0x73>
  58:   31 ff                   xor    %edi,%edi
  5a:   8d b6 00 00 00 00       lea    0x0(%esi),%esi
  60:   8b 44 bb 0c             mov    0xc(%ebx,%edi,4),%eax
  64:   89 f2                   mov    %esi,%edx
  66:   83 c7 01                add    $0x1,%edi
  69:   e8 92 ff ff ff          call   0 <_visit_node>
  6e:   3b 7b 08                cmp    0x8(%ebx),%edi
  71:   7c ed                   jl     60 <_visit_node+0x60>
```

```
  73:   8b 43 04                mov    0x4(%ebx),%eax
  76:   89 34 24                mov    %esi,(%esp)
  79:   89 44 24 04             mov    %eax,0x4(%esp)
  7d:   ff 56 08                call   *0x8(%esi)
  80:   8b 5c 24 10             mov    0x10(%esp),%ebx
  84:   8b 74 24 14             mov    0x14(%esp),%esi
  88:   8b 7c 24 18             mov    0x18(%esp),%edi
  8c:   83 c4 1c                add    $0x1c,%esp
  8f:   c3                      ret
  90:   8b 43 04                mov    0x4(%ebx),%eax
  93:   89 14 24                mov    %edx,(%esp)
  96:   89 44 24 04             mov    %eax,0x4(%esp)
  9a:   ff 52 0c                call   *0xc(%edx)
  9d:   85 c0                   test   %eax,%eax
  9f:   74 22                   je     c3 <_visit_node+0xc3>
  a1:   8b 53 08                mov    0x8(%ebx),%edx
  a4:   85 d2                   test   %edx,%edx
  a6:   7e 1b                   jle    c3 <_visit_node+0xc3>
  a8:   31 ff                   xor    %edi,%edi
  aa:   8d b6 00 00 00 00       lea    0x0(%esi),%esi
  b0:   8b 44 bb 0c             mov    0xc(%ebx,%edi,4),%eax
  b4:   89 f2                   mov    %esi,%edx
  b6:   83 c7 01                add    $0x1,%edi
  b9:   e8 42 ff ff ff          call   0 <_visit_node>
  be:   3b 7b 08                cmp    0x8(%ebx),%edi
  c1:   7c ed                   jl     b0 <_visit_node+0xb0>
  c3:   8b 43 04                mov    0x4(%ebx),%eax
  c6:   89 34 24                mov    %esi,(%esp)
  c9:   89 44 24 04             mov    %eax,0x4(%esp)
  cd:   ff 56 10                call   *0x10(%esi)
  d0:   8b 5c 24 10             mov    0x10(%esp),%ebx
  d4:   8b 74 24 14             mov    0x14(%esp),%esi
  d8:   8b 7c 24 18             mov    0x18(%esp),%edi
  dc:   83 c4 1c                add    $0x1c,%esp
  df:   c3                      ret

000000e0 <visit_node>:
  e0:   8b 54 24 08             mov    0x8(%esp),%edx
  e4:   8b 44 24 04             mov    0x4(%esp),%eax
  e8:   e9 13 ff ff ff          jmp    0 <_visit_node>
```

Let's slice that down to just the `element_node` case, which the listing above shows us begins at 0x90.

```
00000000 <_visit_node>:
   0:   83 ec 1c                sub    $0x1c,%esp
   3:   89 5c 24 10             mov    %ebx,0x10(%esp)
   7:   89 c3                   mov    %eax,%ebx
   9:   8b 00                   mov    (%eax),%eax
   b:   89 74 24 14             mov    %esi,0x14(%esp)
   f:   89 d6                   mov    %edx,%esi
  11:   89 7c 24 18             mov    %edi,0x18(%esp)
  15:   83 f8 01                cmp    $0x1,%eax
  18:   74 76                   je     90 <_visit_node+0x90>
  ...
```

```
  90:   8b 43 04                mov    0x4(%ebx),%eax
  93:   89 14 24                mov    %edx,(%esp)
  96:   89 44 24 04             mov    %eax,0x4(%esp)
  9a:   ff 52 0c                call   *0xc(%edx)
  9d:   85 c0                   test   %eax,%eax
  9f:   74 22                   je     c3 <_visit_node+0xc3>

  a1:   8b 53 08                mov    0x8(%ebx),%edx
  a4:   85 d2                   test   %edx,%edx
  a6:   7e 1b                   jle    c3 <_visit_node+0xc3>
  a8:   31 ff                   xor    %edi,%edi
  (confusing multibyte nop padding removed)

  b0:   8b 44 bb 0c             mov    0xc(%ebx,%edi,4),%eax
  b4:   89 f2                   mov    %esi,%edx
  b6:   83 c7 01                add    $0x1,%edi
  b9:   e8 42 ff ff ff          call   0 <_visit_node>
  be:   3b 7b 08                cmp    0x8(%ebx),%edi
  c1:   7c ed                   jl     b0 <_visit_node+0xb0>

  c3:   8b 43 04                mov    0x4(%ebx),%eax
  c6:   89 34 24                mov    %esi,(%esp)
  c9:   89 44 24 04             mov    %eax,0x4(%esp)
  cd:   ff 56 10                call   *0x10(%esi)

  d0:   8b 5c 24 10             mov    0x10(%esp),%ebx
  d4:   8b 74 24 14             mov    0x14(%esp),%esi
  d8:   8b 7c 24 18             mov    0x18(%esp),%edi
  dc:   83 c4 1c                add    $0x1c,%esp
  df:   c3                      ret

000000e0 <visit_node>:
  e0:   8b 54 24 08             mov    0x8(%esp),%edx
  e4:   8b 44 24 04             mov    0x4(%esp),%eax
  e8:   e9 13 ff ff ff          jmp    0 <_visit_node>
```

I separated the recursive function from the entry point so that by making it `static`, GCC could pick a more reasonable calling convention for it; which worked, but only up to a point.

Here our per-node overhead is 7 instructions of preamble, 2 instructions of dispatch (by chance, GCC happened to make dispatch fastest for this case), 3 instructions to call `element_callback` (using the three-byte `call` encoding optimized for function pointer tables), 2 instructions to conditionally skip the child nodes, 4 instructions of loop setup, 6 instructions per loop iteration (which we can count toward the child node's cost), 4 instructions to call `element_end_callback`, and 5 instructions of cleanup, for a total of (+ 7 2 3 2 4 6 4 5) = 33 instructions, of which four are conditional jumps, providing opportunities for branch misprediction. (However, the loop is simple enough that I imagine branch misprediction will be very rare.)

The listing of the C code points out that if the visitor is not a quaject, just a regular struct with function pointers, then we can eliminate two `add` instructions, which probably don't matter, and 4 of the 37 bytes, which probably do, and also build visitors in plain C instead of assembly, at the cost of an extra indirection. That is:

```
 0:   55                    push   %ebp
 1:   bd 80 d0 38 fe        mov    $0xfe38d080,%ebp
 6:   ff 53 04              call   *0x4(%ebx)
 9:   73 0f                 jae    0x1a
 b:   e8 fc 0f 38 fe        call   0xfe38100c
10:   e8 40 d8 38 fe        call   0xfe38d855
15:   e8 a6 1d 39 fe        call   0xfe391dc0
1a:   ff 53 08              call   *0x8(%ebx)
1d:   5d                    pop    %ebp
1e:   c3                    ret
```

Now our executable DOM node is 10 instructions and 31 bytes, only 7 bytes more than the 24 bytes the struct needs. Of those 7 bytes, 3 should properly be charged to the child nodes, so the overhead is more like 5 bytes and 8 instructions per node.

http://localhost:8000/wikipedia_en_all_nopic_01_2012/A/Dependoncy%20theory.html has 512 elements, 681 text nodes, and 17 other nodes. Traversing the whole thing should cost some (* 8 (+ 512 681 17)) = 9680 instructions, or about ten microseconds. The document text is 48962 bytes. The tree structure in this form should cost (* 21 (+ 512 681 17)) = 25410 bytes, plus the cost of the metadata (element names, string sizes and lengths, attributes). The entire document tree, then, nearly fits in the 32KiB L1 cache of my netbook, (although I think that's split, so only 16KiB is instructions), and very easily indeed in its 512KiB L2 cache.

The C code might be more efficient if its tree were binary.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- C (p. 3359) (28 notes)
- Assembly language (p. 3328) (25 notes)
- Code generation (p. 3381) (2 notes)

Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB

Kragen Javier Sitaker, 2013-05-17 (9 minutes)

I wrote an Arduino program to generate musical scores with the following strategy: first, generate M 4-beat measures, each beat containing a note or not; then generate M 4-measure sequences from those measures; then generate M 4-sequence supersequences of 16 measures each; then generate M 4-supersequence hypersequences of 64 measures each; then use one or more of these 256-beat hypersequences as your score.

This worked reasonably well, fit the whole score into 128 bytes (I used one byte per sequence element and M=8, so each of the four levels was 8*4 = 32 bytes), and allowed me to encode the iteration state into a single byte. If I'd pressed harder, I could have used three bits per sequence element, for 4×8×4×3 = 48 bytes, except that I actually was using four bits per note.

It occurred to me that perhaps you could use this approach for other things, like general-purpose data compression or image compression, which I haven't tried yet, or building a pseudo-character-cell terminal with very low RAM requirements.

A typical character-cell terminal, like a VT100 or H-19, had about 25×80 character cells, each with about 5×8 pixels; and either one or two bytes of RAM per character cell, plus a ROM font containing something like 5×8×128 bits, say, 640 bytes. Each scanline could be built on the fly out of bits read out of the ROM, indexed by some simple arithmetic, so you only needed some 2000 bytes of RAM instead of the 80 000 (400×200) you'd need for a full screen image.

Suppose, instead, you used this quadtree scheme, but with a slightly unusual aspect ratio: 5×8-pixel characters, 16 lines, 128 columns, for 2048 characters on screen. Each character position would have a byte, as before, but instead of having a byte for each possible character position, you'd have M groups of four bytes for distinct 2×2 squares, where M was something less than 512. You'd need four levels of these, with each block at each level corresponding to respectively 4, 16, 64, and 256 character positions, plus 8 bytes at the top level for the 8 horizontal 16×16 chunks of the screen.

What would M be? You'd probably want it to be 256 or less in order to be able to fit the pointers to the next level into a byte; suppose it were 128. Then each level of the hierarchy would be 512 bytes (128 blocks of 4 bytes) and the total would be 2048+8 = 2056 bytes to hold the screen contents.

So far this sounds stupid, because you've "compressed" 2048 bytes into 2056 while losing the ability to have an arbitrary 2048 bytes on the screen! But there are some interesting possibilities here:

• You don't really need M=128 for the top. The top level of quadtree can only have a max of 8 of its nodes displayed at once. This approach means you could have M=8 and 32 for the top couple of

levels, cutting the memory usage almost in half, to (+ 8 (* 8 4) (* 32 4) 512 512) = 1192 bytes. You can use some of those bytes to give you "spare" blocks at the upper levels, to expand M further beyond 128 for the bottom level (so you can have, say, 1024 bytes of incompressible text at the bottom level).
• Scrolling horizontally by a multiple of 16 columns just involves changing the 8 data pointers at the top level, and possibly constructing a new "blank node" at each lower level.
• Scrolling by 8 lines, half a screenful, just involves modifying the 32 bytes that control the contents of the top-level quadtree nodes.
• You can use the saved RAM for a soft font, which gives you the ability to draw arbitrary graphics! As long as they're repetitive and compressible.
• If your terminal is programmable, you could use video RAM you're not using at the moment (because nothing points to it at the moment) for buffers and code.

It would probably make more sense to make your nodes horizontal instead of square. This means adjacent lines of text no longer interfere with each other's compressibility. Then you have:

• level T0, 256 nodes of four consecutive characters per node;
• level T1, 128 nodes of 16 consecutive characters per node;
• level T2, 32 nodes of 64 consecutive characters per node;
• level L1, eight nodes of two lines of 128 characters each;
• level L2, two nodes of eight lines;
• level L3, two bytes pointing to L2 nodes.

Total is (+ 2 (* 2 4) (* 8 4) (* 32 4) (* 128 4) (* 256 4)) = 1706 bytes, saving about 300 bytes over the traditional representation, at the cost of cutting the possible incompressible text on the screen in half.

Suppose that, instead of using the 5×8 characters that were common at the time, you took the quadtree approach all the way down to the pixels, using 8×16-pixel characters, just with a higher clock rate. Then you'd have, say, three more levels:

• Level P0: 4 8-pixel scanlines of a character, stacked vertically.
• Level P1: 16 scanlines of a character vertically, i.e. an entire cell.

A typical font ROM would likely be highly compressible with this approach, since lots of characters contain blank lines (not containing ascenders or descenders, say) and other repeated features. Suppose we have 96 ROM font characters, and each P0 node is used an average of twice. Then we have 96*8 bytes for the P0 nodes and 96*4 bytes for the P2 nodes. But that's assuming significant horizontal slices of characters are shared. That puts us at a total of 1152 bytes of ROM.

Anyway, what I was thinking was that if you had another 32 nodes in P2 that were in RAM (128 bytes), you could use that for a graphical character set, especially if you have another 128 bytes or so of P0 to play with.

But what about the latency of all these levels of indirection? We're talking about a total of eight indirections to get down to the pixels, now, right? Wouldn't that cut way down on your effective dot clock?

I think the answer is "no", because although there are lots of indirections, they're very *predictable* indirections, in lots of different

memories, so you can pipeline them. A four-byte FIFO at most levels would suffice: P0 needs only two 8-bit output registers, one of which is shifted to the electron gun, and a new output byte is latched into the "next byte" register; P1 similarly needs only a one-byte buffer, which feeds the next address to P0 for when it's ready; T0 fills up a four-byte FIFO going to P1 every 16 pixels of the scan line; T1 fills up a four-byte FIFO going to T0 every 64 pixels; T2 fills up a four-byte FIFO going to T1 every 256 pixels; L1 fills up a four-byte memory that T2 reads 32 times every 32 scanlines, or 32768 pixels; L2 fills up a four-byte FIFO going to L1 every eight lines, or 128 scanlines, or 131072 pixels; and L3 feeds L2 a byte every eight lines, or twice a screen. Each of the next-pointer reads can be loaded into a double-buffered register as soon as the previous value is used, so it's not in the latency critical path. P0, P1, and T0 each need to read a random byte every 8 dot clock cycles; T2 every 64; L1 every 65536; etc. So anything slightly over a single byte read per 2 dot clock cycles should be sufficient. Since we're talking about a (* 60 8 16 128 16) = 15 728 640 Hz dot clock, that would probably have involved separate hardware memories for some of these in the 1970s time period we're talking about, just to keep the RAM reads down to a manageable level, below 8 megabytes per second per channel.

However, I think this approach should make it possible to do a megabit windowing system on a terminal with on the order of 6kiB of RAM, rather than the usual 128kiB.

What about using this approach to generate random images?

Topics

Similarities between Golang and Rust

Kragen Javier Sitaker, 2017-01-11 (updated 2017-01-17) (7 minutes)

Golang and Rust have a lot of similarities, some stemming from their birth around the same time in groups of programmers with similar backgrounds and overlapping goals, and others due to direct borrowing.

Both use the C memory model of nested records and arrays with pointers, but add automatic memory management — a garbage collector in Golang's case, an innovative affine typing system in Rust's case (although Rust was born with a GC, it became garbage along the way and has been collected). This is a huge and underappreciated difference from mainstream garbage-collected languages like JS, Java, and Python, in which memory is just an object graph.

Both have interface inheritance but not implementation inheritance (and both use the keyword `struct` where C++ would use `class`). In both, an existing type can be retroactively caused to implement a newly invented interface (called a "trait" in Rust), but only in Rust does this permit you to add new methods to an existing type.

Both languages have untyped constants, where a number like `0` or `3.14` in the source code doesn't have a fully determined type — the precision of its type is determined by its context of use.

Both languages have limited type inference, eliminating nearly all type declarations for local variables but requiring nearly full type declarations for top-level module items. In both cases, the designers made the misguided choice to cede the `=` operator to mutation, while using a somewhat more verbose locution for the more common operation of declaring and initializing a variable. Rust's type system supports full parametric polymorphism, while Golang's type system, like C's, has some parametrically polymorphic types, but no mechanism to define new ones.

Both languages have very limited use of exceptions in the sense of nonlocal transfers of control (which both call "panics"), instead handling normal errors using return values somewhat richer than are practical in C and similar languages. In both languages, unused-value messages from the compiler prevent you from accidentally suppressing an error signaled by this mechanism. In both languages, the mechanism to explicitly ignore such an error is to assign it to a pseudovariable called `_`, a language feature I think both copied from ML.

In both languages, there is a dichotomy between "arrays" and "slices", which last are pointer-plus-length references to part of an array, while array sizes are statically known. Array indexing is checked at runtime in both languages.

Strings in both languages are sequences of implicitly UTF-8 bytes, rather than sequences of UTF-16 encoding units (as in Java and JS), of UCS-4 codepoints (as in Python), or bytes that each implicitly encode a character (as in C).

Both languages propose nonmainstream concurrency mechanisms in order to escape the disadvantages of pure event-loop languages like

ordinary JS and implicit-shared-everything languages like Java (and C or C++ with pthreads or Win32). Rust's novel type system statically prevents data races between threads; Golang has lightweight threads ("goroutines") and supports shared mutable state with the same concurrent-access bugs that Java does (and, at least originally, didn't even try to prevent segfaults from concurrent access to the same map), but provides special syntax for interthread communication over type-safe channels to minimize the use of shared mutable state.

Both languages add a simpler syntax for iterating over a range. In both languages, the range representation is half-open, including its left bound and excluding its right bound. Both of these are so universal nowadays that they hardly seem worth mentioning, but e.g. C, Java, and JS lack range syntax, and R, Matlab, and Fortran fail to use the half-open representation.

Both languages have acclaimed library package management systems (Cargo and `go get`) with integrated network access. Rust leans more heavily on its package manager in the sense that its standard library includes almost nothing — its `std::net` module, for example, implements no protocols higher-level than TCP, and it apparently offers no way to format a timestamp, parse a CSV file, or encrypt a string, although it does have things like binary heaps, hash tables, pathname handling, and some limited Unicode encoding. By contrast, Golang's standard library includes AES, ELF, DER, CRC, JPEG, suffix arrays, RFC-2822, SMTP, URLs, and JSON-RPC. In Rust these are available as external crates, except apparently there's no SMTP server; crates.io lists 7542 crates.

Both languages have syntactic support for user-defined external iterators, which sounds like a minor detail but turned out to be absolutely transformative to Python when Ka-Ping Yee introduced it.

The biggest difference seems to me that Rust focuses on static safety, even at the cost of programmer convenience, in a way that Golang does not. So, while in Golang it's common to define functions or data structures that pass around `interface{}` references — which can be converted to interfaces of the proper type with a run-time-checked type conversion — in Rust such practices are virtually nonexistent, instead using its parametric polymorphism facilities. In Golang all references are implicitly initialized to `nil` (as in Java, but much less problematically due to the difference in memory models), while in Rust null references are statically impossible, being supplanted by the `Option` type and pattern matching, which doesn't exist in Golang.

I'm tempted to make the analogy to the Pascal/C dichotomy of the 1970s: Rust, like Pascal, prioritizes static correctness and mathematical purity, while Golang, like C, prioritizes programmer convenience, practicality, and compatibility from one month to the next.

I think things might turn out differently this time around, though. First, probably no programming language will ever again be as dominant as C was during the 1980s; neither Rust nor Golang is likely to die within the next century. Second, static verification has become dramatically more effective over the last 40 years, which means that you can get a lot more safety with a lot less hassle. Third, we can afford a lot more CPU cycles and RAM for running compilers

nowadays, even if running the C++ preprocessor over hundreds of gigabytes of repetitively `#included` header files is not a particularly productive use of those CPU cycles. Fourth, cooperation with many other programmers is more important now than in the 1970s, and static safety helps a lot with that.

Topics

Fabric optimization

Kragen Javier Sitaker, 2019-10-28 (updated 2019-10-29) (17 minutes)

I was thinking about generating shapes via laser-cut geometry, and it occurred to me that laser-cut fabric can inexpensively form netting with quite flexible 3-D geometry, which can be applied to a variety of important applications.

Background: auxetic materials and conformal surface mapping

I just watched the presentations of "Beyond Developable" from SIGGRAPH 2016 and "Rapid Deployment of Curved Surfaces via Programmable Auxetics" from SIGGRAPH 2018, collectively by Mina Konaković-Luković, Julian Panetta, Keenan Crane, Mark Pauly, Sofien Bouaziz, Bailin Deng, and Daniel Piker. (I think this is mostly Mark Pauly's group at EPFL LGG.)

In the 2016 paper, they cut relatively rigid sheets of material into an auxetic triangular metamaterial pattern which can linearly stretch by up to a factor of 2, but more or less uniformly — that is, if a region of it is stretched by a factor of 1.3 along the x-axis, it's also stretched by about 1.3 along the y-axis, along the $x = y$ line, along the $x = -y$ line, and so on. The stretch can vary from one region of the material to another, but only gradually. These work out to mean that it expands *conformally*, which turns out to mean that it can be relatively easily hand-bent into computer-designed shapes or adapt to the form of the wearer's body.

In the 2018 paper, they extend this work in a couple of ways: cutting some of the material "pre-expanded", distorting the triangular mesh to vary the properties of the metamaterial, and sometimes replacing the cut sheet of material with a bunch of solid triangles linked together, with the sizes varied to limit the expansion. Then they force it to expand in different ways: by inflating it with a polyethylene bag or a balloon or by allowing it to hang freely from a frame.

Laser-cut netting from fabric

Although I haven't tried, converting fabric to netting by laser cutting should be straightforward: you just cut some holes in it. This is probably better to do with acrylic†, cotton, or silk fabric than nylon or polyester, although Engadget says it is also used with nylon and polyester. Cotton cut this way without further treatment will probably fray, but the other materials might form melted edges that impede fraying.

If you draw a Voronoi diagram of the hole centers, each vertex of the Voronoi diagram is in the middle of a piece of cloth which is connected along lines to adjacent vertices; these adjacent vertices are connected by cloth "bridges" under the Voronoi lines. If we expand the holes without allowing them to cross these lines, the bridges become thinner, but the overall surface retains the same Gaussian curvature as the original cloth, which had better be zero if we are going to be doing this on a garden-variety laser cutter.

If we want to allow a region of the netting to develop positive Gaussian curvature, the simplest thing we can do is to make the Voronoi lines of that region follow the bias rather than the threads of the fabric, assuming it's a woven or nonwoven fabric. (Actually, even simpler is to use a knit fabric, which is stretchy enough to curve any which way, but that gives up precisely the control I'm exploring here.)

However, we can do better than that. By making the fabric bridges squiggle, we can make them longer — potentially by orders of magnitude, serpentining almost arbitrarily far across the space from one node to another, filling in most the neighboring holes with their squiggling. The inside radii of these squiggles need to be large enough to prevent stress risers, also known as "rips", from starting; in some cases this will require discarding a teardrop shape from the inside of the curve, but in other cases the outside radius of a neighboring squiggle can fill the space, thus maximizing the use of cloth.

In this way, it should be possible to expand a 100-mm square of fabric (or even less) into a 1-m square (or more) of netting which assumes some complicated three-dimensional shape when fully inflated — either a fully determined form, if the graph of net nodes consists entirely of triangles (Bucky Fuller's "omnitriangulated"), or a more flexible form if some or all of the holes in the net have other shapes. By applying the process to heavy, strong fabrics such as twill, denim, seatbelt webbing, canvas, or burlap, it should be possible to quickly manufacture fairly complex, strong three-dimensional forms.

In some cases, particularly without omnitriangulation, squiggling is not necessary, because the nodes can be mapped into the plane of the original fabric in such a way as to put network-nearby nodes far apart. Consider, for example, a series of concentric rings of cloth linked together by quarter-turn spirals: if hung from the center of the cloth, the spirals untwist and become lines along a cone from its vertex, while the rings run around the cone. Some teardrops will still be needed at junctions to prevent stress risers.

Uniform amounts of squiggling over an area will not produce positive curvature when fully extended, but zero Gaussian curvature; in that case, a local reduction in squiggling will produce a region of *negative* Gaussian curvature, as demonstrated in the 2018 paper mentioned above.

† "Acrylic" fiber is polyacrylonitrile, which is not the same chemical as the poly(methyl methacrylate) "acrylic glass" (Lucite, Plexiglas, or Perspex) which is so popular as a laser-cutting medium. So it might not be as safe or convenient.

Geometry limitations

The shapes you can make from the fully extended omnitriangulated nets will necessarily be pretty convex; they can't have "pockets" in them, in the machining sense of a cavity eaten out of a surface, rather than the sartorial sense. Moreover, the net geometry, even if omnitriangulated, only controls the local curvature of the surface, not its global curvature. Once an omnitriangulated surface curves around to form a closed surface, the global curvature becomes fully determined, but until then it is potentially kind of floppy. For clothing this is usually considered advantageous, but not for some other possible applications.

Long protrusions are possible, but limited — they necessarily imply cloth density limited to their circumference-to-length ratio. That is, if they are twenty times longer than their circumference, they cannot have more than 5% cloth coverage.

Further alternative materials

Although ordinary textiles are probably the best material to use for this for many purposes, other possibilities exist. Nonwoven polyester (polar-fleece and *friselina*) can be dramatically cheaper and may be more computationally predictable, though not as strong or as stiff. Paper or Tyvek are cheaper still, and probably faster to cut. EVA foam ("foam rubber") is laser-cuttable and is widely used for shoes.

I haven't personally seen gel-spun UHMWPE fabric yet, whether woven or knit, but if you can get it woven, it might offer better strength and enormously better dimensional control than the other fabrics mentioned above. I've seen advertisements for pantyhose and bras (Katherine Homuth's SheerlyGenius, now Sheertex) and backpacks (Loctote) made from gel-spun UHMWPE fabric, knit in both cases.

Other sheet-cutting processes, such as high-powered laser cutting, waterjet cutting, and CNC torch cutting, could be used to cut sheet metal into similar netting-like structures, though its behavior under strain would be different; if you do manage to stretch it into shape, it will probably work-harden in the process, especially if you're using annealed copper or aluminum. Your dimensional precision is going to be shit.

Cloth made from carbon fiber, glass fiber, or basalt fiber would also provide higher strength and better dimensional control than more common fibers, but they might not be cuttable with a low-power laser.

Woven aluminum or steel window-screen material is another candidate for a higher-strength, higher-dimensional-precision material (though perhaps inferior to UHMWPE), and it could be cut very rapidly, possibly in several layers at a time, with plasma torches or high-powered lasers.

Coloring the net

The amount of fabric in an area has an upper bound determined by the degree of expansion that area experiences when going from the original flat fabric to the fully extended net, and a lower bound determined by the required strength of the netting. However, these will commonly be quite wide limits. This means that you can vary the net coverage in an area to moderate its color. In particular, white netting over a black background should be able to produce significant changes in luminance, though the maximum contrast will be limited as described.

Moreover, by using three or four layers of netting nested inside one another — red, green, blue, and possibly white — full-color images should be feasible, as long as moiré patterns and coregistration can be kept under adequate control.

Uses of shaped netting

Clothing and hammocks are the most immediately obvious uses. A

laser-cut woven or nonwoven netting layer can provide shape, while a fine knit layer underneath can prevent the netting from being uncomfortable or transparent when those are not desired. Netting shirts are already popular for hot days and for dance clubs.

Aeron-style chairs typically use knit netting for the seats and backs, providing much better ventilation than more traditional styles of chairs; laser-cut netting should be a reasonable alternative. Director's chairs and foldable camping chairs traditionally have canvas seats, but breathable fine netting would be an improvement.

Some other household items can be feasibly made in this way. My shoes hang in the closet in a cloth organizer that I think could be made from netting. A collapsible bucket for carrying water can consist of a hoop, some a plastic-bag liner, and netting to support the bag. Stuffed animals can be made of netting to give shape to fine knit cloth containing the stuffing. Houseplants can be potted in netting pockets suspended from railings. Backpacks, curtains, and clothes hampers can be made from netting.

Breathable shoe uppers can be made from EVA-foam "netting", and other uses might include objects like toolboxes that would benefit from being flexible and lightweight but cannot afford cloth netting's tendency to snag on anything sharp.

Inflatable sculptures, like those made by Ophélie Dorgans, can be easily inflated to multi-meter size, but historically it has been somewhat difficult to control their shape; most bouncy castles and advertising balloon critters are, geometrically speaking, kind of shitty. This approach offers an alternative: inflate a big shapeless plastic bag inside a shaping net. (This is demonstrated in the 2018 paper mentioned above.) As explained above, the shaping net can also provide color.

Such an inflatable sculpture could be inflated with methane, hydrogen, or helium and unleashed over a city, as with the 1970s UFO art prank Theo Jansen participated in and the earlier Los Angeles Meteor prank. (I can't remember if that last one is still anonymous or not.)

Blow molding of plastic bottles and vacuum molding of sheets of PET and other thermoplastics should be feasible using netting forms made in these ways, particularly if the fabric is cotton or nylon. As with clothing, a layer of fine knit fabric between the net and the hot plastic may be a useful mechanism for reducing netting marks on the surface.

With a reasonably flexible and gritproof liner — again, perhaps fine knit cotton — these netting forms could be suspended from frames and used as molds for poured-in plaster, concrete (which additionally demands alkali resistance, which cellulose and polyester have), lime cement, fluidized greensand, waterglass-bonded sand, or perhaps even slip (*barbotina*) for slipcasting.

Many of the above-mentioned castable materials are most interesting as refractory mold materials for metal casting, but of course in metal casting usually you want pocketing, which is, as I said above, impossible — assuming that the molding materials are on the convex side of the netting. An alternative might be to inflate the netting with a plastic bag, then pour plaster (or cement or whatever) around it to make the mold.

By painting, epoxy-impregnating, airplane-doping, sizing,

stuccoing, spraycreting, or applying *barbotina* to an inflated net, perhaps with knit-fabric layers on one side or the other, you should be able to make a thin composite material, reinforced by the fabric layer or layers to reduce its brittleness. Depending on the composition, this composite may be adequately sturdy even for some metal-casting processes.

In some cases, just filling the netting with sand may be sufficient; I suspect this may be one of the lowest-cost ways to get acoustic panels.

Cutting starched cloth board

The only laser-cutting machine I've used can cut 6-mm MDF at about 24 mm/s, but can cut thinner materials faster. Many interesting laser-cuttable fabrics are in the neighborhood of 200 µm thick, suggesting that the laser could cut them at a speed on the order of 1 m/s. However, particularly cutting complex shapes, the machine is mechanically capable only of a dramatically lower speed than that.

As I understand it, typical mass textile fabrication is done by cutting a thick stack of fabric to the pieces of the pattern with a bandsaw knife before sending the pieces out to be pieced together. Hundreds or thousands of layers of cloth are cut at once. (Mistakes here are ruinously expensive, ruining thousands of meters of cloth with a single bad cut.)

Typically the laser-cutting shop is not willing to cut multiple sheets of material at once, perhaps because they blow apart under the air blast, ruining the laser focus. But perhaps we could laminate several layers of cloth together with an adhesive to form a board similar to MDF, cut them, and later remove the adhesive.

One promising adhesive for this purpose is food starch: it is a polysaccharide like cellulose, so it doesn't stink when it burns, and it is commonly used for stiffening common fabrics, even polyester, which is a material notoriously resistant to being stuck to. As "wheat paste", it is also commonly used as an adhesive in bookbinding and papier-mâché. In fabrics that can withstand boiling — basically anything *but* polyester — removing it is also easy.

Papier-mâché practice demonstrates a risk of the cloth-board approach: drying time is necessary between layers in order to avoid trapping excessive moisture within the material. A warm air blast should be able to speed up this process.

This approach should make it possible to cut dozens of layers of cloth at once at MDF-like speeds, thus greatly reducing the cost of complex shapes, if you wanted more than one of them, anyway.

Alternative chemistries

Other candidate dissolvable adhesives for this purpose include sugar, PMMA, polystyrene, carrageenan, gelatin (or hide glue), calcium stearate, albumin, urea, and pectin. All of these could also be used as sizing to paint inflated netting to solidify it, as described earlier.

Salt is probably bad because it would produce HCl, chlorine, and sodium gases; waterglass is probably bad because it wouldn't cut with a low-power laser, and might swell up and block the cut instead; baking soda might work but might produce NaOH when cutting; nitrates are probably bad because they might offgas nitric acid and

nitrate the cellulose when fiercely heated; sulfates probably wouldn't cut (and might offgas acid sulfur oxides and produce toxic sodium sulfide); and soluble phosphates might have similar problems. (Ammonium phosphates in particular decompose to ammonia and molten anhydrous phosphoric acid upon fierce heating.) Some of these, especially waterglass and in-situ-formed phosphates of calcium or copper, might be suitable for painting onto an already-cut inflated netting form.

In other contexts, the possibility of selectively and precisely forming such gases on a surface or within a material with a laser-cutting machine might be very useful. Ben Krasnow on his Applied Science YouTube channel has reviewed a tactile-printing paper whose surface swells up when locally heated, for example.

Topics

Dercuano search

Kragen Javier Sitaker, 2019-05-16 (2 minutes)

I'd like to add full-text search to Dercuano; God knows I grep it often enough. Right now I have 4.5 megs of Markdown in 341 notes, producing a 2.0-megabyte gzipped tar file with static HTML files (plus tables of contents), and I'm planning to expand these numbers by a factor of about 2.5 by the time I'm done: less than 1200 notes amounting to 5 megabytes compressed, 11 megabytes plaintext. The index needed for full-text search with snippets is pretty much equivalent to the corpus of notes.

Grepping the 4.5 megabytes with grep takes 0.6 seconds on my laptop, but only 28 ms of that is user time; a second grep takes 42 ms of which 32 ms is user time. Generating a 200-megachar string in Firefox with `Array(100*1000*1000).join('a')` takes a few seconds and provokes a warning about slow scripts, but then `.indexOf('a,b')` in it takes 1.7 seconds, and `.indexOf('a,a,b')` in it takes 2.2 seconds, and `a,a,a,a,b` takes 2.8, which, interpolated down to 11 megacharacters, would be 150 ms; so even brute-force search, without even building an index, may be a reasonable implementation strategy.

The difficulty is that modern browsers generally don't allow file:// HTML to access other file:// URLs via XMLHttpRequest, although old versions of MSIE and the ActiveX XMLHttpRequest control did. So leaving the notes in static HTML makes full-text search impossible.

A possible solution would be to put the actual note text into .js files and load them with `<script src>`, rather than putting it into HTML. The various HTML documents could load their respective text from their respective .js files, while the search engine page could load all of them and maybe build an in-memory index.

To keep the search-engine page reasonable to load, the number of .js files should be limited. 64 might be a reasonable limit; this would be 4–7 notes per 70-kilobyte .js file at present, increasing to 17–20 notes and 170K each by the time it's done. This doesn't seem like a grossly unreasonable number for a page load from the local filesystem, and indeed it might end up being comparable to the amount of common JS liabilities they load.

Topics

Household thermal stores

Kragen Javier Sitaker, 2018-12-02 (updated 2018-08-19) (27 minutes)

Today is 2018-11-16. My house here in Buenos Aires has another power outage, an event which happens regularly during the summer months. They think power will be restored in some four hours, but this is uncertain. On one occasion a few years ago, I had a three-week power outage.

(Note: in fact, as I wrote this, the power came back and stayed on.)

The first thing I did was to check the refrigerator and put ice into it. So I was thinking about the sizes of the thermal stores that would be needed for the various temperature-control applications of electrical energy that I had been suddenly denied.

Refrigerators

In preparation for such events, I have some soft-drink bottles filled with water and frozen in the freezer, to absorb heat through their enthalpy of fusion, and more soft-drink bottles filled with water and chilled in the refrigerator, to provide thermal mass. When there's a power outage, I move ice bottles from the freezer into the fridge.

Unfortunately, my refrigerator is not a super-efficient model like the Sun Frost line; it's a rather normal refrigerator with a compressor, a magnetic-strip gasket, and a few centimeters of foam between the inside and outside walls. I've scoured it in vain for information about its energy-efficiency, except that it claims its nominal power is 230 watts, and it uses 180 watts to defrost the freezer and 15 watts for the light. Let's assume that 230 watts is the maximum power, but that it doesn't run the compressor and the defroster at the same time, so the compressor uses 215 watts. And, although I haven't measured this, the compressor needs to run somewhere around half the time. (This is normal. If your compressor needs to run 80% of the time or more to keep the fridge cold at normal ambient temperatures, you're in danger of going over the temperature setpoint and spoiling your food if it warms up a bit outside, say from 20° to 24°, and so compressors are sized to prevent this. But if your compressor only needs to run 20% of the time, you could have used a compressor half the size, and the fridge would be a lot cheaper.)

So let's say the compressor averages 110 watts at normal ambient temperatures, and let's suppose it has a coefficient of performance of 2, which is a normal kind of coefficient of performance for refrigeration systems. That means it draws off 200 watts of heat from the food within, on average, to compensate for the heat leakage through the walls and the seals and the losses when the door is opened and all the cold air falls out. (The bottles in the fridge also serve a bit to reduce the air losses.) And that means that those leakages are about 220 watts.

The enthalpy of fusion of ice is about 333 kJ/kg, so 220 W is about 660 mg/s of melting ice, which works out to 57 kg/day (19 MJ/day). I don't have 57 kg of ice in there. I have the following bottles:

| N | ℓ | total |

```
|---+-------+--------|
| 1 |  2.25 |   2.25 |
| 3 |  1.25 |   3.75 |
| 2 |   0.6 |    1.2 |
| 8 |   0.5 |     4. |
| 1 |  .375 |  0.375 |
|   | total | 11.575 |
#+TBLFM: $3=$1*$2::@7$3=vsum(@2$3..@6$3)
```

So I have about 11.6 kg of ice in there, which works out to about 4.9 hours. Whew, that's not much! And I guess that's why old-fashioned iceboxes worked on a basis of daily door-to-door ice delivery.

Wrapping the refrigerator in a blanket might help the situation. But another possible solution would be a coolth reservoir where I actually could store 50 kg or more of ice, frozen when there's power and used for cooling when there's not. It could be relatively small, and indeed to keep the power costs acceptable it would need to be relatively small and relatively well insulated.

Dehydrators

At Burning Man, where you need to either burn or haul away any garbage you generate, down to the smallest fern leaf, we dehydrated our garbage by hanging it out in the dry wind in a net bag; this prevented it from rotting while we were there, improved its qualities as fire fuel, and reduced its weight enormously for the trip back. Though this is rarely done in the US, the same procedure is commonly used to dry clothes after washing them, though usually without the bag. And of course this is a common way to preserve sliced produce as well.

Putting air in contact with moist things causes the humidity in some surface layer of air to reach 100% (in the case of pure water) or whatever the equilibrium relative humidity of the solution is (when it has solutes such as salts). Water can diffuse across this layer into other air, or you can blow the surface layer away and replace it with new air, and in either case you remove water from the vicinity of the moist thing and increase the drying rate.

The amount of moisture required to reach 100% humidity rises exponentially with air temperature, doubling about every 10½°, so by raising the temperature of the air a few degrees, you can get very substantial increases in the drying rate. This is complicated by the fact that the evaporation itself cools down the air and the moist thing, and so your surface temperature is always a bit lower than the incoming air temperature; after a short while, the heat thus absorbed is necessarily replaced by the heat in the incoming air.

The principal difficulty with dehydrators is that they have to work relatively fast to be useful, and this requires the air to be both hot, which requires a lot of energy, and fast-moving, which requires a bit of energy but mostly careful airflow design. There's the additional difficulty that, if you're heating things up with fire or electricity, you need a thermostat, and ideally one that turns the heat off if it fails, rather than unexpectedly setting fire to things and melting things.

The enthalpy of vaporization of water is 2.26 MJ/kg, an enormous number, and it swamps the specific heat of air or vapor, so nearly all

of the energy that goes into the dehydator is used to evaporate water.

Here, I've been using my electric oven to dehydrate garbage. The oven has a thermostat that goes down to 50°, and also a fan and a shutoff timer. So far, the thermostat has been dependable, but I'm not sure that it's designed to fail safe; the shutoff timer fails when there's a power outage (as I said before, a frequent event), and in those cases the oven turns back on when power is restored. This is unacceptable and so I cannot depend on the shutoff timer.

(Ideally a dehydrator would be driven by a humidistat and would shut off when the desired humidity was reached.)

Depending on what you're dehydrating, different temperatures may be acceptable, and clothes dryers are designed with a relatively sensitive thermostat for precisely this reason — the high heat that most quickly dries cotton clothes would be disastrous for delicate synthetics. 35° is a fairly safe temperature for anything other than certain exotic materials you probably shouldn't be playing with anyway. Raw-foodists require their food not to be heated above 42° for somewhat debatable nutritional reasons. 75° is generally considered adequate for sterilizing food; although there are a few exotic thermophile microbes that can survive higher temperatures, they are not active at 37° (the temperature they are at after you eat them) and so they are generally not pathogenic. Some common plastics, especially PET — the polyester used for both cloth and Coke bottles — soften and change shape above 90°, and in my oven I've observed dried egg whites browning somewhere in the 70°–90° range. Water, of course, boils at 100°. Wood starts to brown around that temperature, too, though the autoignition temperature of paper is 233° according to Bradbury.

Using the psychrometic exponential rule of thumb, even at 35°, you evaporate water almost 3× as fast as at 20°, and at 75° you evaporate it 38× as fast. Furthermore, the same air heated to 75° has 38× lower relative humidity, so if the relative humidity is 70% outside, it's 2% in the dehydrator. This means not only that you can dry things faster, but also that you can dry things that wouldn't dry at all in room-temperature air, because they have enough salt or other hygroscopic substance in them that they're already in equilibrium.

Predicting the exact drying rate is very complicated, because it depends not only on the rather complicated diffusion and advection processes I mentioned above, but also on airflow patterns, specimen thickness, humidity diffusion rate through the specimen, salinity, and the presence of other hygroscopic substances. But usually, in my oven at its nominal uncalibrated 75°, my tens-of-millimeters-thick food garbage dries out satisfactorily in an hour or two.

How much energy does this require? 100 g of food garbage (orange peels, chicken bones, outer onion layers, food particles washed off dishes, and whatnot) might be a meal's worth, assuming you don't let food rot. Say that's 75% water, so 75 g of water. That's 170 kJ at 2.26 MJ/kg. Air's fairly low specific heat of 1.01 kJ/kg/K means that putting 170 kJ into heating air from 20° to 75° means you need to heat about 3 kg of air to deliver that energy; at 1.2 kg/m³, that's 2.5 m³ of air you need to pass over the meal's worth of food, or 510 kJ or 7.5 m³ per person per day, 5.9 W. You might need to double that if some food goes bad. It's still a factor of 20 to 40 smaller than the reservoir you need for the refrigerator.

In a situation of a prolonged outage of municipal services, dehydration is probably also a reasonable weapon to have in your arsenal against food decay, corpse decay, and feces, although if you're going to be dehydrating feces with hot air you probably want to use a separate dehydrator from the food dehydrator, and you might also want to use some kind of scrubber on the output air to keep the smell down. (This in turn suggests you might want to use a closed-loop system like the one modern condensing clothes dryers use.) Dehydrated feces can be burned, stored, or more easily composted than fresh feces, especially diarrhea, and if you can sterilize them in the process, all the better.

Cooking

My stove is, unfortunately, electric. Cooking a meal involves, minimally, heating the food to the right temperature and holding it at that temperature until it's cooked. The temperature is invariably between 40° and 100°. Some meats can be cooked *sous vide* at temperatures as low as 60°, at least once they're sterilized, but many vegetables require 90°. Kidney beans in particular are dangerously poisonous if cooked below about 95°.XXX

The traditional means for holding the food at the right temperature is to continuously apply heat to it, while it's losing heat to the environment by another route, so that the thermal equilibrium is around the desired cooking temperature, but other approaches are possible; for example, you can use a thermos or hooikist to maintain the temperature passively, or a thermostat together with some insulation to maintain it more actively, as crockpots, *sous vide* cookers, and my oven do.

Typically a meal's worth of food for a person is about 500 g, and it's largely water, with a specific heat in the neighborhood of water's. Heating it from 20° to 80° thus requires 20 kilocalories, 84 kJ, which is 2 minutes in a 700-watt microwave oven at full power. Thermal losses might push the energy required higher, perhaps to 150 kJ. Three meals a day then require 450 kJ per person per day, or 5.2 W, very close to the 510 kJ required to dehydrate the food garbage.

Indoor climate control

Due to deplorable construction techniques, a refrigerative air conditioner of 2000 watts or more is necessary in many Buenos Aires apartments to maintain the temperature at bearable levels. 2000 watts of input produces about 4000 watts of heat removal (CoP ≈ 2), but typical duty cycles are about ⅓ on a 24-hour basis, meaning that you only need to remove 1300 watts on average (115 MJ/day). Typical set points are in the 18°–26° range; 24° is about the most I can take.

Ice vests

Workers in very hot environments, such as some parts of power plants, often use ice vests, which either have ice packets in the vest or coolant tubes in the vest running to an ice backpack containing some phase-change material at or below 20°, typically water ice at 0°. This allows you to shunt the 100 W or so of heat that your body produces into the ice rather than finding a way to reject that heat into a hostile environment. If you are adequately insulated, that 100 W, or up to

2000 W if you're exercising hard enough, is all the ice vest needs to absorb.

Usually ice vests are used for short stints, like an hour or two, but an ice vest that could last 8–12 hours would be very useful in the Buenos Aires summer. An hour at 100 W is 360 kJ, or a bit over 1 kg of ice, so a 12-hour ice vest would require about 13 kg of ice, which would be an uncomfortably heavy backpack.

As an alternative to indoor climate control, an ice vest with easily interchangeable ice packs, or even that you could leave plugged into a flexible coolant line as you move around your apartment, could be lightweight, and it would require less energy than air conditioning, since it would only need to absorb the heat produced by your body, not the sunlight flooding in through the windows or the hot air seeping in under the door.

Showering

The building has a central electric hot-water system with a huge stainless steel tank, so my hot water didn't actually go out during this power cut, but it would during a longer cut. My shower delivers about 250 mℓ/s of water at typically about 40°, but needs to be able to reach 45°; further temperature range is undesirable, as water above 45° can scald you before you can react. I typically shower for about half an hour a day, which is perhaps longer than most people, and works out to about 450 ℓ of fresh water that would have otherwise flowed unmolested past Buenos Aires into the Atlantic, where it turns brackish in the estuary of the Rio de La Plata.

450 ℓ of water at $\Delta T = 25$ K is 47 MJ per person per day, or about 540 W.

Summary of thermal energy needs

So here are the controlled heat flows per person, with a per-day average. This is not the electrical energy, but the thermal energy.

Use	W (avg)	MJ/day	target	
Air conditioning	1300	115	20˚	cooling
Refrigerator	220	19	4˚	cooling
Ice vests	100	8.6	20˚	cooling
Showering	540	47	45˚	heating
Cooking	5.2	0.45	100˚	heating
Dehydration	5.9	0.51	75˚	heating
total	2171.1	190.56		

#+TBLFM: @8$2..@8$3=vsum(@2..@7)

So the first thing to notice is that air conditioning actually accounts for 60% of the total, and the problem gets substantially easier if we can eliminate it, for example with ice vests or better building envelope design. The second thing to notice is that the high-temperature heat applications are very low-volume, while the high-volume heat application, showering, is very low-temperature.

Possible reservoirs

What combination of thermal reservoir designs could we use to satisfy these needs? We can use thermal mass or phase-change materials, and we can recharge the reservoirs from different sources of

heat and cold.

We can consider the hot reservoirs and cold reservoirs separately, because they are

Cool reservoirs

Although ice vests could in theory use a higher-temperature phase-change material, you pretty much need water ice in even larger volumes for food refrigeration, and probably any other phase-change material would be both more massive and more expensive. So it's probably best to just use ice for the phase-change cool reservoir. 28 MJ of ice is 84 kg of ice per day, of which 8.6 MJ or 26 kg is for your ice vest; a week's store of ice would be 590 kg, almost a cubic meter. This is a substantial-sized home appliance, especially with the extra volume needed for cooling extra people.

The difficulty with water ice is that you need something below 0° to get it, unless perhaps you make clathrates or something at a slightly higher temperature. Although some very dry places do reach such temperatures during the night — and you might be able to do a bit better still with reflective optics and some evaporative cooling — here in Buenos Aires, this pretty much requires active refrigeration, either with a compressor or some other form of refrigeration, like a vortex tube or an ammonia-absorption refrigerator.

(An ammonia-absorption refrigerator powered by solar thermal energy is quite a reasonable possibility.)

But it's quite common for the temperatures to reach cool and even chilly levels during the night; as I write this, for example, it's 18° outside. If you could accumulate a large mass of 18° material during the night, for example by passing night air through it, you could use that for indoor climate control during the day, even if not directly for food refrigeration. But how much mass would you need?

Supposing that it's in the form of water bottles at 18°, which you then allow to warm up to 23° as you pass air over them in order to sink the undesired daytime heat. To compare apples to apples, we'll suppose we're using this to cool your no-longer-ice vest rather than to air-condition your house, so we only need 8.6 MJ. But 8.6 MJ / ((23 - 18) calorie / g) = 410 kg instead of the 26 kg for the corresponding ice. (The other 58 kg were for cooling food down to 4°, for which the 18° water would be counterproductive.) Cooling the 410 kg of water back down at night involves passing at least 8.6 MJ / (5 K · 1.01 kJ/kg/K) = 1700 kg = 1400 m³ of 18° air over them. That's 66 ℓ/s or 139 cfm if you do it in the 6 coolest hours, which is a quite feasible airflow rate, but one that will require forced air in any reasonably-sized system.

If you have a phase-change material that changes phase at a temperature between 18° and 23°, you could use that instead, and maybe you could get it down to 50 kg or something. Unfortunately, the only things I'm aware of that fits the bill is certain expensive grades of paraffin and some outrageously expensive metals.

The potential advantage of using a second cool reservoir at a more moderate temperature is that you can replenish it in relatively simple ways, like with a box fan running at night, rather than a compressor or vortex tube or something. But it's probably not worthwhile, since you need the larger, colder cool reservoir for food preservation anyway.

Hot reservoirs

On the other hand, the system should almost certainly use separate reservoirs for heat: one for the ≈1 MJ daily of cooking and dehydration heat at 100°, and another one for the ≈50 MJ daily of hot water at 45°. Both of these temperatures can be easily obtained from solar thermal energy; the equilibrium radiation temperature of "one sun" is 94°, which is to say that thin things on Earth's surface only fail to reach 94° when they're at right angles to sunlight because they have air blowing around to cool them off, and "two suns" — the intensity of sunlight you get from direct sun plus a flat mirror — gets you to 163°. Anything up to "ten suns", which would get you to 379°, is considered a "low solar concentration ratio" and is easily achieved.

Small high-temperature reservoir

If we have to gather the 11.1 W of cooking and dehydration heat during 6 hours of sun exposure — a reasonably pessimistic figure — then we need 44 W during that time, which is 0.044 m², a 210-mm-square area. In order to be able to cook and dehydrate at other times of day, we need to transfer the heat from the absorber (which is of a size somewhere less than 0.04 m² to achieve the required temperature, and somewhere larger than 0.004 m² so that the optics aren't too complicated) to some kind of reservoir. Bricks, for example, or sand. The easiest way to move this heat around is with some kind of coolant, and the easiest one that has no trouble with temperatures from 20° to a bit over 100° is air.

If we add a bit of pad, let's suppose that the reservoir itself is at 120° when full and stores 3 MJ of heat that are released when it cools down to 100°, at which point we consider it "exhausted". Storing 3 MJ in 20 K of ΔT in a material with, say, 1 J/g/K, means that it needs about 150 kg of active mass, say about 1500 ceramic tiles of 100 g each, which will occupy on the order of 0.15 m³ of volume including air spaces between them. This will have on the order of 1.5 m² of surface area around it through which it can lose heat to the environment with its ΔT of about 90 K above ambient. We'd like it to lose no more than, say, 1 MJ during 18 hours when it's not warming up, which is a rather demanding 15 W or so, 9 W/m², or 0.1 W/m²/K. It's probably not safe to use organic insulators like polyisocyanurate, styrofoam (33 mW/m/K; see Deep freeze (p. 1465)), or straw (90 mW/m/K) at these temperatures, but fiberglass should be fine. I don't have fiberglass's insulance handy, but let's say it's 50 mW/m/K. Then you need 500 mm of fiberglass around your thermal reservoir to reach such a demanding specification. This is somewhat unreasonable.

So let's add a lot more pad. Let's let our reservoir be at 200° when full and store 20 MJ of heat by cooling down to 100°. Now it needs to be 200 kg, which would be a cube 58 cm on a side at 1 g/cc, with a surface area of 2.1 m². Say we can afford to lose 16 MJ (leaving 4 MJ) during 18 hours, which is 250 W (better not keep it indoors in the summer) at a ΔT of 130 K, which works out to 120 W/m² and 0.9 W/m²/K. (50 mW/m/K) / (250 W / 130 K / 2.1 m²) = 55 mm of fiberglass insulation, which is eminently feasible, comparable to what my refrigerator uses.

You can't, of course, support 200 kg of tile or whatever on

580×580 mm of fiberglass and maintain the fiberglass's insulance. But you can suspend the mass in a frame in the middle of the fiberglass, hanging it by metal wires. 1mm-diameter music wire would in theory be enough, but probably more like 8 such wires would be advisable to keep the suspended frame from shifting around and opening gaps in the fiberglass.

If you can use a phase-change material with a transition temperature between 100° and 200°, you might be able to simultaneously reduce the mass, and the volume, the working temperature of the reservoir, the variability of the temperature, and the surface area through which heat is lost. Alkali nitrates and typemetal occur to me as possibilities.

The absorber that heats up the air to heat the reservoir

Peak shaving and TOU metering

Some electrical utilities, like PG&E in California, are introducing time-of-use metering, where they charge you more for electricity used during "peak" hours than during "off-peak" hours. While this is still a long shot from true demand response, it should create a market for appliances such as refrigerators

https://enphase.com/en-us/products-and-services/storage

Topics

Studies support authority

Kragen Javier Sitaker, 2017-04-10 (2 minutes)

If the powers that be are sensible, then studies will show objective evidence that actions they take have positive effects, but not their negative effects, because the negative effects can be designed to not be objectively verifiable for a long time.

As a clear example, official crime statistics showed a drop in the murder rate in the first years of the Argentine dictatorship in the 1970s. It was known that the state was also fighting against "subversion", meaning a couple of armed Marxist insurgencies. But the people who were kidnapped and murdered by the police — some for being Marxist insurgents, others for being merely suspicious, or for having children that could profitably be adopted by military families — did not show up in the official murder statistics. During this time, a friend of mine was arrested, interrogated, and raped by the police. Her rape did not appear in the official statistics.

Some ten thousand "disappearances" and murders were documented, using the official records kept by the military, after the return to democracy. Another twenty thousand people are said to have disappeared during this time, but we do not have good evidence to know how many of them are victims of the dictatorship whose records were destroyed and how many were killed by non-government organized crime, how many abandoned their families and set up new identities elsewhere, and how many died in accidents and whose bodies were never found.

Topics

Bubble display

Kragen Javier Sitaker, 2017-01-24 (updated 2017-08-03) (1 minute)

A bubble in gel scatters light. I think it scatters an amount of light proportional to its cross-sectional area, which is proportional to the square of pressure. If so, the average amount of light scattered from the bubble is a function of the RMS of the pressure, rather than the average pressure; oscillating pressure will increase the total light scattered.

Gel without bubbles can operate as a fiber optic, restricting light to within itself by total internal reflection.

By connecting a piezoelectric hydrophone to a bubbly gel column, you can generate a traveling wave in the column. But if the column is closed, you can generate a *standing* wave, in which bubbles in some parts of the column move back and forth, while bubbles in other parts merely grow and shrink. I think you can in fact generate an arbitrary superposition of standing waves simply by taking the Fourier transform of the desired display scattering function. Then you can illuminate the gel column with total internal reflection.

A somewhat better approach is probably to illuminate the column with a small duty cycle so you don't have to deal with the multiple nodes and antinodes.

Topics

The delta from QEmacs,with only 88 commands, to a usable Emacs, is small

Kragen Javier Sitaker, 2013-05-17 (2 minutes)

I downloaded and built QEmacs. It's a good enough Emacs I could almost use it! Which is inspiring, since it was an individual spare-time project by one person, even if that one person is Fabrice Bellard. So I thought I'd take some notes on its deficiencies to see if it's fixable, or how much more work would be needed to make something really usable.

• Crashes sometimes in help. Open qe_g, C-h b q
• open a C file and type
typedef struct node {
and QEmacs takes you to the beginning of the line. Follow that up with `enum {` and it happens again.
Type `typedef int buf[4];` and it happens again. I think this is just part of a general problem it has with indenting C.
• Yes, it can open a text file of nearly 300 megabytes. But going to the end of the file is slow.
• Doesn't support M-^, M-;.
• C-x C-e (compile) tries to execute the make command in the directory where qemacs started, not where the file is.
• repeated C-k doesn't append to the latest kill-ring item as it should, so you can't use it to cut blocks of text.
• M-q takes you out of your paragraph, so you can't just keep typing.
• Redisplay is visibly slow and not double-buffered.
• Doesn't support ~ in filenames.
• Doesn't support M-/
• Control-backspace is "help" instead of "backward-kill-word".
• When there's pending keyboard input, it wastes time updating the screen with already-stale state!
• Undo works in individual buffer changes, rather than commands, which is quite suboptimal with M-q and also with undoing typing.
• Doesn't support prefix arguments. M-5 M-6 M-g prompts you for a line number.
• M-y (yank-pop) causes subsequent C-y to yank the same thing. Maybe this is an improvement.

Still, it's impressive how much like Emacs it can feel with only 88 commands!

Topics

• Programming (p. 3658) (286 notes)
• Editors (p. 3426) (13 notes)

Interval radiosity

Kragen Javier Sitaker, 2016-07-27 (1 minute)

You can think of radiosity rendering as something like seeking the first eigenvalue of a transformation matrix; each vector element identifies the luminous flux at a particular point in space oriented in a particular direction. The transformation matrix identifies the transformation function of this six-dimensional light field as it propagates through space and bounces off objects, along with whatever sources of illumination are needed. This vector must necessarily be of very high dimensionality in order to provide a reasonable approximation, but it may be possible to do a reasonable job by subdividing space and directionality recursively using interval arithmetic until you reach an adequate approximation.

Topics

A minimal window system

Kragen Javier Sitaker, 2018-04-27 (updated 2018-10-26) (12 minutes)

See also Window systems (p. 1335).

For whatever reason, windowing systems are de rigueur for personal computing systems. What's the smallest one you could build? Computers are fast enough now, since about 2000, to redraw the whole screen every frame, so there's no need to faff about with hacks to avoid redrawing parts of the screen. We just have to keep a cap on how many times per frame we draw each pixel, on average.

A pull shared-memory windowing system

Each application shares a memory segment with the window system. The window system has a list of windows, which it can reorder, each with four numbers: (dx, dy) for the window origin and (sx, sy) for the window width and height. Each frame, the window system composites the application windows together into the framebuffer, using the painter's algorithm. This involves copying rows of pixels from the shared memory segments into the framebuffer, some of which will be overwritten later with other pixels. Each application also has an input queue for keyboard, mouse, and frame events. Keyboard events always go to the topmost window; mouse events go to the topmost window that contains the mouse, except that the mouse-focused window stays fixed while the mouse moves with a button held down.

Frame events indicate the completion of a frame, telling the application that it is now free to scribble over the window buffer used in that frame. Atomic pointer writes allow the application to update its window to a different framebuffer (in the same shared-memory segment) for resizing or double-buffering.

Two possible worthwhile enhancements: support (premultiplied) alpha; don't draw windows that are invisible because they are completely off the screen or completely covered by another window, and (combined with that optimization) divide the screen into 32×32 pixel "subscreens" or tiles that are drawn independently. The first enhancement gives you not only window transparency but also a crude approximation of shaped windows; the second should keep the compositing overdraw to a minimum under most circumstances.

A push tile-stream windowing system

Each application sends the window system a sequence of commands, which can include requests to position or size its window or tiles of pixels to draw in it. The window manager sends back a sequence of events.

If the IPC mechanism supports transferring ownership of blocks of memory (or, sort of equivalently, immutable data) then the tiles need not be copied between memory spaces. If they are, say, 32×32 tiles of 32-bit pixels (4096 bytes each), then a 3840×1024 screen would be 120×32 such tiles, 3840 of them in all. If the drawing command itself is an (x, y, w, h, framenumber) tuple with 16-bit fields, the 3840 tiles work out to 38400 bytes of messages, while the pixel data is a bit over

15 megabytes, 409.6 times larger. Sending 15 megabytes in 3840 write() calls on Linux on my laptop would work out to about 300ns · 3840 = 1152µs for the calls plus 171ps · 15728640 = 2689µs for a total of 3.8 milliseconds — a somewhat excessive amount of time, especially when you consider that the window system needs to read all those bytes, too, taking roughly as long.

A simple test program on Linux is able to create 65000 4096-byte memory mappings for a 4096-byte file in 290ms using mmap(), or about 4.5 µs per mapping. (It fails when it attempts to create more than that.) This means that mmap() is actually a bit slower than copying for such a small mapping, but it doesn't get any slower when the file goes up to 8 megabytes. A different operating system might make somewhat different performance tradeoffs, but there's no strong reason to suspect that Linux's implementation is profoundly suboptimal.

There's the question of whether the tiles should be fixed-size and whether they should be required to be pixel-aligned on a grid. If we take as a priority that the window system should be efficiently nestable, the answers are no — we want intermediate window servers to be able to pass along drawing commands as they arrive, but they may not draw a full tile, and they may be offset so that tiles that are pixel-aligned in the window aren't pixel-aligned on the screen.

This architecture is not nearly as amenable to alpha-blending.

Performance thoughts

Consider the tiled pull design on a 128×32-tile four-mebipixel screen with an average of two drawable windows per tile — some tiles have only one, while other tiles have window overlaps and translucent overlays. It needs to read 8 mebipixels (32 mebibytes) from window buffers and write them into the 4-mebipixel (16 mebibytes) output image, but the tiles are 4096 bytes and thus fit in cache, so this will probably only amount to 16 MiB of write traffic to main memory, for a total of 48 MiB.

My laptop can manage about 2 GB/s of large memcpy traffic on one core and about 4 GB/s across all four cores. At 60fps that's a budget of 67 MB of memcpy traffic per frame — this is cutting it pretty close, because it means 75% of the CPU's RAM bandwidth would be devoted to just filling the screen. It also includes an "Intel HD graphics" engine with 16 execution units, which you could imagine might be capable of blitting quite a bit faster. Wikipedia confirms that the GPU has 25.6 GB/s of memory bandwidth.

This approach requires only some 8192 blit commands per frame rendered, leaving about 2 µs to process each one. Even CPython function calls would be fast enough, as they take only about 200 ns on one core. However, if the commands a CPython program was sending were one level lower — individual scan line segments — there would be roughly 32 times as many comands, and CPython performance would be inadequate.

As it happens, Numpy is capable of doing this kind of thing, because it has mutable multidimensional arrays. A quick test finds that copying 128 MB of RAM via numpy takes 64 ms, which works out to 2 gigabytes per second. However, doing a million small blit operations in this way took 5.4 µs per blit, which is too slow by more than a factor of 2.

```
def draw(n):
 for i in range(n):
  j = (i % 200) * 5
  fb[j:j+5, 0:8] = fb[:5, :8]
```

Factoring out the source image, as if that were realistic, got it down to 3.9–4.0 µs.

```
def draw(n):
    src = fb[:5, :8]
    for i in range(n):
        j = (i % 200) * 5
        fb[j:j+5, 0:8] = src
```

So I could maybe make it work, barely, by taking advantage of multicore, or with a smaller screen, like 1Mpix, or like 30fps. Or by giving up on the small-subscreen approach and doing more precise occlusion calculations to do bigger blits and eliminate overdraw. (My laptop is only 1920×1080.)

If we have 3ms to draw the screen with an overdraw factor of, say, 2 due to alpha, then our 2 GB/s gives us 6 MB.

Vectorizing precise rectangular occlusion calculations

For more precise rectangular occlusion calculations, we could imagine a scan line as a sequence of (pixelcount, sourcewindow) pairs, and a screen as a sequence of (linecount, scanline) pairs. To compute a scan line, we can sort the window vertical edges (dx and dx+sx) and walk from left to right across the scan line, maintaining a Z-ordered heap of windows under the current pixel position, yielding spans when the topmost window changes. A similar calculation for window horizontal edges yields spans of identical scan lines, although it may not be a priori clear which scan lines are going to be identical; it might be better to use an identical-span-merging procedure on the sequence of generated spans, both horizontally and vertically:

```
def coalesce(spans, reducer=operator.add):
    c = None    # Current span
    for k, v in spans:
        if c is None:
            c = k, v
            continue

        pk, pv = c
        if k == pk:
            c = k, reducer(v, pv)
        else:
            yield c
            c = k, v

    if c is not None:
        yield c
```

This constant-space algorithm is a generalization of run-length

encoding, `uniq`, and `uniq -c`, and is actually the reduce phase of map-reduce, assuming the sorting in between the map and reduce is already taken care of. (God damn it, I'm going to end up programming in Rust after all, aren't I?) For simple data, like `list(zip(floor(arange(128)/10), ones(128)))`, it takes about 5µs + 750 ns/item.

To make this work with some windows having alpha, for each span, we need to store either the whole window stack or enough of the layers on top to reach opacity, rather than just the topmost window. If there is no alpha and everything is opaque, it's overdraw-free.

The whole computation doesn't quite fit into the map-reduce mold because, although the windows map to pairs of edges, there's a computation in between to find the differences of the sorted list.

There's a way to express this algorithm, for the addition case, in three lines of Numpy, but I hesitate to describe it as "straightforward" because I keep putting bugs in it when I implement it:

```
def coalesce(ks, vs):
    last = concatenate((ks[1:] != ks[:-1], [True]))
    v = cumsum(vs).compress(last)
    return ks.compress(last), concatenate(([v[0]], diff(v)))
```

In APL, given the inputs in k and v, I think that would be something like this:

```
(L/k) ,[1] w[0],(1↓w)-¯1↓w←(L←((1↓k)≠¯1↓k),1)/+\v
```

except that that doesn't handle cases where k and v have different data types, and also I haven't tested it because LinuxMint doesn't come with APL.

For largish arrays (1e8 trivial items) on my laptop this takes 11 ns per item, about a 20th of a 200-ns Python function call; for a 256-item array, it takes 75 µs (290 ns/item); for a 512-item array, it takes 80 µs (160 ns/item, or about 20 ns per additional item), while for a 65536-item array, it takes 805 µs (12 ns/item). This suggests some slight nonlinearity and a rather hefty Numpy overhead of about 70 µs (about 350 Python function calls) per call. This makes it faster than the longer generator version above for more than about 90 items, but potentially an order of magnitude slower for small cases.

A more assembly-style imperative Numpy implementation might do a better job by reducing memory allocations. For example:

```
def coalesm(ks, vs):
    last = ones(len(ks), dtype=dtype('bool'))
    not_equal(ks[1:], ks[:-1], last[:-1])
    v = cumsum(vs).compress(last)
    result_v = zeros(len(v), dtype=v.dtype)
    result_v[0] = v[0]
    result_v[1:] = v[1:]
    result_v[1:] -= v[:-1]
    return ks.compress(last), result_v
```

This is somewhat faster on large data sets — the earlier version takes

about 19 ns per item to count the duplicates in `floor(arange(1e8)**.1)`, while this version takes about 11 ns. Both take 11 ns per item for trivial int arrays. It is, if anything, about a microsecond *slower* for small problems, as you would probably expect.

Topics

We should use end-to-end optimization algorithms for 3-D printing design

Kragen Javier Sitaker, 2015-09-03 (14 minutes)

I just spent a few hours designing and printing things in PLA with a US$400 Prusa Mendel RepRap, using a workflow that goes like this: OpenSCAD → Slic3r → Pronterface → RepRap → break them. I think this is the wrong approach, for a couple of reasons.

Printing problems I've seen

The RepRap does not realize your 3-D model with perfect precision or reliability. Here are the ways I've seen it fail:

• Printing on dirty glass, or glass that's not hot enough, the bottom layer can peel up from the glass when pulled by tension from the extruder. This may be corrected by subsequent layers, or it may generate a runaway positive feedback loop as the extruder crashes into the debris, peeling more and more of it.
• Of course anything needs a sufficient area of contact with the printing bed to anchor it during printing.
• When the bed is insufficiently well leveled, printing the first layer in some places will not deposit any plastic and may cause the pinch wheels to lose their grip on the filament. This could be remedied in software by using a thicker bottom layer in models with a large X and Y extent.
• Sometimes, particularly with a horizontal overhang, Slic3r tries to delineate a layer boundary by depositing filament on top of empty space. This works up to a point, particularly if it starts on top of non-empty space and then goes out a very short distance onto empty space.
• A tall, skinny model can start to flex with the extruder as the extruder drags across the top of it; this results in a sort of backlash. This is relatively innocuous when you're depositing 10% infill, but when you're depositing 100% infill, it can result in the plastic mounding up, which creates positive feedback — the mound of plastic obstructs the extruder further, which results in further bending and sometimes even getting unstuck from the bed.
• Any kind of obstruction can result in lost steps in X and Y, since the RepRap runs entirely open-loop.
• Depositing a horizontal surface on top of an open space (either one in the model or one generated from infill) always sags somewhat; the extent of the sag, and whether further layers on top of it are able to become fully solid, both vary somewhat. It's easy to end up with just a set of parallel threads that don't cohere into a surface.
• Sometimes narrow vertical walls encounter the same fate: just a set of parallel, unconnected threads. I think that when I saw this, it was from an infill setting that was just on the edge of instability being driven over the edge by the cooling blower.
• I saw one design that drove the X drive belt into one of the support

beams when it drove the Z-axis up past 110 mm. We aborted the print before it could break the printer. Apparently the electronics and Pronterface don't enforce a sufficiently safe bounding box in Z.
• Layers that are too discontinuous (that is, that consist of separate islands that are too small) suffer from the unavoidable imprecision in start and stop times of the extruder. Sometimes this results in prints completely failing, as the extruder wiggles around in the air millimeters away from the actual workpiece, perhaps spewing spaghetti.
• Very small layers can remain so molten that the achievable overhang drops dramatically.
• Warping during cooling can delaminate the piece and peel it up from the build platform.
• Vibration of the machine produces small but visible and palpable horizontal displacement errors.
• For some reason, polygons bulge at their vertices ("corner blobbing"), with the result that their outer edges are concave. Perhaps the extruder stops for a moment before changing direction.
• Horizontal or nearly-horizontal surfaces that are supposed to be solid sometimes come out porous, due presumably to miscalibration of filament diameter, and sometimes to other kinds of failures mentioned above on top layers.
• Stringers, of course, which are partly a problem of getting your forward and reverse pinch wheel settings right to prevent ooze, and also of having a toolpath that isn't too discontinuous; and you can wiggle the extruder around to break existing stringers sometimes. Skeinforge has a setting to try to build separated vertical "towers" each a few layers at a time, rather than potentially creating stringers between them on every layer, but unfortunately this needs a model of the extruder in order to work properly.
• Too small of a base area can make it too easy for the model to unstick from the glass.

Often one problem leads to another: maybe flexing leads to mounding, which leads to lost steps, which leads to spaghetti spewed into the void.

All of these problems can be solved with software, and e.g. Skeinforge has settings you can use to solve them already. So what am I suggesting that we do differently?

We can automate solving these problems so you don't have to use trial and error to get a working print.

Design Rule Checks

Semiconductor fabrication has had decoupling between the design and fabrication steps since about 1980, using the methodology designed by Carver Mead and Lynn Conway. As I understand it, which might be wrong, this works as follows. First you design your masks in software with some kind of abstraction of the process steps provided by your fab; then you simulate the circuits to get a good idea of analog performance; then you send the masks (originally on tape, thus the term "tape-out") to the fab, who produces some number between 10 and 100 000 of your new design, which you can test a few weeks later.

A crucial aspect of this process is a thing called "design rule checks", which you run on your masks before simulation, and which the fab

also runs on your masks before doing any photolithography. DRCs are mechanically checkable requirements which, if met, that ensure that your circuit will come out working properly despite the limitations of the fabrication process, things like "no wires less than 2Λ wide" and "minimum 2Λ spacing between wires". If your layout passes the DRCs specified by the fab, it's dramatically more likely to produce a working chip.

Shapeways publishes a sort of set of DRCs (they call them "design guidelines" or "printability checks") for STL files that you'd like them to print; for example, their ceramic page says, among other things:

Max bounding box: 340 × 240 × 170 mm and X + Y + Z ≤ 400 mm
Min bounding box: X + Y + Z ≥ 120 mm
Min density: 5% material density
Max wall thickness: 15.0 mm thick
Min embossed detail: 2.0 mm high & wide

There are another ten parameters describing what kinds of models they can print, at the STL level of abstraction. They do some of these DRCs (density, bounding box, wall thickness, model integrity) when you upload a model, while others are done manually.

But I'm thinking about DRCs a layer of abstraction down from there, because none of the problems I've seen with RepRap prints are things that printability checks would have caught, except for the bounding-box violation I mentioned above.

I want DRCs that are checks you run on your final G-Code that would detect problems like these:

• Excessively steep overhang (with different thresholds for cantilevered overhang and void-spanning overhang);
• Overhang deposited in the wrong order (spaghetti spewed into the void);
• Bounding box violations;
• Insufficient time to cool before printing another layer on top;
• Insufficient margin for error in filament diameter (resulting in possible mounding or possible unwanted porosity); and
• Ooze-producing patterns of movement (e.g. traveling immediately after shutting off the extruder)

Simulation and Optimization for Toolpath Design

To a great extent, you could use AI search techniques to find a toolpath that most closely approximates the desired shape (as expressed in an STL file) without violating DRCs, and then to optimize the toolpath for metrics like minimal plastic use and maximal speed. If you take this a little further, you find yourself simulating the plastic as it comes out of the extruder and cools, rather than using rules of thumb about allowable overhang, slenderness, and cooling time, with dozens of parameters that you have to tweak until you get reliably good prints.

(I'm assuming here that the thermodynamics and mechanics of the plastic coming out of the extruder can be characterized with a smaller set of parameters, ideally parameters that can be measured to a few significant figures rather than attempted via trial and error.)

But really you'd like to optimize things besides plastic use and print speed, while keeping the shape as a given. For example, maybe you'd

like to optimize strength or smoothness along with plastic use. But what is "strength"? That depends on the situation.

Why Are Long Bones Curved?

A lot of our long bones, like femurs, tibias, ulnas, and phalanges, are frequently subject to compressive stress. Since they're long and slender, an easy way for them to fail under compressive stress is by buckling: bending to one side or the other, giving the compressive stress progressively more leverage against the tensile strength on the outside of the bend, until the bone breaks. It's well known how to maximize resistance to buckling for a given slenderness of strut: make the strut as close to perfectly straight as possible, so that a greater force is needed to initiate buckling.

So why are all these long bones curved, since a broken bone is such a serious injury and so important to prevent?

I suspect that the answer is that early buckling, far from increasing the danger of broken bones, decreases it. The long bone's diaphysis acts as a spring, absorbing up to a certain amount of energy without damage, and limiting the forces experienced by other parts of the bone and the body until the energy and displacement are very large.

Now, I thought this was just a crank hypothesis I'd dreamed up, but it turns out that actual scientists published basically this idea in 1988: JE Bertram and AA Biewener, and they've been cited 149 times ; they cited other forms of it published in 1984.

So long-bone curvature is actually an advanced design technique for making bones and bodies less brittle. It was evolved by exploring the possibilities thoroughly with genetic algorithms, and has produced remarkably resilient and strong skeletons for our bodies by the judicious use of an unremarkable mineral.

Simulation and Optimization for Shape Design

I printed a connector to connect two plastic coke bottles together, with a vertical axis. It had a coke-bottle thread (PCO-1810) on each side, and a lip in the middle. The print took 45 minutes; as soon as I got two bottles into it, it delaminated, each bottle keeping its section of the thread.

I rotated it to horizontal, took the opportunity to thin out the lip a bit, and printed it again. As soon as I got two bottles into it, it delaminated again, but this time that meant it split lengthwise; still unusable.

I gritted my teeth and added massive ugly discs girdling the pipe. This pumped the print time up to 75 minutes, put a bunch of extra plastic in places that weren't weak, added ugly cylinder corners that made it look like a ray-gun plumbing fitting, and made the knurling on the bottlecap model I was using inaccessible to your fingers, but it was an easy few lines in OpenSCAD. This time it withstood several double-bottle insertions, only failing once I tried to really crank two bottles down onto the gaskets to see if I could get it to seal, and it only cracked instead of failing completely like the older models. It used almost three meters of filament.

Much of the problem, of course, is that PLA is comparatively weak compared to the polypropylene normally used for bottlecaps, but

more — it's brittle, in the sense that it can't be stretched very far before it breaks. But that's true of hydroxyapatite, too.

Suppose, instead, we were optimizing a feasible toolpath to provide strength *and resilience* with constraints to provide a lip against which the bottles could seal, without obstructing the bottles' threads (including as they screw in) or necks, and resisting movement of the bottles in all six degrees of freedom.

A first cut at the problem could be a huge block of solid plastic with bottle-shaped cutouts in it; a sufficiently large block would provide the required strength. (FEM simulation would be needed to find out how large.) Then, simply removing material from that block in different places would reduce the amount of plastic needed. Hill-climbing search might steer us to remove plastic less stressed by the bottle neck being screwed in tightly; some possible removals would reduce strength while improving resilience, while others would result in infeasible toolpaths. The ultimate result would surely be lighter-weight, more resilient, and more aesthetically pleasing than what I have sitting on my desk now, and it would take much less time to design and produce.

Topics

What it means that HTML is "not a programming language", and why the ignorant sometimes think otherwise

Kragen Javier Sitaker, 2019-09-09 (updated 2019-10-01) (24 minutes)

XXX tone down the arrogant pedant attitude

(Edited and considerably expanded from my comments on the orange website.)

What does it mean for something to be a "programming language"? It might seem like a trivial discussion of semantics, but as it turns out, it's important to care for semantics in order to have a conversation that conveys information, which is necessary for having culture.

Moreover, the real question here is not one of word definitions, but one of ontology: what kinds of categories exist in the real world? If we attempt to reason with incoherent categories, such as "non-elephant mammals plus mosquitoes", we cripple our reasoning. Reasoning with such categories, we are likely to believe that mosquitoes probably bear live young and produce milk, even if we haven't been able to observe these phenomena yet, and be unable to use observations of elephants to reject false hypotheses about mammals.

In the 20th century, after thousands of years of effort, the humans finally succeeded in formalizing the notion of an *logical procedure* or *algorithm* in the construct of a *computer program*, and the implications of this discovery touch on some of the most profound questions about the limits of the knowable, the foundations of mathematics, and even the nature of the physical universe; and subsequent experiments rapidly disproved many long-held philosophical notions about the nature of human thought. The concept of a *programming language* is emerging as fundamental to this far-reaching intellectual breakthrough.

Dismayingly, a recent article in IEEE Spectrum incorrectly described HTML as a programming language, so it seems to me important to clear up the confusion.

Popular confusion about programming languages

One person eloquently defended the popular confusion as follows:

> People don't care for semantics. If you write it and it causes a PC to do something (even if it's just to show a website), it's a programming language. Doesn't need to have branching, variables, etc... (not to mention that HTML includes JS and CSS) `print "hello world"`, `<b>hello world</b>`, same difference.

This seems to amount to the position "my ignorance is as good as your knowledge".

Unfortunately, this proposed definition of programming languages includes CSV, JPEG, Word, and URLs. Does that mean that anyone

who snaps a photo on Instagram is a programmer? This does not seem to be in accordance with the usual meaning of the word "programmer". Is it "programming" to write a dunning letter in Word? It seems to me like a qualitatively different activity.

Upon being presented with this argument, the misguided-egalitarian admirer of foolishness responded as follows:

> And yet people don't call them "programming languages", while they do call HTML that — so that even if HTML is not a programming language, there's still a not formally expressed difference that people can intuitively grasp with those other things...
>
> You can't program most things in SQL either (not without some modern extensions that make it Turing complete), but it is still considered a programming language...

So what is it that really distinguishes an ontologically coherent category of "programming languages" from other things, and why are the ignorant so frequently confused into thinking HTML is a programming language? What is this "not formally expressed difference" that "people can intuitively grasp"? And how much *can* you program in SQL, anyway?

The deep essence of programming languages

Programming languages are all more or less equivalent; although they have a variety of paradigms, and although they are supplied with a variety of I/O facilities, you can program more or less precisely the same set of *computations* in all of them. This is called "Turing-completeness".

Consider the differences between C, Forth, Smalltalk, Prolog, Fractran, Lisp, Brainfuck, Haskell, VHDL, amd64 machine code, Malbolge, Scratch, Octave, Python, and R: some of these do not even have branching or loops or variables, but they are all programming languages; none of them can compute anything that any of the others cannot.

The exception might be things like Turner's "Total Functional Programming", which excludes nonterminating computations without, he hopes, excluding much of practical interest; Excel, excluding the macro language, also has this limitation, but in Excel it is more onerous.

By contrast, you can't compute so much as a polynomial or NAND in HTML. That is to say, you can't program at all in HTML. So HTML is not a programming language. The same is true of CSV, JPEG, URLs, and Word documents (except insofar as Word documents might include macros).

You might say that HTML includes JS, but this is wrong. English does not include Latin, although, e.g.†, phrases like "*inter alia*" and "*et cetera*" can be used in English, and it's not uncommon for an English sentence to quote a Latin motto. Similarly, HTML does not include JS and CSS, just as it does not include the URL syntax, and Python does not include SQL or HTML; they are six separate languages.

† *exempli gratia*

An example of a borderline case: tabulating polynomials in SQL

It's fairly uncommon for people to call SQL a "programming

language", but you actually can program quite a bit with SQL; even without the recursive common table expressions alluded to above by the promoter of vulgar misconceptions, you can do surprisingly complex computations with it, and even define functions in the form of views, as long as they are finite. Certainly neither NAND nor evaluating polynomials poses any difficulty for SQL. In fact, here's an example of *arbitrary* polynomial evaluation in SQL which works in MySQL, MariaDB, Postgres, Oracle, and SQL Server:

```
select x, sum(a * power(x, e)) from data group by x;
```

In SQLite, you easily can evaluate a *particular* polynomial, or, with a bit more hassle, any polynomial up to some finite degree, but evaluating an *arbitrary* polynomial would seem to be out of reach without using recursive CTEs. Here's an example of how to do this for polynomials up to the fourth degree without recursive CTEs:

```
select x, sum(a * case e when 4 then x*x*x*x
                         when 3 then x*x*x
                         when 2 then x*x
                         when 1 then x
                         when 0 then 1 end) from data group by x;
```

The identifier `data` provides the polynomial to evaluate and the points at which to evaluate it; this can be a table or, in most dialects of SQL, provided directly inline. In SQLite, for example, in this case evaluating $5x^4 + 2x + 5$ at the points 0, 1, 2, and 3:

```
select x, sum(a * case e when 4 then x*x*x*x
                         when 3 then x*x*x
                         when 2 then x*x
                         when 1 then x
                         when 0 then 1 end)
from (select 3 as x union select 0 union select 1 union select 2) xt,
     (select 4 as e, 5 as a union
      select 0, 5 union
      select 1, 2) t
group by x;
```

This syntax works in other database engines, but in, for example, Postgres, you can instead simply say

```
select x, sum(a * power(x, e))
from (select 3 as x union values (0), (1), (2)) xt,
     (select 4 as e, 5 as a union values (0, 5), (1, 2)) t
group by x;
```

MySQL (5.5.62, at least) has `power` but requires the more verbose literal data syntax:

```
select x, sum(a * power(x, e))
from (select 3 as x union select 0 union select 1 union select 2) xt,
     (select 4 as e, 5 as a union
      select 0, 5 union
      select 1, 2) t
```

```
group by x;
```

Note, though, that even the Postgres version of this is somewhat repetitive; to evaluate the polynomial at 100 points, we would have to put 100 numbers and 192 more words into the source code. While this is not as bad as HTML, where we would have to evaluate the polynomial ahead of time (for example, with pencil and paper, or with SQL, or with an actual programming language) it still means that there are things that are easier to program in languages such as Fractran or Malbolge than in SQL — without recursive CTEs, that is.

With recursive CTEs, you can do it compactly, though fairly awkwardly, because recursive CTEs *do* make SQL Turing-complete:

```
with recursive redonditos as (select 0 as x union
                              select x+1 from redonditos where x < 99)
select x, sum(a * power(x, e))
from (select 4 as e, 5 as a union values (0, 5), (1, 2)) t, redonditos
group by x;
```

This works in Postgres (9.5.14) and, with the change explained earlier, in SQLite 3. MySQL doesn't support CTEs, at least as of 5.5.62.

Excel without macros, Coq, and GNU MathProg are other languages that can compute large classes of interesting functions but are not Turing-complete.

What kinds of generalizations are valid about programming languages?

The first and most important fact about programming languages is the Turing Tarpit Principle: they are all equivalent, in the sense that *any of them can compute any algorithm*, which is to say, anything any of the others can compute; but that does not mean that it is equally easy in all of them. ("In the Turing Tarpit," observes Fred Brooks, all computable functions are "possible, but nothing of interest is easy." "Turing Tarpit" languages like Malbolge perversely elevate this to a design goal, and so it took several years for someone to figure out how to write a working loop in Malbolge.)

In particular, the Halting Problem applies to all of them: you cannot always compute whether a program in them will continue to loop forever on a given input, or whether it will eventually terminate. Turing gave an airtight but somewhat lengthy and mindbending proof of this in 1936.

The Collatz conjecture in Python; does this program halt?

But a very simple argument that shows that this is at least very difficult is searching for counterexamples to the Collatz conjecture. In Python:

```
cn = lambda n: 3*n + 1 if n % 2 == 1 else n // 2
(x, n, m, p) = (1, 1, 1, 0)
while True:
    if m == 1 or cn(m) == 1:
        print("{} → 1 in {} steps".format(x, 2*p if m == 1 else 2*p+1))
        (x, n, m, p) = (x+1, x+1, x+1, 0)
    else:
        (n, m, p) = (cn(n), cn(cn(m)), p+1)
        if n == m:
```

```
        print("{} loops without reaching 1".format(x))
        exit()
```

This produces lines such as the following:

```
1 → 1 in 0 steps
2 → 1 in 1 steps
3 → 1 in 7 steps
4 → 1 in 2 steps
5 → 1 in 5 steps
6 → 1 in 8 steps
7 → 1 in 16 steps
8 → 1 in 3 steps
9 → 1 in 19 steps
10 → 1 in 6 steps
```

The Collatz conjecture says that this sequence, where the successor of a number n is $3n + 1$ for odd n and $\frac{1}{2}n$ for even n, reaches 1 if you start at any positive n. Lothar Collatz proposed this in 1937, but nobody has found a proof of it yet, nor a disproof. If the conjecture is true, then the program above will run forever, continuously printing lines of text, if it isn't halted externally. If it is false because there exists a Collatz cycle that doesn't include 1, the program will halt by invoking `exit()`. If it is false because it finds a Collatz sequence that increases without bound, it will instead run forever without printing anything, if it doesn't run out of memory first.

Nobody knows which of these three is the case, despite some 80 years of effort from many of the most famous mathematicians and physicists; at least one book has been published on the subject. (It is entitled *The Ultimate Challenge*.) That is, 80 years of research by many people has not been sufficient to predict the behavior of the 11 lines of code above. And you can translate that code into *any programming language*. (However, it is known that if there is a counterexample to the Collatz conjecture, it is larger than 87×2^{60}, so your translation may involve multiple-precision arithmetic and therefore be several pages of code. Python has arbitrary-precision arithmetic built in.)

The Collatz conjecture in non-Turing-complete languages

You cannot translate it into CSV, JPEG, HTML, URLs, Excel without macros, SQL without recursive CTEs, Coq, MathProg, or Turner's Total Functional Programming. However, you can translate *the computation within the loop* into most of these, at least with numbers limited to some finite size (such as 128 bits or 256 bits); that is, you can translate this part:

```
cn = lambda n: 3*n + 1 if n % 2 == 1 else n // 2
if m == 1 or cn(m) == 1:
    print("{} → 1 in {} steps".format(x, 2*p if m == 1 else 2*p+1))
    (x, n, m, p) = (x+1, x+1, x+1, 0)
else:
    (n, m, p) = (cn(n), cn(cn(m)), p+1)
    if n == m:
        print("{} loops without reaching 1".format(x))
        exit()
```

For example, here is a fairly horrific but apparently correct† translation into SQL (without recursive CTEs), tested in SQLite3 and Postgres:

```
select case when nextx then x + 1 else x end as x,
       case when nextx then x + 1
            else (case n % 2 when 1 then 3*n + 1
                                    else n / 2 end)
       end as n,
       case when nextx then x + 1
            else (case newm % 2 when 1 then 3*newm + 1
                                       else newm / 2 end)
       end as m,
       case when nextx then 0 else p+1 end as p,
       case when nextx then 0=1 else n = newm end as exit
from (select (m = 1 or newm = 1) as nextx, x, n, m, p, newm
      from (select x, n, m, p,
            case m % 2 when 1 then 3*m + 1
                            else m / 2 end as newm
            from (select 16 as x, 4 as n, 1 as m, 2 as p) state) a) b;
```

From there it is a matter of finding some external way to initialize the computation and repeat it until it results in your translation of `exit()`. (And, as Turing argued, this is a generalized property of all programs, not just this Collatz program — as long as you have infinite memory, any of them can be rendered into an even more limited form where each execution step is just a lookup in a finite table of next actions.)

However, even in this limited form, you cannot translate the Collatz program into HTML, CSV, JPEG, or an URL. So, in that way, things like Excel without macros and SQL without recursive CTEs are still *programming languages*, even though they aren't Turing-complete, while things like HTML are not.

† Implementing multiple-precision arithmetic in SQL is left as an exercise to the reader. Maybe you can use `DECIMAL` — in Postgres, `select power(2::decimal(30), 64) - (power(2::decimal(30), 64) - 1)` works correctly, but that syntax is a Postgres extension to SQL.

The Collatz program in SQL with recursive CTEs

In Postgres 9.5.14 the query

```
with recursive collatz as
   (select 1 as x, 1 as n, 1 as m, 0 as p, 0=1 as exit union
    select case when nextx then x + 1 else x end as x,
           case when nextx then x + 1
                        else (case n % 2 when 1 then 3*n + 1
                                                else n / 2 end)
           end as n,
           case when nextx then x + 1
                        else (case newm % 2 when 1 then 3*newm + 1
                                                   else newm / 2 end)
           end as m,
           case when nextx then 0 else p+1 end as p,
           case when nextx then 0 = 1 else n = newm end as exit
    from (select (m = 1 or newm = 1) as nextx, x, n, m, p, newm
```

```
        from (select x, n, m, p,
              case m % 2 when 1 then 3*m + 1 else m / 2 end as newm
              from collatz where not exit) a) b
    where x < 100)
select * from collatz;
```

appears to work, producing 1634 rows in 23 ms.

In SQLite 3.11.0, subqueries cannot refer to recursive CTEs, and it's not obvious to me how to reformulate this query to eliminate the subqueries.

Rice's Theorem

We could describe the above program as computing a partial function of the integers, namely how many Collatz iterations it takes to reach 1 from each integer. It's a *partial function* in that the program might fail to produce an answer for a given input if that Collatz sequence increases without bound. Rice's Theorem extends Turing's proof of the Halting Problem's incomputability to say that there is no algorithm to prove any nontrivial property of the partial function computed by given Turing machines, and thus by given programs in any Turing-complete programming language. The definition of "nontrivial" is amazingly wide.

Computational complexity

There is additionally an issue related to *computational complexity*: in programming languages it is easy to write a program that will take a very long time to run, even if it terminates.

The conclusion: programming languages mean trouble

This means that it is very easy to, in some sense, get ourselves in trouble in any programming language — to produce a program intended to do something in particular, but not be able to tell whether the program does in fact always do that. This leads to the widely-remarked-on phenomenon of buggy software.

Does this mean programmers are awesome and Web designers are not?

No. If we can extract any value judgment from this, it is rather the opposite: it means that programmers are constantly tripping over their own feet attempting to accomplish even seemingly-trivial tasks in their programs. This is the motivation behind the design of non-Turing-complete languages like Coq, GNU MathProg, URLs, HTML, Excel, and SQL (at least until recently): by steering clear of Rice's Theorem, we can ensure that our constructs behave in predictable ways, can be analyzed in certain ways, and can be changed in predictable ways. In the history of the WWW, this was called the "Principle of Least Power"; HTML was deliberately designed to not be a programming language due to experience with using programming languages like PostScript and TEX to represent documents.

In building any system, you should do as little as possible in programming languages and as much as possible in passive data formats like HTML.

The ongoing struggle to navigate between the Scylla of programming by copy-pasting boilerplate and the Charybdis of Rice's

Theorem has given rise to a fruitful and active area of research that continually expands the set of borderline "programming language" cases; the SQL example above shows one of the most successful cases. However, HTML is nowhere near that borderline. It is a format for passive documents consisting of marked-up text.

Perhaps some of the ignorant confuse HTML with programming languages because it's written in plain ASCII text

I have occasionally heard people saying HTML is a programming language. But they're just wrong. Presumably most people at some point thought the world was flat; we shouldn't attempt to explain away their error by saying that they meant something different by "world" or "flat" than we do. Some people think vaccines cause autism; this isn't actually because they're using the word "vaccine" or "autism" in a different sense than we are. They're just wrong. (As for the people who think organic food contains no chemicals, I'm not sure; I think some of them are just wrong, while others are in fact just using a nonstandard definition of "chemical", meaning something like "pure chemical" or "industrially produced chemical".)

In the same way, people are just wrong if they think HTML belongs to the set that includes C, Forth, Smalltalk, Prolog, Fractran, Lisp, Brainfuck, Haskell, VHDL, amd64 machine code, Malbolge, Scratch, Octave, Python, and R, rather than to the set that includes CSV, JPEG, Word, and URLs. Similarly, bash clearly belongs to the former set, and Excel (without macros) and SQL arguably do.

In a lot of cases, they seem to be reasoning based on shallow surface features like the use of plain ASCII text files. (You can see several examples of this even in the original thread discussing the issue on the orange website.) This is similar to the neural network that learned to recognize photos of tanks based on whether the photos had been taken on a sunny day or a cloudy day, or the people who think I'm "hacking into systems" when they see me using a terminal emulator — an understandable error from the ignorant and foolish, but not one we should allow to confuse our own thinking.

After all, to an illiterate person, a page full of random letters and spaces looks much the same as a page of a novel or a page from a legal brief, and quite visibly different from, for example, a painting of the Virgin Mary or a pornographic sculpture. We would not therefore assert that the legal brief is "fiction", nor the page of random letters "legalese". But an unlettered peasant might make such an error, just as they might assert that the Earth is flat.

This, I think, is the "not formally expressed difference" that "people can intuitively grasp": if you don't know how to read it, an editor window full of HTML looks similar to an editor window full of JS, and different from an editor window with a Microsoft Word document in it, because the font has a fixed width and all the letters are the same font size. That's what people intuitively grasp.

The surprise is to find that the editors of IEEE Spectrum have fallen to such abysmal levels of ignorance.

Is it bad to be ignorant?

It is *unfortunate* to be ignorant, not *reproachable*. Reproaching someone for their ignorance is contemptible, like reproaching poor people for their poverty or sick people for their sickness. But to *spread* ignorance is to impede others from escaping that unfortunate situation, and that *is* reproachable. It is malfeasance as surely as robbery or deliberately infecting people with diseases.

Topics

Where did the Rubius comic book come from?

Kragen Javier Sitaker, 2017-01-10 (4 minutes)

It was 2015-10-13. Holy shit, I had future shock. I'd just bought a hardcover comic book (A4 size, full color, sewn in signatures) of Rubius. The Youtuber. At a newspaper stand.

Who is Rubius? He had 14 million subscribers on YouTube; now, 2017-01-10, he has 22.6 million. Has there ever been a public access cable show with 22 million fans? Mostly they're gaming videos. He describes it as follows: "My channel of Gayplays of Minecr... Wait. No. My channel is of Gayplays in general, but I never play anything predefined. Some days you'll find horror games, others fun games, others indie games, etc., but I don't only upload Gayplays!"

I have no idea what "gayplays" are.

I'd heard of Rubius previously because when he had come to Argentina the year before, 2014, there was a problem when his fans showed up at the airport.

Paul Visscher estimated that he easily makes 7 or 8 figures per year with his six million views per video.

I'm just surprised to live in a world where a gamer uploading videos to YouTube is merchandising his brand via comic book spinoffs.

I looked into the economics of the comic book slightly. I bought the book for US$9.50. The printing is done, not somewhere in Asia, but by Cartoon S.A. in Salta, Argentina. The Argentine copy I have says it's the second edition (which I think means "printing"), printed in 30,000 copies. The artist is María Dolores Aldea, aka Lolita Aldea. She posted the first 10 pages in PDF on her web site, from which we can see that the Spain version of the book is printed by Unigraf SL, a Spanish company, in Catalonia, on the Mediterranean coast of Spain, near Valencia. They also print invitation cards and personalized paper napkins.

Cartoon S.A. is a little harder to figure out, but it looks like they mostly do graphic design and printing for brochures and stuff like that.

It seems like the "publisher", as distinct from the two and presumably more printing companies, is the "children and young adult" imprint of Grupo Planeta. It doesn't seem to be a vanity press like, say, Blurb or Lulu. It was founded 70 years ago.

So, I'm thinking that Editorial Planeta ("Planet Publisher") probably bought personality rights from Rubius and are paying royalties to him and the cartoonist. Most of their kids' books seem to be cartoon books where they licensed the characters from someone else: Maya the Bee (although the original book is out of copyright, the book looks like a spinoff from the TV series), Violetta, Pixar's Planes. Apparently this is their second Rubius book; the first one was "The Troll Book".

The thing that surprised me was not so much that there is a guy who is well-known among Spanish-speaking video game fans — his videos are in Spanish — but rather that he was *so* well-known that

random newspaper stands on the street were selling comic-book spinoffs from his YouTube channel. This was future-shockish to me. Again. Like in 1994 when suddenly the number of internet books in bookstores went from like 3 to like 300 in the course of a few months, or 1996 or 1997 when suddenly advertising had URLs in it. It's kind of mindblowing to me that a gaming channel on YouTube is now a licensable brand on the same level as Disney shows and Pixar movies. And this was apparently their "most commented" book in the "young adult" category, although the Violetta and Maya and Planes books are in a different category; *Violetta. En mi mundo* is one example.

It's also interesting that this 70-year-old press is outsourcing the actual printing of these books to random graphic design storefronts in different countries.

Topics

- Economics (p. 3424) (33 notes)
- The future (p. 3746) (20 notes)
- Youtube

Notes and calculations on building luxury underground arcologies for whoever wants them

Kragen Javier Sitaker, 2013-04-17 (updated 2019-08-27) (66 minutes)

Arcosanti was Paolo Soleri's project to build the city of the future in the Arizona desert. The Venus Project is Jacque Fresco's project to build it in the Florida swamps. They have been noticeably less successful than other efforts such as Burning Man, which gives the appearance of having been inspired more by *Mad Max: Beyond Thunderdome* than *Star Trek*, or Dubai. One of the cornerstones of Soleri's vision was the "arcology", a kind of building-sized self-sufficient city.

Here I try to imagine what kind of arcology I would like to live in.

Some of this will probably seem outdated in a short decade or two, in particular my concern for energy conservation. Rather than seeing a movement toward energy self-sufficiency in small groups, I think we'll see something like another decade of energy scarcity, ending around 2026, as bigger and bigger fractions of the globe are devoted to harvesting solar energy, now that it's finally become cheaper than fossil-fuel or nuclear energy.

The population density question

A basic question about such structures is how much population density they can accommodate. For self-sufficiency, they must necessarily use no more energy than they can harvest from their environment; in the simplest case, with no deep drilling and fracking to harvest geothermal and no nuclear reactors, this is merely the solar energy available to them.

The NREL Solar Resource Maps show a photovoltaic solar resource of 125–250W/m² in the US, which they have unfortunately chosen to state in non-SI units as 3–6 kWh/m²/day.

This is necessarily the energy supply before the inefficiencies of conversion to electricity; the solar constant above the atmosphere is only 1400W/m², the sun only shines at most half the time for 700W/m², and it's not direct all of that time; reaching 250W/m² *after* photovoltaic conversion would imply panel efficiencies of 50%, which has not yet been achieved.

A human being normally needs to eat 100–120 W (unfortunately typically rendered in non-SI units as 2000–2500 kcal/day), equivalent to about 5–6 mg/s of carbohydrates or protein, in addition to micronutrients; but micronutrient needs can be satisfied at an insignificant energy cost.

(For brevity, I'll frequently use the Chinese 人 to mean "person" in what follows.)

Dividing these two numbers, we have an *absolute maximum* population density of about one or two people per square meter in temperate areas like the US. A standard 100-meter-square city block could then accommodate some ten or twenty thousand people, which works out to one or two million per km². Wikipedia tells me this is

much higher than any real city:

City	人／km²
maximum for self-sufficiency	1500000
[Friendship Village, Maryland][2]	32000
Delhi	29000
Ahmedabad	22000
Manhattan (in New York City)	27000
Chennai	26000
Mumbai	23000
Paris	21000
Cairo	18000
Buenos Aires	17000
New York City	10600
Taipei	9600
Shenzhen	8600
San Francisco	6600
Hong Kong	6500
Tokyo	6000
Los Angeles	3000

However, real solar plants (see the "Utilities → Food" section below) don't even approach 100% conversion efficiency, nor does any known means of conversion of sunlight to electrical, chemical, or mechanical energy. Typical large-scale electrical efficiencies are 16% for the sunlight that hits the panels, but the panels are placed some distance apart so they don't shade each other when the sun angle is suboptimal — since panels are so much more expensive per m² than sunny land, emphasis is placed on not wasting panels, rather than not wasting land — so typical yields are more like 11W/m² for Optisolar's Sarnia project, 6.25 ac/MW = 40W/m² for Nevada Solar One, 15.5MW/85 acres = 45W/m² for SunEnergy1's Duke Energy project.

There are currently-shipping mass-producible thin-film panels that hit 20%, space-deployed very expensive multijunction semiconductor panels that hit 40%, and concentrating solar can hit the usual 35–40% for large fixed steam turbines. There are some potential nice synergies here: photovoltaic panels, including multijunction panels, are nearly black, so the sunlight that photovoltaic panels fail to convert to electricity is mostly converted to heat rather than reflected, which means you can use it to make steam, and concentrating the light means you can justify the use of much more expensive photovoltaic cells, but if you're concentrating more than one or two suns on your cell, you'll need some kind of active coolant to keep the cell from melting — such as steam! So in theory you might be able to hit 50% or 60% conversion efficiency with the combination, but nobody's done it yet.

Modern life involves using energy beyond the bare necessity for survival, generally quite a lot of it, mostly bought on the market rather than harvested for direct use. A self-sufficient arcology would probably have to be less efficient in its material energy use than the average person in a modern society, who takes advantage of economies of scale and specialization in many kinds of goods. The USA DOE EIA AER estimates 2011 marketed energy use in the USA at 97.30 quadrillion Btu/year, another absurd non-SI unit; this

quantity is actually 3.243 terawatts; if we figure 311.8 million people, that's 10.4 kW per person, the equivalent of about 100 slaves per person, Buckminster Fuller's "energy slaves".

Let's assume that the overall solar conversion efficiency hits 30%, rather than 50% or 65% or staying at 16%, and that the overall electrical energy use of the inhabitants of the arcology is one-third lower than current US marketed energy use, for the following reasons:

- They don't need to heat or cool their houses actively or separately; the arcology's surface area per person is much smaller and well-insulated, and it uses primarily passive climate control.
- They spend very little or no energy on actively chilling their food; the arcology provides a cold reservoir which provides this service free.

- They spend no electrical energy on daytime lighting, because light is provided more efficiently through a system of lightguides.
- Because the arcology is self-sufficient, they don't need to spend much energy on transporting goods to it, or transporting themselves to goods.

If we need 10.4 kW per person, each person would need 83 m² at 125 W/m² or 42 m² at 250 W/m², giving a density of some 12000–24000 人/km². If we double this, on the assumption that we'll lose a factor of 2 in efficiency to generalization, we get 6000–12000 人/km².

XXX what about the factor of 5 in efficiency you lose to current PV cells? Huh? I don't think the NREL figures cover that! RECALCULATE EVERYTHING. unless NREL *does* cover that.

XXX if I just multiply everything by 5 it stops being an arcology at all. Possible optimistic assumptions:

- maybe solar energy efficiency will improve, so that you get, say, 33% of the light, instead of 20%, due to combining concentrating solar with photovoltaic?
- maybe people in an arcology won't use as much energy, because people outside an arcology use a lot of energy to heat and cool their houses and foods, about half of it; so perhaps they'll use 33% less.

With these two 2:3 optimistic assumptions, you use only 4/9 of the current energy and therefore need only 4/9 of the land area.

XXX recalculating again from first principles: current low-cost PV panels are 16% efficient, and current utility-scale PV capacity factors range from 10% (in Germany) to 29% (in California) or a bit more, with the US average being around 24%. (See Japan can achieve energy autarky via solar energy, but not much before 2027 (p. 2819).) Those factors are normally calculated relative to a nominal "solar constant" of 1000 W/m², so a 20% CF means 20% · 16% · 1000 W/m² = 32 W/m². This means that 10.4 kW is **325 m²**, quite a bit more than 42. This gives a density of only 3000 人/km². So it looks like the NREL figures *didn't* account for PV inefficiency.

This is 10× the threshold population density where the EU defines an area as urban but about one sixth the population density of Buenos Aires.

XXX food refrigeration is perhaps a fundamental utility service!

42 m² by itself is a fairly comfortable, if smallish, apartment. So

this kind of "arcology" could comfortably be constructed, at first, as a city block of single-story construction, which means you could bootstrap it fairly incrementally; you don't have to start with a megastructure.

But if you want a megastructure — especially a pleasant one — it's far cheaper to build parts of it all at once, due to certain economies of scale.

Envisioning universal luxury dwelling

Contrary to the usual Cold-War-era utopias, I don't envision an above-ground mass of megalomaniacal architect-ego-worship, nor some kind of giant right rectangular parallelepiped with sad little rectangular holes in its bare gray concrete walls, nor gleaming cubes and towers — I'd prefer to leave the above-ground surface as a park, with crops, meadows, trees, perhaps a forest with some babbling brooks, and keep the human habitation beneath the ground. Sunlight can easily be brought downstairs with the occasional translucent artificial stone cemented in place below the dirt-line, and ventilation requires a few wide ventilation towers, which can be constructed in organic shapes and beautified with vines.

Subterranean dwelling is not a new idea, going back of course to Paleolithic hunter-gatherers, and featuring in many science-fiction stories, but here I explore why it's historically been marginal (mostly confined to extreme conditions and military surplus reuse), why we now have the option to change that, and how completely fucking awesome it could be.

Underground construction is tricky and expensive, particularly in wet areas where the water table approaches the surface. Here in Buenos Aires, each construction site excavation vents a continuous stream of seepwater into the gutter, proceeding, I suppose, from a pump down at the bottom of the pit. Water tends to trickle and seep in through porous materials like concrete, needing continuous pumping to keep it from eventually filling your living space. Less porous building materials (e.g. concrete sealed with water glass or resins) reduce the seepage, but they don't eliminate it. Even missile silos in the Great Plains fill up with water when left unattended.

There are also risks like collapses, poisonous and asphyxiant gases, and explosive gases; these three can be minimized during construction by digging a pit instead of a cave, but in many places, further measures must be taken after construction to prevent them.

But underground construction has advantages. Aside from the benefits of being able to enjoy the land surface as a park, underground construction is silent, naturally temperature-stable, and much less vulnerable to terrorist bombing — perhaps not a concern wherever you live at the moment, but that's what people here in Buenos Aires thought until Hezbollah blew up the Israeli Embassy and AMIA in the 1990s.

Suppose, though, that instead of a single story of underground construction, we have ten stories, but still one person per 80 or so square meters of ground area (6000 人/km², comparable to the density of Tokyo, in between Los Angeles and San Francisco or Hong Kong). Now the average person has 800m² of live/work/storage space, some of it shared with others, plus 80m² of park; the average couple has 1600m².

This is enough for a sort of underground termite mound of palatial mansions. A normal bedroom may occupy some 16m²; 50 of them will fit into 800m². Every man, woman, and child could easily own their own 12-bedroom mansion.

Hearst Castle, that classic of robber-baron gilded-age excess with its 56 bedrooms, occupies some 8400m², so 800m² of floor per person is enough for truly palatial living. The question is whether we can make that kind of thing affordable to everyone.

Excavation volume and costs

Summary: it looks like it costs about US$30k/人 to dig the hole using current mining methods, although that's still really far from the fundamental efficiency limits.

Volume: 3200m³/人; weight: 7000Mg/人; energy: 1.4GJ/人 or US$20/人

If the average height of a story is 4m — with some ceiling height variety to allow some rooms to be cozier and others more majestic — this is 400k(m³) of excavation, some 3200m³ of excavation per person, perhaps 7000 tons of dirt and stone to remove. I have no idea how to estimate the economic cost of this, except that it is one of those things that has truly enormous economies of scale, since big strip-mining steam shovels are vastly cheaper per ton of rock than little backhoes; but the *energy* cost of listing 7000 tons by an average of 20 meters is some 1.4 gigajoules.

Energy is commonly priced in yet another non-SI unit, the megawatt-hour (MWh), at typically around US$40 wholesale (US$11/GJ), and two to five times that retail. A megawatt-hour is about 3.6 gigajoules, so we're talking about an energy cost of excavation of some US$20 per person.

(Looking at it another way, at 125 W/m², 1.4 gigajoules per 80m² is about 39 hours' worth of energy production.)

Landscaping excavation techniques would cost US$1.5M/人

Some random web site says that typical excavation costs are US$299.64 to US$423.77 per cubic yard (another non-SI unit of some 0.764 m³), which would place the cost of excavation of the above at some US$1.3–$1.8 million per person. However, I'm fairly confident that that's a backhoe cost, not a strip-mining cost, while what I'm envisioning is more a strip-mining kind of operation (or "surface-mining", as the miners like to call it these days), followed by construction and surface restoration.

Strip-mining excavation techniques: US$30k/人, 4 months

The Global Surface Mining company web site boasts of a trial where one of their surface-mining machines removed 830 tonnes of limestone per hour of operation. At this rate, you could excavate the 7000 tons for the habitation of one person by using one machine for a single ten-hour shift; excavating an entire 100-meter-square city block, digging self-sustaining luxury homes for 125 people, would take some one to four months.

But how much would that really cost? An online Open Pit Mine Model seems like it might be more relevant:

This mine is an open pit mine producing 5,000 tonnes ore and 5,000 tonnes waste per day. The total resource to be mined is 18,715,000 tonnes. Ore is hauled 1,068

meters to an ore stockpile. Waste is hauled 535 meters to a waste rock dump. Rock characteristics for both ore and waste are typical of those of granite or porphyritic material. Operating conditions, wage scales, and unit prices are typical for western U.S. mining operations. ... November 2007

Some further figures drawn from that model:

Hours per shift	10
Shifts per day	2
Days per year	312
Total Hourly Personnel	38
Total Salaried Personnel	15

Supplies & Materials	$/tonne ore	$2.07
Labor	$/tonne ore	1.96
Administration	$/tonne ore	0.82
Sundry Items	$/tonne ore	0.48
Total Operating Costs		$5.33

Total capital costs	$15,988,500

More than half the total capital costs are equipment, including a single 4½-cubic-meter hydraulic shovel. There are two four-kilowatt pumps included, presumably because of the water seepage I mentioned earlier; that's a heck of a lot of seepage.

So these guys are spending US$5.33 to remove two tonnes of rock (one tonne of ore and one tonne of waste), plus their capital cost. If we were to use the same equipment and methods to dig a 40-meter-deep hundred-meter-square hole, removing 875000 tonnes of rock and soil — let's just call it a million tonnes — then it would have an *operating* cost of US$2.7 million, and at 10 000 tonnes per day, would take 100 days, almost four months. If you divide by the 312/365 days they work, you get 117 days.

If you divide that among 125 people, it's US$21000 per person. Not insignificant — certainly much greater than the US$20 per person that is the inherent cost of the energy to just lift the soil up — but not an overwhelming cost in the world of construction, either.

But wait! I haven't accounted for the capital costs yet. If we figure an interest and depreciation rate of some 20% per year — high for the US, but probably reasonable for much of the world — we end up spending about 6% of the capital cost on interest and depreciation. That adds up to about US$960k, bringing the total cost to about US$3.7M, or US$30k/人. (This is assuming, perhaps optimistically, that capital line items like "Engineering & Management" and "Buildings" represent things you could sell without too much loss at the end of the operation.)

About a third of that cost is labor, and the population of the dwellings is more than double the size of the digging crew, so it seems likely that you could reduce the money cost by a third by means of DIY, or perhaps half or more, if you use methods that are more labor-intensive and less capital-intensive.

It seems possible that a larger-scale excavation operation would be more efficient, but given the number of equipment items of which only a single item is present, it seems likely that a smaller-scale one would not reduce the cost by much.

These fairly enormous capital equipment investments probably explain why underground construction is not traditional, because without them, it's not only dangerous but ruinously expensive.

Alternative to excavation: build a hill

As an alternative that involves moving less dirt, and might help with water seepage, you could build your arcology essentially above ground, but then cover it with dirt. The dirt sloping away from the center will encourage the water to run off, and so the water table won't rise toward the center of the hill as much as you think it might; and you can run drain pipes underground to provide passively-safe drainage and seep protection for the buildings inside.

This is basically your classic science-fiction pyramid-shaped arcology, but with a layer of dirt on top of it, and with its outer walls restricted to a slope of at most 30° above the vertical in order to keep the soil from slumping off. Since in this case the base will be four times as wide as the height, the bh/3 pyramid volume formula in this case reduces to $16h^3/3$, times or divided by some minor deviation from squareness. If you want it to have the $125 \times 80m^2 \times 4m \times 10 = 400k(m^3)$ volume of the previous excavation, it needs to be about 42 meters tall and 170 meters wide at the base; if its surface is covered in three meters of soil (vertically, not perpendicularly), that's about $85000m^3$ of soil, or about $677m^3$ per person, less than a tenth of what you'd have to remove for the excavated version.

A circular mound is slightly taller, narrower, and with less surface area; a triangular or kidney-shaped one would be the opposite.

While this might seem more handicapped-accessible than the purely-underground version, you could of course add small hillocks to that version to provide ramp or elevator access, or ordinary small buildings, as in the Terra Vivos entrance near Barstow.

In the USA, it used to be that once a hill was 1000 feet (almost 305 m) tall, you could call it a "mountain" (although the USGS no longer has such a hard-and-fast rule). So if you scale this proposal up by a linear factor of 8 (an area and population factor of 64, a volume factor of 512), you get 8000 people living inside an artificial mountain, each with an average of 25600 m^3 (6400 m^2). The construction cost per person would remain similar.

Concrete: US$14000–45000/人

Once you have a gigantic hole in the ground, or a building site for a hill, before you can landscape its roof into a park and start building mansions and cathedrals inside, you have to build that roof, and also stop up the seepage from the walls. As far as I can tell, the essentially universal technique at this point in history is to use reinforced concrete, made with portland cement, for floor, pillars, and ceiling, and either concrete or something similar for the walls to keep the water out.

It's conceivable that some other kind of construction technique could work and be cheaper, but I don't know of a candidate.

The surface area of a 100m×100m×40m hole in the ground would be $(20000 + 4*4000)m^2 = 36000m^2$, so each cm of thickness of concrete means $360m^3$, or about 900 tonnes, or 7.2 tonnes per person, of concrete that will be needed. The surface area of a 170m×170m×42m pyramid would be $(170\times170 + 170\times170\times\sqrt{5}/2)m^2$

= 61000m², 610m³/cm, 1.47Gg/cm, or 11.7 tonnes/cm/人.

Unfortunately, I don't have any idea how thick concrete needs to be for this kind of construction. If I guess (I hope reasonably?) that it needs to be some 20cm thick, we end up with:

	total concrete	total concrete	concrete mass/人
square pit	7200m³	18000 tonnes	144 tonnes
pyramid hill	12200m³	29000 tonnes	230 tonnes

These numbers are smaller than the amount numbers for the amount of rock and dirt that need to be removed from the hole, by about two orders of magnitude, so I assume that the labor cost to put this amount of concrete in place will be small compared to the labor cost to dig the whole. Nevertheless, concrete costs US$100–200/tonne, so we're looking at a materials cost per person of US$14000–45000.

The cement in concrete costs about 330–660kWh/m³, which is to say 1.2–2.4GJ/m^3, to produce. At US$11/GJ, this is a very small fraction of the dollar cost of the concrete. At 10.4kW/人, that's about 33–66h/人/m³, so we're looking at about 80–160 days of the arcology's population's usual total energy usage to produce the cement; or, if the arcology were to produce its own cement, 40–160 days of its own energy harvest. In practice, it's probably not reasonable to expect the arcology to be self-sufficient before it's built!

You can probably strengthen the walls substantially and avoid the need for internal support pillars by making them not flat — like an eggshell, they should be everywhere convex outward, except possibly t the bottom — and veining them, like a leaf, an insect's wing, or a plastic injection-molding.

Once you're done with the excavation and sheathing it in concrete, you're done with the parts of the traditional construction process that benefit enormously from economies of scale. Everything past this point can be done incrementally without dramatically increasing its unit cost.

Possible future concrete alternative: biocementation

I was interested in Magnus Larsson's 2009 "Dune: Arenaceous Anti- Desertification Architecture" proposal a few years back to cement desert sands using urea, Bacillus pasteurii bacteria that ferment it into alkaline carbonate, and calcium chloride to form calcite from the carbonate; but nothing seems to have come of it, and the process seems to produce hazardous quantities of ammonia, as anyone who's inadvertently fermented urea can tell you. So it might be a cheaper alternative at some point, but isn't yet; watch Ginger Krieg Dosier's startup "bioMASON" to see if it works out! (Dr. Dosier Tweeted me that in her process, the "by-product is captured in a closed loop system", by which I assume she means they make their bricks in a hermetically-sealed chamber and bubble the ammonia through a sulfuric-acid solution, or maybe just dissolve it in water and recycle it into new urea.)

If it does work out, we can expect the result to be as hydraulic as portland cement, slightly less porous, and perhaps significantly cheaper. You don't need a cement mixer, the raw materials are urea and calcium chloride, and you may be able to use aggregate in-situ by soaking it with the water-soluble cement components. Portland

cement costs US$110/ton, and comprises between a quarter and half of the mass of concrete, and therefore about a third of its cost. Urea costs about US$400/ton, while calcium chloride costs about US$200/ton; but the resulting calcium carbonate will not include most of their mass, since the chloride ion and the majority of the urea (both its amines, which is to say, everything except its carbonyl) are merely waste products.

Even if it's possible to substitute raw piss for industrial urea, it would not reduce the cost substantially. Piss is only about 1% urea, So you'd need 100 tons of piss per ton of urea, and I think you get 1 mole of carbonate per mole of urea, which is to say 60g of carbonate or 100g of calcium carbonate per 60g of urea — that's assuming the other oxygen comes from dissolved O_2, not from more urea molecules — so you'd need 60 tons of piss per ton of cement, or 8600–14000 tonnes of piss per person. At a rate of 1–2 ℓ of piss per person per day, this would amount to 12–38 millennia of piss production. Gathering the piss of millions of people for the construction project would cost more than simply buying industrially-produced urea.

Reinforced biocement would seem to pose a couple of major problems. The first, which may not matter in this application, is that calcium carbonate expands and contracts very little with heat, while portland cement has an expansion coefficient similar to that of the reinforcing steel, so reinforced lime cement will tend to crack when exposed to thermal stress — which hopefully subterranean construction won't experience! The second is that chloride ions can corrode the reinforcing steel even in extremely alkaline environments, so to reinforce biocement with steel, you probably need a source of calcium other than calcium chloride. Unfortunately, the other soluble calcium salts that occur to me (calcium bromide and calcium iodide) are much more expensive, and furthermore, I'm not sure they wouldn't cause the same corrosion.

An alternative that's been sometimes explored in the past, which might solve both problems, is reinforcing concrete with bamboo rather than steel. You need a higher fraction of bamboo than steel to reach the same strength, but I'm pretty sure chloride doesn't degrade the cellulose that gives the bamboo its strength.

Possible concrete alternative: submarine electro-accumulation

As reported in the March/April 1980 *Mother Earth News*, you can get a concrete-like substance by electrolytically extracting minerals from seawater on submerged wire armatures; futurist architect Wolf Hilbertz founded companies in the 1970s and 1980s to commercialize the project for, among other things, healing reinforced concrete that had been damaged by corrosion. The resulting accretion on the cathode, principally composed of aragonite and brucite, is known as "seacrete" or "biorock"; it can grow about 5cm/year, and the efficiency is about 400–1500g/kWh (100–400kg/GJ). The strength is comparable to concrete when accreted at 5–100mA/ft² (50–1000mA/m²); typically you use 12V.

If it were logistically feasible to use seacrete to construct the arcology, the 144–230 tonnes needed per person would therefore cost 350–2000GJ, or US$4000–23000.

However, discussions on the Seasteading fora debate this, claiming the real efficiency is closer to 50g/kWh (14kg/GJ). If this is correct,

it would push the cost an order of magnitude higher than concrete.

Utilities

To keep the structure fit for human habitation, it's necessary to provide light and clean air at comfortable temperature and humidity; to exhaust or consume CO_2; to prevent the growth of toxic or allergenic molds; to haul away garbage; to provide food and drinkable water; to provide means of ingress and egress; to limit the spread of vermin such as cockroaches, fleas, mosquitoes, mice, and rats; to contain, ventilate, and neutralize eruptions of noxious chemicals (say, when some poor dummy foolishly mixes bleach with ammonia while cleaning — or just when someone has a night of farts with a lot of hydrogen sulfide, or burns his toast); and to deal with sewage, ideally by composting and refining it. It's also highly desirable to provide electrical energy, internet access, individual climate control to taste, hot water, compressed air, and vacuum suction for cleaning, not to mention some way to wash your laundry and stuff.

Many of these problems are simply slightly-larger-scale versions of the problems that face large contemporary apartment buildings, but some are not, and some admit better solutions than the traditional ones.

The compact shape of an arcology makes such utilities easier to provide. In the 100×100×40 pit, for example, you have 400k(m^3) of space within a 74-meter radius; you never have to lay more than 74 meters of pipe or duct to reach anywhere. The corresponding distance in the pyramid is 119 meters.

Light: skylights, lightguides, and electricity at night

The most obvious question about living underground: "Won't it be too dark?"

It depends. It can be a great deal brighter than many current dwellings. Aboveground, nobody lives in a glass house, because if you stop convective and radiative cooling with glass, you get a massive greenhouse effect, and you can rapidly reach dangerous temperatures — think of a toddler locked in a closed car in the sunshine. In above-ground passive solar design, according to Daniel D. Chiras's *The Solar House: Passive Heating and Cooling*, typical allowable glazing on sun-facing walls ranges from 7% to 12% of the floor area, depending on issues like thermal mass, overhang, and local climate.

That means that the above-ground passive solar houses we think of as "luminous" and "bright" are already between 88% and 93% dimmer than direct sunlight, just in order to remain at a livable temperature without outrageous amounts of heating and air conditioning!

And traditional construction techniques (the picturesque clapboard and adobe houses we all know from landscape paintings) more or less adhere to these limits.

The skylights: 4m²/人

Bringing 10% of the sunlight underground *would* more or less imply that 10% of the land needs to be skylights. (You can also illuminate electrically, but this adds the 80–85% inefficiency of the photovoltaic panel to the 85% inefficiency of the LED or fluorescent tube, giving you a total of 97–98% loss and therefore requiring 30–50 times as much land area; better to bring the light inside directly when you

can.)

However, first of all, the skylights don't need to look like windows. They can, and should, take the form of massive glass or glass-like sculptures, frosted luminous stones, clear-bottomed waterways, or gigantic artificial formations of quartz, sapphire, fluorite, or salt crystals; quartz crystals can be grown at a millimeter per day with the hydrothermal method, and giant thick, but hollow, glass or acrylic vessels can be filled with a cheap material with a similar refractive index, such as water. LiTraCon-style embedding of optical fibers in an opaque matrix can even produce mostly-opaque rocks which nevertheless conduct a substantial fraction of the light that strikes them down into the ground.

Second, we only really need to illuminate the rooms we're in. If each of our 125 inhabitants is, most of the time, in a room of some 3×6 meters, we only need some 2m² of skylight per person — let's say 4m²/人 to be safe, for a total of 500m² out of some 10000m² — and beam the light down to where we want it using light pipes, either fiber optics or air-core metallic lightguides. They don't have to be very high-quality fiber optics, since the light only has to travel some 40 meters at worst, and losing half of it over that distance is tolerable, as long as it doesn't damage anything (e.g. melting it). That is, you could lose 3dB/40m, or 75dB/km, and be fine.

Generally it's a lot easier to increase illuminance than to decrease it, so it might be a good idea to have a few skylights that focus sunlight to provide more than one sun of illuminance for high-illuminance applications.

How to carry light down to the depths: lightguides

Lightguide illumination has been deeply investigated already as a way to illuminate buildings, and it works.

For solid fiber-optic lightguides, I think the *circumference* of the lightguide must be proportional to the absorptivity of the material, or a bit more, to prevent overheating; this would seem to suggest bundles of thin lightguide cables so you can blow air through them. The total power absorbed in the fibers will be quite significant, at least if you don't use high-quality glass: if you're collecting 500m² of sunlight at the surface, that's about 500kW, and if you absorb half of it in the fibers, you need to dissipate 250kW from the fibers. Liquid-core lightguides could use the liquid itself as the coolant, providing hot-water heating from the light absorbed from the lighting system.

Typical optical fibers used today for communication have attenuation of about 0.3dB/km, which is about 0.012dB in 40 meters; so if you managed to use them, they would absorb only about 0.3% of the sunlight, or 1.5kW, so you could use about 30× less of them, by circumference, than of some cheaper material. However, they cost a lot more than 30× as much, they're a lot trickier to connect, and they'd need complicated light-gathering and great care taken with the highly concentrated sunlight within them, so it wouldn't be worth it except perhaps for special applications, like solar surgery or welding.

Ocean water attenuates at about 10dB/75m or 5dB/40m, despite its particulate content, so transparent pipes full of pure water might work fine as light pipes for this purpose if you don't bend them too sharply, as long as the pipe material's refractive index isn't too much

higher than the water's. (On the other hand, other published attenuation coefficients for seawater point at more like 10dB/40m in even the clearest waters; apparently it depends on the wavelength range you consider. Some book on the Chesapeake Bay says, "Lorenzen (1972) estimated the attenuation due to water alone to be 0.038 m^{-1}, though his measurements were for deep ocean conditions, in which measurements generally commence at depths > 5 meters."; this works out to be 0.17dB/m, or 6.7dB/40m. Pope and Fry's 1997 measurements find absorption coefficients varying by wavelength from 0.00442/m at 417.5nm, which they point out is "more than a factor of 3 lower than previously accepted values...unquestionably a result of contamination by [the other experimenters'] stainless steel cell", up to 1.678/m at 727nm; I don't even know how to interpret an absorption coefficient >1, but if the other one meant what I thought it did, it's 0.019dB/m or 0.77dB/40m, which is quite acceptable. If too much of your light at depth were blue, you could use fluorescence to reconvert some of it to yellow; I used to have a dark green plastic bottle that fluoresced bright yellow in the light of my blue LED, but wouldn't fluoresce in ultraviolet light at all.)

Fiber optics have the advantage that you can "switch" them by filling junctions with liquid water, which allows light to flow instead of suffering total internal reflection.

I don't know if you can get reasonably cheap glass, acrylic, or polycarbonate of sufficient clarity. Apparently Toshikuni Kaino et al. (Applied Physics Letters 41, 802 (1982)) describes an acrylic (PMMA) plastic with a 50dB/km loss, which would be ample. Apparently this is what is now known as "POF", is widely available, and costs US$0.25/m in 1mm inner fiber diameter, but that stuff has unacceptably high losses of up to 200dB/km; so I suspect the answer is no.

Alternatively, you might be able to use an air-filled pipe lined with aluminum, aluminized mylar, silver, or gold, to a thickness of a few dozen atoms. The trouble is that you lose energy on every reflection. Aluminum is only, at best, about 92% reflective in the visible, so you lose 0.36dB per bounce, while gold reaches 97% in the red, thus losing only 0.13dB, but absorbs a lot of blue and even yellow light. Silver nearly hits 97% across the visible. With sufficiently wide light pipes, say bigger than 50cm, these could maybe perform better than fiber-optic light pipes full of water.

What to do with light in the depths: luminaires

The actual luminaires illuminated by that the sunlight delivered through the lightguides can be of almost any form. You can illuminate the water in an indoor fountain or pool, a crystalline object such as a large salt crystal, or anything made of frosted glass. A planar lightguide made of glass could even out the illumination over a large surface made of something like marble bonded to it. Or you could backlight a water-cooled LCD display as a sort of virtual window.

Fluorescence and filters (dichroic or otherwise) can be used to alter the colors of the lighting.

Dimmer lighting with lightguides

You might prefer to have dimmer lighting some of the time, even during the day; for example, many computer screens are more readable in dim light. Direct sunlight, like sunbathing on the beach, is

around 100klux; full daylight out of the sun is around 15klux; a cloudy day is around 1klux; a typical office is around 400 lux; a typical living room at night is around 50 lux; you can see when things are illuminated with around a millilux; and you can see light sources down to a few nanolux.

Accordingly, if you choose to light your office like a typical office during the day, instead of using 4m² of skylight for yourself, you'll only be using 0.04m², leaving more for other arcologists.

Fallback lighting with LEDs and fluorescents

You can, of course, use electricity to get lighting at night or when you want more light than is available from the sun. The best LEDs are around 200 lumens per watt, and a lux is a lumen per square meter, so to reach 15klux with those you'd need 75 watts per square meter, and about US$75 per square meter. Fluorescent lights are a little lower in efficiency, but also in cost.

10.4 kilowatts is enough energy to run quite a lot of incandescent lights, too, but you probably want to avoid doing much of that, just because it will contribute undesirably to heating, adding extra heat that needs to be removed.

Underground rain-forest gardening

One of the key features of historical luxury homes has been their gardens. But, aside from the park at the surface, it might seem that we don't have sufficient lighting to maintain hundreds of square meters of lush gardens per person; after all, to preserve the surface, we're only letting a small fraction of the sunlight into the arcology.

But, if our purpose is lush vegetation rather than high agricultural productivity, a small fraction is plenty.

Photosynthesis and productivity in different environments, vol. 3, (ed. John Philip Cooper), has on p. 20 a graph of "relative illuminance", showing that on the floor of an evergreen oak forest, illuminance was only some 5% of full sunlight, while in the ten meters above the floor of a tropical rain forest, illuminance never exceeded 2% of full sunlight, only reaching 5% at some 20 meters; at the forest floor it was more like 0.5%. (He also provided absolute numbers for PAR, or "photosynthetically active radiation": 5–50 "cal cm^{-2} d^{-1}", which I suppose is cal/cm²/day; these non-SI units, sadly, are ambiguous, as there are two separate units of energy known as the calorie, which differ by three orders of magnitude). *Tarsiers: past, present, and future*, (ed. Wright, Simons, and Gursky) cites "Grubb and Whitmore, 1967" for the observation (p. 39):

> At the forest floor, rain forests have low relative illuminance of visible light, often less than 1%.

Frances Baines, in the Daintree ancient tropical lowland rainforest in Queensland, measured 2klux in the shade, where she was examining ultraviolet illuminance on reptiles, compared to 90klux "in the sunlit area". She notes, however:

> It is interesting to note the wide range of temperatures, light levels and UVB light levels available on the forest floor – all within feet of each other, and constantly changing as the sunlight moves through the canopy above.

This is consistent with some 2% average illuminance.

So, plants adapted to life in forests below the canopy could be gardened productively with minimal amounts of light, if its spectral composition isn't too far off. Every square meter of light captured at

the surface could be distributed to feed 50 or 100 square meters of underground gardens simulating a rain-forest understory, with its ferns, mosses, shrubs, insects, and reptiles.

You'll probably need substantial ultraviolet, though, so soda-lime glass and water won't cut it as light-pipe materials.

Exhausting CO_2

People and fires produce carbon dioxide, which acidifies your blood, makes your lungs feel like you're suffocating, and would eventually replace all the oxygen in your air and suffocate you. But for plants, CO_2 is an essential nutrient, which they struggle mightily to extract from the air; normal air is only 0.039% CO_2 (up from 0.032% CO_2 in 1960). If you feed them air with 0.1% CO_2, they can grow 50% faster and produce 12% higher yields. You can deal with sustained breathing of air with 1% or 2% CO_2, but normally you want to try to keep it to 0.2% or so. Your own exhalation is about 5% CO_2, totaling about 1kg/day (12mg/s); 1% CO_2 is enough to kill some insect pests.

Also, in some places, CO_2 will seep out of the ground, and you need to ventilate spaces fast enough to keep it from building up. This is particularly tough underground, because CO_2's molecular mass of 44 is much higher than the molecular masses of nitrogen and oxygen (30 and 32). It's so much denser that you aren't going to be able to do it with convection at any kind of safe temperature. You have to do it with chemicals or air pumps.

Compost also produces CO_2.

So you'd ideally want to route a lot of CO_2-rich air to surface greenhouses, both to reduce the CO_2 emissions of the arcology and to promote the productivity of its crops, ultimately using nearly all of the produced CO_2 (average 1.4g/s across all 125 people, not counting sources other than respiration) into the biomass in the greenhouse. But that turns out not to be possible, as explained below.

The numbers above suggest that you need to change the air in a room after you've breathed at most a twenty-fifth of it. If the room is 3m×3m×6m, it contains about 54m² and 54kg of air, and it needs to be replaced every time you emit 0.2% × 54kg = 108g CO_2, which at 12mg/s, is 150 minutes — an air change every two and a half hours. However, as shown below, other considerations require an air change several times more often than that!

Nuclear submarines and space stations sometimes have "scrubbers" consisting of metal oxides (CaO, LiO) which absorb CO_2 from the air, producing carbonates; and houseplants can also absorb CO_2. These might be useful for resiliency purposes, but they probably aren't necessary for day-to-day living.

Clean air

It's necessary to replace the air in a room several times an hour to keep it habitable, due to accumulation of other things than CO_2: humidity, bad smells, dust, and heat. A list of uncertain provenance gives numbers that range from two air changes per hour (that is, 2m³ of air per m³ of room volume per hour) for a warehouse, up to 15–60 for kitchens, or 20–30 for spaces like bars, taverns, clubhouses, and repair garages.

(Dust, aside from being potentially allergenic, can also be explosive.)

This suggests that, for luxury, you need at least 10 air changes per hour in the rooms where you are, and more if there's a crowd or things are on fire; and at least 4 air changes per hour in the rooms where you aren't, which will be the vast majority of the arcology and therefore dominates the total ventilation need.

If you figure that all of this air needs to come from the outdoors, which seems like a conservative but perhaps not unreasonable assumption, that's the full 3200m³/人 of air, every 15 minutes (≈1mHz), or roughly 3.2m³/人/s, or 400m³/s across the whole arcology. In the non-SI units usually used for HVAC work in the US, that's 850 000 cfm, which is a fucking hell of a lot of air, all of which has to go through highly-efficient heat exchangers to prevent excessive heat loss. That's not hard to do, since you can easily lay hundreds of meters of ducts, but it's something to keep in mind.

The Passive House guidelines require not exceeding 30m³/人/hour, which means 0.008m³/人/s, "to avoid overly dry air", because "humidifying the air within the ventilation system is to be avoided for reasons of hygiene". You're obviously going to be violating the crap out of both of these recommendations, along with most of the rest of the Passivhaus program, because although it's awesome, it's designed for ordinary buildings, not underground cities.

Changing the air so often means your breath will be useless for feeding plants unless you concentrate it further, but whatever.

Ventilation power

A small US$80 six-inch fan does 400CFM, which is to say, 0.2m³/s. You need sixteen thousand times that, which would, linearly extrapolating, cost US$1.3 million of fans.

More digging around suggests that you can order custom-built 3.5PSIG 50 000CFM blowers, of which you'd need 17, but if you have to ask, you can't afford one. At 3.5 PSIG, which is to say 24 kPa difference input to output, each of these would be outputting 570 kW, for a total of 10MW, or 77kW/人 — way outside our energy budget, even before factoring in motor losses!

That means that if you really want to do 400m³/s, you need to do it at more like 1kPa, not 24 kPa. A table of duct sizes suggests that for 50 000 cfm (24m³/s), you should use a 41" (104cm) round duct, which will have a frictional loss of 0.67 inches water (160 Pa) per 100' (30.5 m).

If you used 17 such ducts for 40 meters of input and 17 more for an equivalent distance of output (a combined total of 104cm * π * 80m = 4400m² of galvanized sheet metal), then you'd have 426 Pa of pressure dropped in those ducts, for "only" 171kW. That gets us down to just over 1kW per person, which is manageable but still seems high. Despite the chances of infiltration by movie villains, it seems like it might be worthwhile to use still larger ducts in order to further reduce the energy usage, noise, and surface winds; and perhaps very wide, flat ones, so you can make them work effectively as heat exchangers. (They have to be wide way out of proportion to their flatness.)

XXX figure out ducting and heat exchanger heat loss

Filtering

It may also be necessary to filter air coming in from the outside to remove smoke, volcanic ash, carbon monoxide, ozone, or other

pollutants.

XXX

Heating

XXX

Cooling

XXX

Mold

XXX

Garbage

The traditional approach to garbage in big apartment buildings was the garbage chute: a nearly-vertical shaft with a door on each level into which you would dump your garbage bags, emptying into a bin in the basement. This seems to have gone out of style; XXX I don't know why.

The approach taken in the apartment buildings I've lived in in Buenos Aires in the last few years is, instead, either for each inhabitant to carry their garbage separately to the curb for pickup, or (especially in larger buildings) to have a garbage can on each level, which the building caretaker empties each day, typically dirtying up the freight elevator.

Garbage in general is a pretty big problem for would-be self-sufficient communities; simply stated, a group that generates garbage is not really self-sufficient. The universe has no garbage, only recycling. But full recycling in a modern society is quite difficult, and probably requires a group much bigger than 125 people.

Nevertheless, we can make substantial improvements over the typical state of modern urban life, within the scale of this single modest building.

A 2008 New York State study found the following composition of the garbage of the residential and commercial/institutional sector:

paper	33%
glass	4%
plastics	14%
metal	7%
"organics" (food?)	23%
textiles	5%
wood	3%
other	11%

The total is 18.3 million tons; if the state contained 19 million people, that's one ton per person per year, or 28mg/s/人. With that in mind:

		mg/s/人
paper	33%	9
glass	4%	1.1
plastics	14%	3.9
metal	7%	1.9
"organics" (food?)	23%	6.4
textiles	5%	1.4
wood	3%	0.8

| other | 11% | 3.0 |

Of these, paper, wood, textiles, and "organics" can all be composted for fertilizer, a process which dramatically reduces its mass over the course of some six months to two years. Plastics can be recycled into other plastic (especially if PET), melted into structural material, or burned for energy and CO_2 for the greenhouses. Metal and glass can be profitably sold outside the arcology for scrap, or recycled within. This leaves only "other", much of which could also be used for structural material; but if not, we're left with 3mg/s/人 of dumpit, 375mg/s in total, or almost 12 tonnes per year for the arcology as a whole, perhaps 6m³.

Since this "other" contains nothing that could plausibly rot, and hopefully nothing that's chemically unstable, it doesn't need to be taken out immediately. You could quite reasonably take out the arcology's garbage once a month instead of once a day, to keep it from piling up; or if you decided you had to let it pile up, you would have enough space to pile it up inside the arcology for 68 millennia. Given the likely future strategic value of material resources in garbage, this is probably a good idea.

In my book, that essentially removes garbage as a problem for self-sufficiency, as long as composting can be made to work, which turns out to be pretty easy.

Composting garbage

From the above, we get an estimate of 64% of garbage as compostable, a total of 18mg/s/人, or 2.2g/s. Of this, some 6.4mg/s/人 is "organic", which presumably means "wet"; that probably shrinks by some 75% if you dry it out, which you probably want to do before you compost it, reducing it to 1.6mg/s/人, and reducing the total to 13mg/s/人 or 1.6g/s.

This compost, if managed as "hot" compost and watered and turned properly, takes some 1–3 months before it's ready for use as fertilizer. If you take the high end of that range, 3 months, you can see that you have some 13 tons of compost (say, 13 m³), plus water, constantly fermenting. This is an eminently, almost trivially manageable quantity, and it will diminish further once I get to discussing sewage, since much of this compost will be diverted to sewage treatment.

This is, however, a size of compost heap that will benefit from a mechanical compost tumbler, regular temperature and humidity checks, and XXX

Food

As I mentioned before, you, individually, need to eat 100–120W or 5–6mg/s of carbohydrate and protein to survive.

Soy: not enough of a miracle, needs 1000m²/人

Soy, one of the highest-yielding crops, yields on average about 45 bushels/acre/year (12 picometers per second), and each bushel yields 11 pounds (5kg, 141 g/l) of oil and 48 pounds (22 kg, 620 g/l) of meal, which is about 80% carbohydrate and protein. 620 g/l * 12 pm/s * 80% = 6.1 micrograms/m²/s, which means you need about 1000m² of soy to feed you, a quarter of an acre. Not much!

Well, that's unfortunate, because you only have 80m², and that has

to include your solar panels and skylights too. You're too low by a factor of about 12.

Sugarcane

Sugarcane is one of the most energy-efficient crops. Experimental small plots in Brazil, says WP, have yielded 250 tonnes/hectare/year (25 kg/m²/year, or 790 micrograms/m²/s). Suppose you can raise it in greenhouses with enriched CO_2 and fertilization and everything; how much can it feed you? Suppose, like Louisiana's farmers in 2011, you get 230 pounds of raw sugar per ton of cane, 11.5%, and suppose that's basically 100% carbohydrate. That's 91 micrograms/m^2/s, which is 15 times better than soy! You're saved! Barely!

But, well, you do need some protein. And 25 kg/m²/year is way beyond normal; normal is 7. Those Louisiana farmers were delighted to get 9.6. 25 was the NPK-fertilized trial in Brazil reported in Bogdan's 1977 *Tropical pasture and fodder plants*, and it's anyone's guess whether you can really reproduce that.

How can it be that 80m² provides 10kW of XXX oh, of energy, not of electricity.

Fungus

You may be able to supplement your diet with fungus. Quorn, for example, is made from the hyphae of the *Fusarium venenatum* soil mold, processed to remove RNA and DNA.

Ingress and Egress

Somewhat to my surprise, it appears that it's actually possible for an arcology to comply with, say, the Pennsylvania fire code, because the fire code doesn't actually require you to be able to get out of a building in a reasonable time; it just requires you to be able to get to the other side of a firewall (an "exit") in a reasonable time, and from there to outside the building (an "exit discharge").

The fire code's guidelines on number and width of doors seem reasonable:

- at least two exits (four would be better);
- at least 28 inches (71cm) width for each exit; probably 2m is better.
- "exits shall be so arranged that the total length of travel from any point to reach an exit will not exceed 150 feet (45 m)", or 200 feet (60 m) if sprinklers installed.
- "The capacity in number of persons per unit of width for approved components of means of egress shall be 60 persons. Buildings protected by automatic sprinkler systems shall be allowed 90 persons per unit of width;" here a unit of width is basically the 71cm mentioned earlier. For 125 people, then, two exits is almost sufficient, and two double-wide exits is quite sufficient.
- maximum slope of an exit ramp: 8.33%. Unfortunately this would demand that you run almost 500m to get up and out of the pit arcology using the ramp.

The fire-code occupancy guidelines for housing occupancy is to assume at least one occupant per 125 square feet (11.6m²) of bedroom. If each person did actually dedicate a quarter of their 800m² to bedrooms, you'd end up with 200m² of bedroom per person, with the rather absurd result that you'd need to build means of egress for some 17 times the sustainable occupant load of the arcology — almost 2200

people. That means you'd need 36 "units of width" on the exit stairways. The "units of width" measure is a little bit nonlinear, and it turns out that, with four exit stairways, you could manage it if each one were 4.6 meters wide. Four stairways of 4.6 meters wide, by themselves, should be no problem. However, should someone decide to set up a classroom or dining area or something in one of their rooms — something with fewer square feet per person — they could easily exceed the 2200-people "occupancy" limit!

Certainly, exits sufficient for emergency use by thousands of people should be adequate for regular use by 125 people.

Moving-sidewalk ramps carrying you rapidly up from the lower stories to near the surface, in the case of the below-grade arcology, would seem to be a good idea.

Vermin

XXX

Poisonous Gases

XXX

Sewage

XXX

Electricity

XXX

Internet Access

XXX

Individual Climate Control

XXX

Water

A bacteriostatic water tank at the top of the arcology, filled with chlorinated water, can provide on-demand pressurized clean cold-water supplies to all residents.

Where do you get the water? Well, maybe you can use a municipal water supply, but that's not very self-sufficient. You can use a well, if you're in a zone where the groundwater is safe and abundant. Or you can harvest rainwater or moisture from the air.

As mentioned later, the usual Burning Man figure is that you need about 8 ℓ/人/day, or 93 microliters/人/s. Spread over 80m²/人, that's 1.16 nm/s of rain you need, or in the non-SI units typically used by meteorologists, 37 mm/year or 1.43 inches/year. There *are* parts of the world that get less than that: much of the Sahara and the Atacama, and small parts of Arabia, the Australian Outback, Nevada, Greenland, and Siberia, and presumably Antarctica; but *most* of the Sahara, of Arabia, of the Gobi, and so on, get more rain than that.

But suppose you don't? Suppose you're in the middle of the Sahara? You have to condense water out of the air, either using a thermal store (radiate away heat at night and store the cold in a water tank, then use the cold water to condense more water on the surface of a pipe, warming up the cold water until the next night) or, in the worst case, using an electrical heat pump, aka an air conditioner. The electrical heat pump needs to be able to lower the temperature of the air to its dewpoint, at which point its coefficient of performance is probably closer to 1 than the usual 2 — it's burning 1 J for every J of

heat that it manages to pump away from its cold side. [Condensing water takes] about 2.26 MJ/ℓ; at 93μℓ/人/s, that's 210W/人, an easily payable energy cost.

So you can totally build a viable arcology in the middle of the Sahara, just as long as the Tuaregs are cool with it.

Hot Water

XXX

Compressed Air

XXX

Central Vacuum

XXX

Laundry

XXX

Stored supplies for resiliency

It seems inevitable that some of the people attracted to such a community will be interested in resiliency against external shocks, such as terrorist attacks, economic collapse, or military occupation. So it might be worthwhile to investigate how much storage space you need for emergency supplies.

A year's worth of water: 2.9m³/人, 360m³ total

Of all the groups who store up for emergencies, the Mormons are the most extreme; they are recommended to keep three months' worth of food on hand, and normally a year's worth. Of all the supplies you need, oxygen is the bulkiest, but for the time being, let's exclude the emergencies that require you to use stored oxygen! Water is next. How much water do you need?

At Burning Man, in the desert, you needed about 8 liters of water per person per day. A year's worth, then, is 2.9m³/人, which is an almost insignificant fraction of the 3200m³/人 we're talking about. You could probably get by with less, since arcology air isn't desert air, but why bother?

If you had a centralized water tank with storage for the whole arcology, you'd need only 360m³, which is about 7 m in diameter. While this would provide great resiliency against external supply disruptions, it would only provide limited resiliency against internal prisoner's-dilemma problems; see "Community issues" below. A centralized water tank can also provide gravity-feed water pressure for the majority of the arcology, as described above under "Utilities".

Community issues

As anyone who's attended a condo association meeting knows, cooperating with your neighbors can be by far the hardest part of living in a community. And the more common resources you share, and the less trusting your neighbors are, the harder this is.

So it seems that the problem of growing a community to create the arcology is likely to be at least as difficult as the physical architecture; you could say that's what sunk Arcosanti.

XXX

Topics

Gradient overlay

Kragen Javier Sitaker, 2018-04-27 (2 minutes)

Let's say some color component, say red, is a linear function of pixel position (x, y) in some region: $r_i = a_0x_i + b_0y_i + r_0$. This is an arbitrarily oriented linear gradient. Now, if we α-blend this red r_i with some other red r_j painted over it with some opacity α, we get $r_k = (1 - \alpha)r_i + \alpha r_j$. This gives us a new gradient:

$$\begin{aligned} r_k &= (1 - \alpha)(a_0x_i + b_0y_i + r_0) + \alpha(a_1x_i + b_1y_i + r_1) \\ &= ((1 - \alpha)a_0 + \alpha a_1)x_i + ((1 - \alpha)b_0 + \alpha b_1)y_i + (1 - \alpha)r_0 + \alpha r_1. \end{aligned}$$

That is, the three components of the gradient (r, a, b) blend according to the alpha-blending equation just as if they were colors. Except really they are nine components; in some sense a linear gradient is a transformation matrix from (x, y, 1) space into (r, g, b) space. These matrices then react to alpha-blending in the same way that individual color vectors do: $(1 - \alpha)M + \alpha N$, at least if this α is constant and not a gradient as well.

This means that if we have several layers, each of which is divided into disjoint regions, each painted with a linear gradient with a constant alpha, we can compute the intersection regions of the single-layer regions in different layers, and compute the resulting gradient for each of those intersection regions, then render the resulting gradient.

For things like drop shadows, it would be nice to have gradients of alpha, as well, so that things can be gradually feathered into complete transparency. Linear alpha multiplied by a linearly varying color gives us a quadratic gradient.

Topics

- Graphics (p. 3483) (91 notes)
- Gradients (p. 3481) (8 notes)

Saturation detector

Kragen Javier Sitaker, 2013-05-17 (3 minutes)

Ferromagnetic materials have very high magnetic permeability, but saturate at some point; in some cases, especially for extremely ferromagnetic materials like the electrical steel used in transformers, the transition is quite abrupt.

A major problem in the historical development of radio was the "detector": some way of converting the high-frequency AC signal of the detected radio wave into a DC or low-frequency signal that could be used, for example, to activate a solenoid. This was eventually solved by the development of the vacuum-tube diode and later the semiconductor junction diode, but before that, there were a number of Rube Goldberg contraptions, some of which remained in use for a long time in special circumstances: the "coherer", which sintered metal particles together with the RF energy and then measured the DC resistance of the result; the "cat's-whisker detector", a delicate Schottky diode made with a point contact between a finely-pointed wire and a crystal of a semiconductor such as galena, iron pyrite, carborundum, or even the iron oxide on a razor blade of a "foxhole radio"; Marconi's "magnetic detector", which used the nonlinear hysteresis behavior of moving iron wire to convert an RF magnetic field into a tiny DC voltage; and Fessenden's "electrolytic detector", which used the electrolytic formation of a layer of bubbles on a fine platinum wire electrode to preferentially impede current in one direction. Somewhat related is the "mercury-vapor rectifier", which uses the enormous difference in work function between mercury and graphite to conduct in only one direction.

It occurs to me that the saturation transition in low-hysteresis electrical steel could be used to form a detector for frequencies up to some limit, as follows. You bias the primary winding of an iron-core transformer almost to saturation with DC, then superimpose the AC signal on it. The part of the AC signal opposing the DC bias will dip into the high-permeability region and will therefore see strong inductive effects --- a high impedance, either inductive (in the case of an open-circuit secondary winding) or resistive (in the case of a dummy load connected across the secondary). The part of the AC signal in concert with the DC bias will experience much smaller inductive effects, perhaps two orders of magnitude less.

This should give you an entirely-solid-state "detector" that works without any semiconductors or vacuums.

I think this device is limited in frequency only by hysteresis and eddy-current losses, which increase linearly with frequency. In this application, though, much larger losses are acceptable than in the usual applications of transformers: a 1%-efficient detector is still usable. Wikipedia tells me that laminated-steel transformers with especially thin laminations are still sometimes used at 10kHz, so I am guessing that this detector should work usably with a steel core up to some 100kHz, and with ferrite or powdered iron, up to 1GHz.

You should be able to substitute a permanent magnet for the DC bias, eliminating the need for a power supply.

Topics

Negative weight undirected graphs

Kragen Javier Sitaker, 2019-11-01 (8 minutes)

The classic shortest-path algorithms on graphs such as Dijkstra-Moore, Floyd-Warshall, and Bellman-Ford are commonly applied specifically to digraphs. In particular, Bellman-Ford is much slower than Dijkstra-Moore for graphs of any reasonable size, but is guaranteed to be correct on a more general class of graphs --- those including negative arc costs, but not negative cycles.

The usual reduction from weighted graphs to weighted digraphs replaces each undirected arc with a pair of antiparallel arcs of equal cost. Unfortunately, from Bellman-Ford's point of view, this transforms a negative arc cost into a negative-cost cycle. So, what's the best way to find lowest-cost paths in a weighted *undirected* graph with negative arc costs, but no negative cycles?

Manfred Weis discussed this in the Bellman-Ford context last year. He found a transformation from an undirected weighted arc into five directed arcs (three zero-weight) and two new vertices that seems to have the right properties. And Johnson's algorithm is another known approach. To my surprise, this seems to be a somewhat active research area.

I thought it should be straightforward to modify the Floyd-Warshall algorithm to handle this situation. But now that I think about it further, I'm not so sure.

Floyd-Warshall, like Dijkstra-Moore and Bellman-Ford, proceeds by successive relaxations, but those relaxations amount to adding a possible intermediate node k to the set of all possible paths. The central step in Floyd-Warshall is

$$d_{ij} \leftarrow d_{ij} \wedge d_{ik} + d_{kj}$$

where $\wedge$ is the binary minimum function. This amounts to considering whether the best known path from i to j can be improved by chaining together the best known path from i to k with the best known path from k to j; at the point that the algorithm does this update, all paths through all nodes preceding k have already been taken into account. (So doing this update for a given k and all i, j ensures that k and all the nodes before it have been taken into account.)

That is, either node k isn't an intermediate node on an optimal path from i to j, or it is; the two possibilities are handled by the two operands of $\wedge$. In some sense, the possible paths Floyd-Warshall considers between any two vertices are the powerset of all the vertices. In particular, this means that non-simple paths --- those that visit the same node more than once --- are not contemplated, except that a path is (in some sense vacuously) permitted to pass through either or both of its own endpoints.

Let's consider 2-cycles, paths from some node p to some other node q back to p, without any intermediate nodes. These can only exist in digraphs. (Cycles in either graphs or digraphs can only include the same arc once.) If our weighted digraph represents a plain weighted graph in the way described above, the two costs are equal; if they are

positive, this cycle cannot form a part of any optimal path (since you could improve the path by removing it), but if they are negative, they form a negative-cost cycle that means that there is no shortest path anywhere reachable from them. But this does not carry over to the original undirected graph: it may well have no negative-cost undirected cycles, because the single undirected arc is not a cycle.

So, either way, if we're trying to apply Floyd-Warshall to an undirected graph, we'd like to avoid considering these 2-cycles.

I thought I saw a simple way to do this, but now I'm not so sure. I thought it would be straightforward: just don't create paths from i back to i with only a single intermediate k. But now I see that paths from i back to i that run through multiple intermediate k nodes will necessarily start out as paths with only a single intermediate k.

A couple of ideas:

• When you decide to overwrite an optimum cost using nodes prior to k with an optimum cost going through k, you could compute how many arcs are on that path; say, start with an array of path lengths p_{ij} entirely filled with ones, and when you update $d_{ij} \leftarrow d_{ik} + d_{kj}$, also update $p_{ij} \leftarrow p_{ik} + p_{kj}$, which may amount to either increasing it or decreasing it. This enables you to add a special case: when considering updating a cost $d_{ij} \leftarrow d_{ik} + d_{kj}$, you can check whether either d_{ik} or d_{kj} is a length-2 cycle (i.e., the two coordinates are equal and $p_{kk} = 2$) and just not update in that case.

(There's an algorithm which stores the actual paths in rope form rather than just their lengths, but if you want the paths rather than their lengths, using ropes is no faster than the standard next-pointer modification of Floyd-Warshall for path reconstruction, and it needs more space.)

• But why bother with p? Including cycles in your candidate optimal path is never beneficial: either they have positive cost and you can eliminate them, or they have negative cost and the optimal path fails to exist. You could just decline to include cycles altogether by the simple expedient of not considering the possibilities $i = k$ and $j = k$. Then the algorithm will compute the lowest-cost *simple* path between each pair of nodes.

Well, not quite! It turns out that it can still unintentionally consider cycles in the following way. Suppose that the lowest-cost path from a to c goes through b, and the lowest-cost path from c to d also goes through b. If k = b before k = c, in the b step, we will compute a d_{ac} and a d_{cd} that each depend on going through b. Later, in the c step, one of the candidates we will consider for d_{ad} is $d_{\mathrm{ac}} + d_{\mathrm{cd}}$. If this is cheaper than $d_{\mathrm{ab}} + d_{\mathrm{bd}}$, which can happen if there's a negative-cost cycle between b and c, we may take it. The same logic shows that the approach in point 1 is also flawed.

However, I assert without proof that it does still consider all the simple paths, even if it considers some nonsimple paths as well.

Here's the implementation of Floyd-Warshall I was using, which appears to work in cases without negative-cost cycles:

```
def floydwarshall(edges):
    V = {u for u, v, w in edges} | {v for u, v, w in edges}
    inf = float('inf')
    d = {(i, j): inf for i in V for j in V}
```

```
    for u, v, w in edges:
        d[u, v] = min(d[u, v], w)

    for k in V:
        for i in V:
            for j in V:
                d[i, j] = min(d[i, j], d[i, k] + d[k, j])

    return d
```

For reproducibility and to attempt the possibility #2 above, I modified it as follows:

```
def floydwarshall(edges):
    V = {u for u, v, w in edges} | {v for u, v, w in edges}
    inf = float('inf')
    d = {(i, j): inf for i in V for j in V}

    for u, v, w in edges:
        d[u, v] = min(d[u, v], w)

    # Sorted order so that cases where the algorithm doesn't give
    # incorrect results at least get repeatable results:
    for k in sorted(V):
        for i in sorted(V - {k}):
            for j in sorted(V - {k}):
                d[i, j] = min(d[i, j], d[i, k] + d[k, j])

    return d
```

I was generating the graphs to run it on with this routine:

```
def randomgraph(n, m, o=10, p=1):
    rv = []
    existing = set()

    for i in range(m):
        u = v = None
        while u == v or (u, v) in existing:
            u, v = random.choice(n), random.choice(n)
        existing.add((u, v))
        rv.append((u, v, random.randrange(o)-p))

    return rv
```

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)

Fast gsave

Kragen Javier Sitaker, 2018-11-27 (5 minutes)

I was reading *TeX: The Program*, and in §§220 et seq. explaining *eqtb* and in particular §§268–284, I found the solution to implementing PostScript's `gsave` operator efficiently in an interpreter, a solution that's more efficient than either deep binding or the usual approach to shallow binding.

`gsave` saves the drawing parameters on a stack of drawing parameters, including not just the current path, but also things like the current clipping path, the current line dashing pattern, the current line join, the current line width, the current color, and so on, which change relatively rarely. Later, they can be restored with `grestore`. This allows code inside the `gsave`/`grestore` pair to change these parameters with impunity.

Now, the standard shallow binding approach to this problem is to allocate a new saved-graphics-context object on the saved-graphics-context stack, copy the entire current-graphics-context to it, and then continue. To restore, you copy the saved-graphics-context on top of the stack over the current-graphics-context. This means that accessing the variables in the current graphics context is fast, because they're always in the same location in memory in the current-graphics-context. (A slight variation of this accesses those variables directly on the top item on the stack, thus making access to them indexed off the stack pointer, which was slower on old machines but is pretty much just as fast nowadays.) But, with this approach, `gsave` and `grestore` are fairly slow.

The deep-binding alternative, as I understand it, is to maintain the graphics context as a stack of setting-value pairs. To access a graphics-context variable, you search down the stack until you find it. This is slow, but `gsave` and `grestore` are much faster, since they don't need to copy anything. But it's somewhat hairier than deep binding in its original Lisp context, where you're pushing some new bindings onto the stack, because `gsave` doesn't have a list of graphics parameters to override; it just arranges for later graphics parameter changes to be pushed onto the stack.

How can you implement this? Presumably you need to tag each name-value pair on the graphics parameter stack with the `gsave` level it belongs to. `gsave` increments the level and saves the stack pointer, and operators like `moveto` or `setlinejoin` first search the stack to find an entry, then check to see if its tag matches the current level. If it matches, they can just overwrite the value with the new value; if it doesn't match, they need to push the new level on the parameter stack. `grestore` then restores the stack pointer and decrements the level.

So variable access is slow, but `gsave` and `grestore` are fast, since they hardly do any work. This is not a good tradeoff because you commonly do several graphics operations per `gsave`/`grestore` pair, often very many, and those operations typically access several graphics parameters each.

But wait. What if we don't have to search the stack to find the current value? This is the TeX approach. The value of, say, glue between paragraphs (`par_skip`) is stored in `eqtb[glue_base+2]`, which turns

out to be `eqtb[1+256+256+1+2100+10+257+1+2]`, which is `eqtb[2884]`, a constant memory address. But the item stored at that address includes not just the value of the glue parameter, but also the save level it came from. You can ignore that when you're reading it, but when you're setting it, it tells you whether you need to save the old value on the stack or not. Then `grestore` (in TeX, "unsave") iterates over the save-stack items, restoring their old values, until it hits a level boundary.

This means that variable access is pretty fast, `gsave` is fast, and `grestore` only does a small amount of work proportional to the number of parameters that must be restored — comparable to the work done in overwriting them in the first place.

Unfortunately I don't know how to generalize this to a wider class of problems. It's perfect for dynamically-scoped variables and for propagating a potentially large class of top-down properties down a tree during a tree traversal. It's useful for graphics contexts, where it offers a much better alternative to copying a whole graphics context because you want to change a few things in it and a noticeably better alternative to changing a few things in it and then changing them back. But it isn't useful for, for example, Emacs buffer-local variables or efficiently cloning a heap object from a prototype.

Topics

Xor 1 to 1 hashing

Kragen Javier Sitaker, 2017-07-19 (updated 2017-08-03) (10 minutes)

Suppose you want to map, say, 32-bit words to 32-bit words in an apparently random but easily computable and guaranteed 1-to-1 way, such that you can generate an apparently random sequence by applying this mapping to subsequent values of a counter.

A very simple way to do this is to do a boolean matrix multiply by a nonsingular but otherwise random bit matrix. The roles of multiplication and addition here are taken by AND and XOR, i.e. multiplication and addition in GF(2) — in the 8×8 case, this is the MMIX MXOR instruction. Because of linearity, this will always map 0 to 0, but it can map any nonzero value to any nonzero value.

As a 4-bit example, we can take the following product:

```
0 1 0 0  b3
0 1 1 0  b2
1 0 1 0  b1
0 1 1 1  b0
```

If $b_3b_2b_1b_0$ is 0001, we get 0111; if $b_3b_2b_1b_0$ is 0010, we get 1010; if $b_3b_2b_1b_0$ is 0011, we get 0001 ⊕ 1010 = 1011; and so on. All 16 possible 4-bit bitvectors are produced in this way, but in the somewhat apparently random order 0000, 0111, 1010, 1101, 0110, 0001, 1100, 1011, 0100, 0011, 1110, 1001, 0010, 0101, 1000, 1111. It's not extremely random — for example, the low-order bit simply repeats the sequence 0, 1, while the high-order bit simply repeats the sequence 0, 0, 1, 1 — but it isn't obviously ordered either. And it has the advantage that the sequence can be accessed in an arbitrary order. Because it's all linear, the mapping can also be inverted, which may be an advantage or a disadvantage depending on the context.

Empirically, about 30% of 4×4 bitmatrices are singular, and also about 30% of 8×8 bitmatrices and of 16×16 bitmatrices. This was surprising to me; I am going to conjecture that the actual number is 1/e.

In discrete domains like GF(2), there's no such thing as an ill-conditioned matrix.

As a longer example, consider this 8×8 matrix:

```
1 0 1 0 1 0 1 0
1 1 0 0 1 1 0 1
0 1 0 0 1 1 1 0
1 1 0 1 0 0 1 0
1 1 1 1 0 0 1 0
0 1 0 1 1 1 0 0
0 1 0 1 1 0 1 1
0 1 1 1 0 1 0 0
```

(More compactly, we could say [170, 205, 78, 210, 242, 92, 91, 116] or "ªÍNÒò[t".)

If we apply this matrix to successive counter values, it generates the sequence [0, 116, 91, 47, 92, 40, 7, 115, 242, 134, 169, 221, 174, 218, 245, 129, 210, 166, 137, 253, 142, 250, 213, 161, 32, 84, 123, 15, 124, 8,

39, 83, 78, 58, 21, 97, 18, 102, 73, 61, 188, 200, 231, 147, 224, 148, 187, 207, 156, 232, 199, 179, 192, 180, 155, 239, 110, 26, 53, 65, 50, 70, 105, 29, 205, 185, 150, 226, 145, 229, 202, 190, 63, 75, 100, 16, 99, 23, 56, 76, 31, 107, 68, 48, 67, 55, 24, 108, 237, 153, 182, 194, 177, 197, 234, 158, 131, 247, 216, 172, 223, 171, 132, 240, 113, 5, 42, 94, 45, 89, 118, 2, 81, 37, 10, 126, 13, 121, 86, 34, 163, 215, 248, 140, 255, 139, 164, 208, 170, 222, 241, 133, 246, 130, 173, 217, 88, 44, 3, 119, 4, 112, 95, 43, 120, 12, 35, 87, 36, 80, 127, 11, 138, 254, 209, 165, 214, 162, 141, 249, 228, 144, 191, 203, 184, 204, 227, 151, 22, 98, 77, 57, 74, 62, 17, 101, 54, 66, 109, 25, 106, 30, 49, 69, 196, 176, 159, 235, 152, 236, 195, 183, 103, 19, 60, 72, 59, 79, 96, 20, 149, 225, 206, 186, 201, 189, 146, 230, 181, 193, 238, 154, 233, 157, 178, 198, 71, 51, 28, 104, 27, 111, 64, 52, 41, 93, 114, 6, 117, 1, 46, 90, 219, 175, 128, 244, 135, 243, 220, 168, 251, 143, 160, 212, 167, 211, 252, 136, 9, 125, 82, 38, 85, 33, 14, 122].

Shuffling

One particular use for such arbitrary, randomly-accessible permutations is shuffling — for example, generating a random but repeatable ordering of a playlist. This allows you to resume the shuffled playing at some later time while only storing your position in the playlist and, possibly, the matrix — which, in the 8×8 case, can be represented as a 64-bit number such as 12848028463340370089. (The 32-bit case instead requires a 1024-bit number.)

Only a tiny minority of all possible permutations can be generated by this method. (For example, for the 16×16 case, there are about $10^{2871^{88}}$ possible permutations of the $2^{16} - 1$ nonzero values, but this algorithm can generate only about 10^{76} of them.) Is this subset "fair" in the sense that every nonzero value is equally likely to be second in the permuted sequence?

Special cases

The identity matrix exists, of course, as does a bit-reversal matrix, and every other mere permutation; reflected-binary Gray code (RBGC) is generated by the identity matrix with an extra bit set to the right of the diagonal, and its inverse is the full upper triangular matrix with all bits on the diagonal or to its right set.

```
1 0 0 0   0 0 0 1   1 1 0 0   1 1 1 1
0 1 0 0   0 0 1 0   0 1 1 0   0 1 1 1
0 0 1 0   0 1 0 0   0 0 1 1   0 0 1 1
0 0 0 1   1 0 0 0   0 0 0 1   0 0 0 1
Identity  Reverse    Gray   Inverse Gray
```

Homomorphism for XOR

Because this mapping is linear, it's a homomorphism for XOR (addition in GF(2)), but of course not for other operations such as those in $GF(2^n)$. For example, consider the matrix given earlier:

```
1 0 1 0 1 0 1 0
1 1 0 0 1 1 0 1
0 1 0 0 1 1 1 0
1 1 0 1 0 0 1 0
1 1 1 1 0 0 1 0
```

```
0 1 0 1 1 1 0 0
0 1 0 1 1 0 1 1
0 1 1 1 0 1 0 0
```

We map 10 (0x0a, LSB in the last row) to 169 (0xa9, LSB being in the last column), 11 (0x0b) to 221 (0xdd), and 1 (10 ⊕ 11) to 116 (0x74). And 169 ⊕ 221 is exactly 116.

The inverse matrix is:

```
1 1 1 0 1 0 1 0
1 1 0 1 1 1 1 0
0 0 0 1 1 0 0 0
0 1 0 0 1 0 1 1
0 0 0 1 1 1 0 1
1 0 0 0 1 1 0 0
0 1 1 0 1 1 1 1
1 1 1 0 0 1 0 1
```

And, using this inverse matrix, we can easily do the reverse mapping; it maps 116 back to 1, 221 back to 11, and 169 back to 10.

An interesting question is whether the use of AND as the multiplication function is necessary to get this homomorphism.

Reverse-engineering the matrix from the sequence

For an N×N matrix, given the mappings for any N linearly independent values, you can reconstruct the matrix. What if you are just given M sequential output values, without knowing the corresponding inputs, only that they are sequential counter values? I think you can extract the last roughly lg(M) matrix rows but not necessarily all:

```
>>> m = bitmatrices.random_nonsingular_matrix(8)
>>> v = [bitmatrices.mxor(m, i) for i in range(20, 29)]
>>> [v[i] ^ v[i+1] for i in range(len(v)-1)]
[30, 38, 30, 64, 30, 38, 30, 141]
>>> v
[1, 31, 57, 39, 103, 121, 95, 65, 204]
>>> m
[25, 239, 222, 170, 205, 171, 56, 30]
```

The LSB row of the matrix, 30, appears as a difference between half of the adjacent pairs of output values: four of them. The XOR of the last two rows, 56 ⊕ 30 = 38, appears in two positions. There are two other differences, 64 and 141, which are respectively the XOR of the last four and the last three rows, but I think there is no way to distinguish those; and I think the other rows could be anything at all.

I'm not entirely sure because there seems to be a bit more information leaked: we can see that the last two bits of the index changed in the sequence [00, 01, 10, 11, 00, 01, 10, 11, 00], so we know that the selected subset of the first six rows of the matrix (which, as it happens, are just 170 and 171) XOR to 1 at the beginning of this subsequence. But we already know that those first six rows must span

a subspace including 1 (supposing our matrix is nonsingular) because we can see that the subspace spanned by 56 and 30 (or 38 and 30) does not include 1.

Topics

What might Diamond-Age-like phyles look like in the real 21st century?

Kragen Javier Sitaker, 2014-04-24 (9 minutes)

Neal Stephenson's *The Diamond Age* famously speculates on a future society dominated by tribal groups like the Ashanti Nation and the Neo-Victorians, nations whose members live distributed around the world rather than within a single territory. In the novel, Stephenson calls these tribal groups "phyles". What might phyles look like in the real 21st century?

Current protophyles

The case of the United States of America is interesting. The USA (population 300 million) is one of the few nation-states — perhaps the only one — which levies income tax on non-resident citizens, although currently only those with particularly high incomes. As a sort of compensation for this, a US passport makes it easy to travel to most of the world's territory, even obtaining work visas in many places; and in many cases, the US Embassy or Consulate will intervene if a US citizen encounters legal trouble or needs legal help in a foreign country. Additionally, the US military forces, consisting of some 1.5 million people, occupy parts of about 25 other countries†. In part because of these factors, some 3–6 million of the world's 200 million expatriates are US citizenso.

† The US has military deployments in some 150 countries, which is nearly all of them, but only the US and Afghanistan have over 100 000 US troops; only those countries and Germany have over 50 000; only those countries, South Korea, and Japan have over 20 000; only those countries, Italy, and Kuwait have over 10 000; only those countries and the UK have over 5000; only those countries and Australia have over 2000; and only those countries and Belgium, Spain, Turkey, and Bahrain have over 1000 US troops.

The case of the Romani is also interesting, largely by contrast. The Romani, unlike the USA, control no territory, and have controlled no territory for some 700 to 1000 years; but they are bound together not just by a language (with some two million speakers) and social customs, but also a separate, parallel justice system, the kris-romani. (The kris originated within the Vlax Romani, and has spread to some parts of the Romani diaspora influenced by the Vlax.) The Romani have suffered conflicts with much more populous indigenous peoples during their centuries living in Europe, including mass enslavement, mass killings, childhood kidnapping, mass deportation (most recently in France in 2009 and 2010), and so on, but also including day-to-day discrimination.

Perhaps their position vis-a-vis such events could be improved by better coordination in order to negotiate with national governments collectively. Imagine, for example, if France recognized the Romani as a sovereign state; the 2010 mass deportations of Romani which provoked no particular reaction from Bulgaria and Romania, to

which the Romani were deported.

The Church of Jesus Christ of Latter-Day Saints, also known as the Mormon Church, is a worldwide organization of some 14 million people, more than a third of whom live in the US. Some one-third of this number are "active" members. They tithe, in general, 10% of their income to the church, a centrally controlled hierarchy, which prepares its plans to establish a worldwide theodemocratic government after the predicted collapse of secular government in the days before the Second Coming of Christ. Members who do not tithe cannot enter the temple, even to attend weddings of family members. The church's net worth is estimated at US$30 billion. Most people born into Mormon families remain Mormon, and nearly all male Mormons spend two years proselytizing overseas in the care and 24–7 surveillance of the church, funded by their own money, an experience which strengthens their bonds to the church; at any given time, some 50 000 Mormons are missionaries in this way, at some 340 missions throughout the world. As a result of this proselytism, the church is still doubling in size every 15–20 years. In addition to the 50 000 missionaries, the church has some 100 000 other volunteers.

Among other services, church maintains a "bishop's storehouse", which provides goods to poor people in exchange for service to the church; in theory, these goods are available to non-Mormons as well.

The church's earlier attempts to assume temporal authority in the 1830s through 1858 resulted in wars with the USA and the assassination of its founders. Until 1927, the church's "endowment" ceremony, required of its missionaries and those who would wed in the church, included an oath to "pray to Almighty God to avenge the blood of the prophets upon this nation".

Dubai is a sort of inverse case: 80% of Dubai's population consists of expatriates, who do not enjoy the rights of citizenship in Dubai. The UAE in general does not recognize *jus soli* citizenship, so citizenship in Dubai is a sort of inherited élite status distinguishing a powerful indigenous upper class from a large class of people who, by birth, have diminished legal rights.

The Roman Catholic Church is, in some sense, the remnant of the Western Roman Empire. It operates a worldwide hierarchy with a variety of relationships with other sovereign states; sometimes its local leaders ("bishops" or "archbishops") are supported by local governments, chosen in part by local governments, or both, and sometimes not. It has territorial control over only a single square mile, the Vatican City.

Triads?

Freemasons?

http://nymag.com/daily/intelligencer/2013/07/bart-strike-shows-privatizations-dark-side.html talks about the upper class abandoning public transit in the Bay Area:

> In the Bay Area, many high-earners have already moved on to what amounts to a private services grid. They get to work on corporate shuttles, go out at night using Uber or Lyft cars, and use services like Seamless to bring food and other necessities to them. They've accepted the public sector's dysfunction as a given, and they've already abandoned it for most basic services.

The CEO of Avego strenuously objects in the comments to being lumped in with luxury services like taxis and Uber:

Avego allows people to carpool together and share the cost of the ride. The only thing Avego is doing is providing efficiency and reducing the cost of commuting for both rider and driver. Is this a bad thing? In fact, BART costs the average rider $0.35 per mile (that's their pricing model, if I remember correctly). It's not a particularly cheap network to begin with, although I grant you, it's a great system and San Francisco needs it and I love the good public transport San Francisco has, in general.

But you say Avego could never afford to charge less that the Public Transit system? In fact, the Avego network does (unlike the Ubers and Lyfts and others, which do cost about 5-7x higher than public transit). The cost works out to about $0.20 per mile, below BART's own rate.

It's a little naive to think that technology necessarily makes things more expensive. In Avego's case, we're dramatically lowering the cost of the commute, and, particularly in these difficult economic times, cutting waste and reducing people's cost of living while also improving their quality of life counts for a net win.

http://www.nytimes.com/2013/05/18/technology/financial-times-site-is-hacked.html?pagewanted=all&_r=3& describes the Syrian government's arm's-length relationships with computer crackers who advocate its cause, similar to what Myhrvold was talking about in his Strategic Terrorism paper.

http://www.theatlantic.com/international/archive/2013/07/if-your-government-fails-can-you-create-a-new-one-with-your-phone/278000216/ has lots of interesting points about supplementing states with ad-hoc smart mobs, such as:

But when states fail to deliver governance goods, communities increasingly will step up, digitally. This shouldn't be surprising, given how much excitement there is around the prospect that e-government will significantly improve the capacity of even rich governments to deliver services. However, what we're talking about here is about more than service delivery: it is about the capacity of communities to set rules, stick to them, and sanction the people who break the rules. A sovereign state is one that can implement and enforce policies. When states don't have these capacities, a growing number of communities use digital media to not only provide services, but to do so in a way that amounts to the implementation and enforcement of new policies.

Topics

- History (p. 3500) (71 notes)
- Politics (p. 3639) (39 notes)
- The future (p. 3746) (20 notes)
- Law (p. 3543) (2 notes)
- Mormons
- Dubai
- Courts

Is there an incremental union find algorithm?

Kragen Javier Sitaker, 2019-10-01 (8 minutes)

The union-find data structure, sometimes called a disjoint-set forest, is a well-known data structure that pops up in, for example, constructing minimum spanning trees; it efficiently supports the operations *component*(*n*) and *connect*(n_1, n_2). You are guaranteed that *component*(*n*) == *component*(*m*) if and only if there is a (possibly empty) set of past *connect* calls that create a path between *n* and *m* — if the *connect* calls create arcs in a graph, they are in the same component. Moreover, you can intersperse *component* and *connect* calls freely without losing efficiency, and the structure uses only O(*N*) space, where *N* is the number of nodes, not even the number of arcs.

However, the standard structure does not support edge *deletion*, only edge *creation*. It seems probably impossible for a structure with the same space performance to support such an operation, since its size must remain unchanged as O(*N*) despite the creation of O(N^2) edges, any of which can later be deleted. But is there a way to support edge deletion with "reasonable" performance?

The standard union-find data structure

Here's an implementation of the standard union-find algorithm I wrote last year as part of a maze generation program in Golang:

```
// Data structure for rapidly determining whether two maze cells are
// already connected.
type Unionfind map[Cell]Cell

func (u Unionfind) root(c Cell) (root Cell) {
        parent, ok := u[c]
        if !ok {
                return c
        }
        root = u.root(parent)
        u[c] = root
        return
}

func (u Unionfind) Connect(start, end Cell) {
        u[u.root(start)] = u.root(end)
}

func (u Unionfind) Connected(a, b Cell) bool {
        return u.root(a) == u.root(b)
}
```

For convenience, this implementation uses a Golang `map` to store the parent pointers; you can use a plain array of integers if you identify your nodes with integers, size the array to be big enough, reserve a distinguished integer to mean "nil", no parent, and initialize by filling

the array with "nil". (Alternatively, instead of setting root nodes' parents to a nil value, you can make them their own parents.)

Analyzing the performance of this algorithm is difficult enough that I am not going to try (though there is an extensive literature on the subject), but you can see that `root()` is O(1) whenever it is called a second time on the same node without an intervening `Connect()` call that would change its return value, because it recurses at most once. In such a case it does two `u[c]` lookups in the map, the second of which fails, and one (redundant) `u[c]` update.

For this algorithm variant ("path compression without union by rank"), CLRS gives a worst-case running time of $\Theta(n + f(1 + \log_{n} n))$ for $n - 1$ `Connect` calls and $\frac{1}{2}f$ `Connected` calls (p. 571). When $f \gg n^{2}$ this function is very nearly $n + f$; when $f \ll n$ it is $n + f \lg n$; and when $f \approx n$ it is about $n + f \ln n$. The "path compression with union by rank" variant is asymptotically faster, but it requires a great deal of extra bookkeeping information and has a larger constant factor; it uses twice as much space and, until n is fairly large, it's also slower. If f is much larger than n, it's slower even then.

A logarithmic-time union-find with edge deletion (or maybe not)

Suppose that, instead of parent pointers, we maintain, for each node, a linked list of edges, each indicating a lower-numbered node that it is directly linked to. Now `Connect()` merely has to determine which node has the higher number and add the edge to that node's edge list, but `root()` becomes infeasible except by groveling over the entire set of nodes repeatedly. To avoid this, we add the parent pointers back in, but in the form of an additional linked list of edges on each node — but these edges are edges between components, not just between nodes. These are established using essentially the same rule as before, with the additional proviso that they must point from a higher to a lower node, like the edges between nodes. So now `Connect()` must create both a between-nodes edge and a between-components edge, and unlike before, it does not delete a between-components edge.

Every edge (in the between-nodes list or the between-components list) has a list of dependent between-components edges attached to it. If that edge is removed, its dependents are also removed; and since those dependents may have further dependents, those further dependents are also removed. When a between-components edge is created, it becomes a dependent of the between-nodes edge that was created at the same time (if any), as well as all of the between-components edges that were followed for its creation. The logic of `root()` now must handle the situation of multiple "parent pointers" (that is, between-components edges); it always follows the pointer that points to the lowest-numbered node, since this (XXX I think) is guaranteed to be the most-recently-created surviving parent pointer.

XXX If a between-components edge is deleted because it depended on another between-components edge that has been deleted, but it was added together with a between-nodes edge, should it be recreated from its original between-nodes edge?

Hmm, I was thinking that I could replace the single-hop parent

pointers with a logarithmic number of parent pointers, each traveling up the parent chain by a power of 2, or roughly so, sort of like a skip list, in order to keep the space and time overhead both bounded at logarithmic. But I'm not sure any variant of this approach will work at all.

Merging union-find structures?

Suppose you have built two union-find structures on the same set of nodes by applying two some potentially large sets of *connect* operations. How can you compute the union-find structure that would have been computed from the union of those two sets of edges?

One way to do it is to determine whether the second set contains any edges that bridge components that are separate in the first set — which is to say, whether there are any pairs of nodes that are connected in the second set but not the first. If so, you can apply additional *connect* operations to the first set to unify them.

To connect, in the first structure, all the pairs of nodes that are connected in the second set, it is adequate to connect each node to either its root or its parent (in both cases, in the second union-find structure). That is, the parent pointers provide adequate information for this task; they are a compressed summary of the original second set of edges.

This merge operation suggests that we could build a binary tree of union-find structures in which, at the leaves, we have individual edges (that is, union-find structures containing only a single edge). Then, to remove an edge, we null out its leaf and propagate the merge operations back up to the root.

Unfortunately, although we need only walk through O(lg E) nodes, we are doing O(N) work at each node. It would be faster to just recompute the union-find from scratch, unless there are many more edges than nodes.

Topics

Golomb-coding operands as belt offsets likely won't increase code density much

Kragen Javier Sitaker, 2017-06-15 (updated 2017-06-20) (6 minutes)

Suppose we want a really compact format for representing computer programs. Historically speaking, stack-based representations tend to be the most compact, even though not all operands can be implicit; typically you have a ratio of about one stack manipulation to one computational operation. I thought maybe we could do better, with a construction like the Mill's Belt, and so I did the quick rough analysis below to see what gains were likely; it turns out that you can roughly match 5-bit stack code with a variable-bit-length operand coding, but not clearly beat it.

The idea is that instead of referring to input values implicitly by their position on the stack, you refer to them by how long ago they were produced. At a given point in code with no control-flow join points (jump destinations that can be reached by more than one jump or by non-jumping control flow) each non-constant value was either provided as an input at entry, produced at a single lag into the past. Consider, for example, this piece of assembly code, which copies a byte from stdin to stdout under Linux:

```
_start: mov $__NR_read, %eax       # literal constant  #0
        mov $stdin, %ebx           # literal constant  #1
        mov %esp, %ecx             # use input #-1
        mov $1, %edx               # literal constant  #2
        int $0x80                  # operation: system call with four args #3
                                   # inputs are #0, #1, #-1, and #2
                                   #            (.-3, .-2, .-4, and .-1)
        test %eax, %eax            # result of operation #5 (.-1) #4
        jz exit                    # jump conditional on result of #4 (.-1)
        mov $__NR_write, %eax      # literal constant  #5
        mov $stdout, %ebx          # #6
        mov %esp, %ecx             # input #-1
        mov $1, %edx               # #7

        int $0x80                  # #8 (using #5 (.-3), #6 (.-2), #-1 (.-9), #7
o (.-1))
```

Here we have ten references to non-constant values that are inputs to four subsequent operations (two system calls, a comparison, and a conditional jump). Some of the non-constant values are the result of a previous operation; others are references to the value of %esp at the entry to this code, some time in the past; and others are references to previously introduced constant values. The offsets into the past at which these references occur are distributed as follows:

```
.-1 4
.-2 2
.-3 2
```

```
.-4 1
.-9 1
```

Here's another example, a complete subroutine on amd64:

```
   20 0000 53            pushq   %rbx            # (subroutine prologue)

   24 0001 488B1F        movq    (%rdi), %rbx    # memory fetch from first input #-o
o1 (.-1)    #0

   25 0004 31C0          xorl    %eax, %eax      # literal constant 0              o
o          #1

   26 0006 48C1EB20      shrq    $32, %rbx       # >> with literal constant and #0 o
o(.-2)      #2

   27 000a 83C301        addl    $1, %ebx        # + with literal constant and #2 (o
o.-1)       #3

   30 000d 0FB6CB        movzbl  %bl, %ecx       # zero-extend #3 (.-1)            o
o          #4

   31 0010 4839D1        cmpq    %rdx, %rcx      # compare #4 (.-1) to third input o
o#-3 (.-8) #5

   32 0013 760B          jbe .L7             # conditional jump on comparison #5 (.o
o-1)
   34 0015 5B            popq    %rbx            # (subroutine epilogue)
   38 0016 C3            ret
   40 0017 660F1F84      .p2align 4,,10          # (no-op)
   40      00000000
   40      00
   42                .L7:
   44 0020 4889F0        movq    %rsi, %rax      # (register manipulation)

   49 0023 488D14CD      leaq    0(,%rcx,8), %rdx  # × with literal constant 8 ando
o #4 (.-2) #6
   49      00000000
   51 002b 4889FE        movq    %rdi, %rsi      # use first input #-1 (.-8)
   53 002e 4889C7        movq    %rax, %rdi      # use second input #-2 (.-9)

   55 0031 E8000000      call    memcpy          # invoke 3-input op; result unusedo
o          #7
   55      00

   60 0036 0FB6C3        movzbl  %bl, %eax       # zero-extend #3 (.-5) again (repro
oduce #4) #8

   62 0039 5B            popq    %rbx            # (subroutine epilogue, returning o
olast value)
   65 003a C3            ret
```

Here we see the following distribution of offsets into the past for value references:

```
.-1 5
.-2 2
.-5 1
.-8 1
.-9 1
```

If we sum both tables, we get this:

```
.-1 9
.-2 4
.-3 2
.-4 1
.-5 1
.-8 1
.-9 2
total 20
```

So, what probability distribution are these spans drawn from?

This gives a first impression that the probability of a reference going N steps back declines exponentially in N, which is to say, the span is geometrically distributed. The optimal encoding for geometrically encoded data is Golomb coding, which reduces to unary coding in the case where the exponential parameter is 2, which is pretty much what it is here.

If we encode those 20 spans in unary, they consume (+ 9 (* 2 4) (* 3 2) 4 5 8 9 9) = 58 bits, an average of 2.9 bits per operand; this is pretty close to the same density we would get on MuP21 or GreenArrays stack code with its 5-bit operations. This leads me to believe that 5-bit stack operations are already quite close to the optimal density for operand encoding.

(If we assume that the real distribution is exactly the one the encoding is optimized for, we would get exactly 2 bits per operand on average. But it's probably not.)

However, bytecode normally uses a whole byte per operation. To get such slightly-under-3-bits-per-operand density in stack code, you definitely need your stack manipulation operations to occupy less than 8 bits, because that will get you about 4 bits per operand.

(You would have to elaborate the model a bit to handle merging control-flow, such as at the tops of loops and after conditionals, and to handle functions, and I don't think this will be difficult, but the likely gains are not large enough to motivate me.)

A possible way to rescue this: suppose that for two-operand instructions, one of the operands is *always* implicitly the previous result, and you use a separate one-operand "belt manipulation" copy instruction in the rare cases where that isn't what you want. If these copy instructions are uncommon enough, this could get your average well below the 5.8 bits of operands per computational instruction implied by the above, maybe down to 3 or below.

Topics

Isotropic nonlinear texture effects for letterforms from a scale-space representation

Kragen Javier Sitaker, 2019-09-10 (16 minutes)

I was walking by some of those goofy hipster chalkboard signs in my *cheto* neighborhood today and noticed that a lot of them are produced from a bunch of relatively simple transformations of letterforms: outlining, drop-shadowing, color gradients, and so on. It occurred to me that it's probably feasible to mechanically explore restricted but visually interesting sets of such transformations in the form of DAGs of a simple algebra.

In many cases, it would be ideal for the transformation to be shift-invariant (s-i), rotation-invariant (i), resolution-invariant (r-i), and bounded-amplitude (b) — that is, transforming a shifted, rotated, or resampled image should produce the shifted, rotated, or resampled transformation of the original image, and that it should be possible to compute bounds on the brightness of the pixels in the transformed image, at least given bounds on the brightness of the images in the original image. Moreover, if you want to do image approximation for style transfer (see Image approximation (p. 2394)) it might be helpful for the transformation to be differentiable with respect to some set of parameters. These restrictions reduce the space of possible transformations in a way that should dramatically accelerate stochastic exploration and mathematical optimization.

Note that it is quite explicitly *not* a goal to restrict ourselves to *linear* transformations.

This is quite similar to, and inspired by, the abundance of excellent work in recent years on computer vision using artificial neural networks.

(See also An algebra of textures for interactive composition (p. 1283) and Cheap textures (p. 736).)

Colors

In the below, I generally speak of "images" as two-dimensional regularly sampled grids of scalar numbers. This is most apt to grayscale images; the simplest way to incorporate color images is to treat them as three separate images, one each for red, green, and blue.

The basics: isotropic scale-space representation

Given a sampled image, you can convolve it with some linear filter to get a transformed image; this transformation is shift-invariant (s-i) from the definition of convolution. If the filter's impulse response is isotropic, which is to say rotation-invariant (i), this transformation will be isotropic. If the linear filter f_s has a scale parameter s such that f_{ks} gives the same linear filter resampled to a new sampling grid k times larger, then you can satisfy the resolution-invariance criterion. You can derive reasonable amplitude bounds on the resulting image

from amplitude bounds on the original image and on the filter's impulse response.

A particularly appealing filter is the Gaussian, the scaled function e^{-r^2}, both because it's the only isotropic separable filter and because it can be very inexpensively approximated using CIC or Hogenauer filters, called repeated box blurs in image processing — independent of the radius, a quadratic approximation to the Gaussian (up to a constant scale factor) takes only six subtractions per pixel, given a dimension-indepenent third-order two-dimensional prefix sum (aka summed-area table) of the image, which requires six additions per pixel to compute, and in general about four times as much space as the original image.

(An additional desirable property is that Gaussian convolution is closed under composition: the composition of two Gaussian convolutions is a third Gaussian convolution whose scale parameter is simply the sum of the scale parameters of the two.)

This is the standard scale-space representation used in machine vision since the 1960s, the two-dimensional analogue of the one-dimensional Weierstrass transform used in analysis (and in particular function approximation) since the 1800s.

So you can use Gaussian convolutions with arbitrary constant radii as elementary operations in your algebra of transformations without risk of producing an inefficient, anisotropic (¬i), shift-variant (¬s-i), resolution-dependent (¬r-i), or unbounded (¬b) transformation. Moreover the transformation is differentiable with respect to both the input image and the scale parameter.

Another class of scalable isotropic filters that admit especially efficient implementations of convolution are flat† circles — circular boxcar filters. These are less efficient than Gaussian convolution, especially as kernel sizes grow, but, as discussed in Real-time bokeh algorithms, and other convolution tricks (p. 2661), they still admit much more efficient implementations than are generally known in the literature, on the order of 1–3 additions and subtractions per scan line in the kernel, or less if polygonal approximations are used.

Flat circle convolution is i and r-i except for aliasing artifacts, b, and s-i. It's differentiable with respect to the input image, but it's imperfectly differentiable with respect to the scale parameter; the flatness constraint requires the circle to expand by discrete pixels, which create discontinuities in its derivative. I feel like you should be able to make a differentiable version by lerping between circles of adjacent radii; using the above-linked algorithms, such a filter can be implemented at much less than twice the cost of a single flat circle convolution, since the two circles will be equal on most scan lines.

† "Flat" in the sense that all points within the support of the filter have the same "height".

Nonlinear morphological operators

Another class of shift-invariant, resolution-invariant, and bounded-amplitude transformations on images with efficient interpretations are the nonlinear morphological operations of erosion ⊖ and dilation ⊕; if used with an isotropic "structuring element" or kernel, such as a flat circle, these operations are isotropic. These, too, have been used in machine vision since the 1970s. As described in Some notes on morphology, including improvements on Urbach and

Wilkinson's erosion/dilation algorithm (p. 216), Urbach and Wilkinson published an algorithm that can evaluate these operators with flat kernels at only slightly higher computational cost than the linear convolution with flat circular kernels described in the previous section, and it's straightforward to shave off another factor of 2 or 3 from Urbach and Wilkinson's algorithm in most cases.

So you can use erosion and dilation with flat circles with arbitrary constant radii as additional elementary operations; as with the isotropic convolution cases, this poses no risk of producing an inefficient, anisotropic, shift-variant, scale-variant, or unbounded transformation.

Combining operators

Arithmetic pixelwise combination

Given constant images and the above elementary unary operations on images, we can combine them pixelwise using most of the standard mathematical operators: +, -, ×, ∧ (minimum), ∨ (maximum), and in some cases ÷, if appropriate bounds can be shown to hold for the input images. (But not %.) In particular, note that pixelwise-nonlinear ×, ∧, and ∨ permit the realization of nonlinear image filtering operations even without the use of any nonlinear neighborhood operations such as the morphological operations.

Moreover, you can apply a number of unary operations pixelwise, such as exp, sin, cos, atan, and in some cases ln.

Bounds-preserving operators

However, I think the above set of combining operators is somewhat suboptimal with respect to its boundedness and efficiency. It's true that you can compute bounds on $a + b$ or $a \times b$ given bounds on a and b, but if you're randomly assembling DAGs in this algebra, a lot of them will randomly have very large or very small bounds. It would be desirable to have a similarly expressive set of operations in which, if both inputs are bounded to some range, the output is too.

In particular, consider the case of a, b, $c \in [0, 1]$. Then these continuous, pixelwise combining operations preserve that bound:

- $1 - a$
- $\frac{1}{2}a + \frac{1}{2}b$
- $\frac{1}{2}a - \frac{1}{2}b + \frac{1}{2}$
- $a \times b$
- $a \vee b$
- $a \wedge b$
- $a + b \times (c - a)$ (lerp)
- $\lg(1 + a)$
- $2^a - 1$
- $\frac{1}{2}\sin(\omega a + \varphi) + \frac{1}{2}$
- 0 if $a = 0$ else $a/(a + b)$
- 0 if $a = 0 \vee b = 0$ else $2/(1/a + 1/b)$ (scaled resistors in parallel; soft minimum; harmonic mean)
- $\sqrt{(a)}$
- a^k, more generally

Of these, all but ∧ and ∨ are everywhere differentiable; those may happen to be differentiable in a particular case. It might be possible to

use the harmonic mean above or $\sqrt{(ab)}$, the *geometric* mean, in place of $a \wedge b$, but it's not a very good substitute.

Anisotropy

The algebra of image filtering above can express a wide range of effects, but since all of its operators are isotropic, it can't express anisotropic effects, which for better or worse include calligraphic stroke emphasis (including goofy hipster ironic Victorian puffery effects) and drop shadows. By adding just a simple shift operator to the algebra, $s_{m,n}(p) = (x, y) => p(x - m, y - n)$, we can gain a wide variety of such anisotropic transformations, while still guaranteeing b, s-i, and r-i—except for aliasing artifacts. Differentiability suffers from the same kind of aliasing artifact as the flat circle convolution, and similarly bilinear interpolation is sufficient to mostly restore differentiability. However, bilinear interpolation suffers from its own artifacts, having to do with the lousy approximation the triangle kernel is to a sinc.

To fully restore differentiability, we need a differentiable interpolation operator, such as quadratic spline interpolation, which additionally is a much better approximation of sinc. It requires nine multiply-accumulates per pixel, though perhaps I can strength-reduce that to three, or one if it is separable or close enough.

To account for both the computational cost and the anisotropy we would like to minimize, it may be a good idea to include a special penalty term for the use of the shift operator in cost functions used for mathematical optimization.

Dynamical systems

Discrete time

In addition to static graphs of image transformations, we can think about evolving an image over time. One simple way to do this is to iterate a transformation some number of times — that is, run it with its previous output as its new input. This can give rise to interesting behavior even with very simple transformations. (This is also one of the motivations for seeking combining operations with output bounds the same as input bounds.)

Continuous time

However, the number of times the transformation runs in that state-machine scheme is necessarily discrete — and such discretized time means that the result cannot be differentiable with respect to time. As a possible alternative, we can use a transformation to define an *ordinary* differential equation that specifies the temporal evolution of the image: the pixelwise difference between the original image and the transformed image gives the pixelwise time derivative of the evolution. (It can be seen that, except for numerical approximation errors, this will never lead the pixel outside the smallest interval containing both its own range and the range of the transformed pixel; so if both are [0, 1], it will remain within [0, 1].) Then we can use garden-variety Runge–Kutta numerical integration of ordinary differential equations to compute the image's evolution, time step by time step, with high precision.

This makes the resulting image differentiable with respect to the

time parameter — somewhat by fiat, as it were — and allows us to vary that parameter continuously.

Differentiable crossbars

In the above I've mostly talked about the DAG of image-processing operators as if it descended from heaven. But there occur to me three major ways to handle this:

- Expose image-processing operators directly to a user, whether through an API, a GUI, or some hybrid, and let them play around.
- Generate DAGs randomly, possibly using genetic-programming techniques, or exhaustively, possibly using some kind of heuristic breadth-first search.
- Use a fixed topology that is nevertheless flexible enough to generate a wide variety of effects.

The third approach is the one taken by most current artificial-neural-network research, usually through "fully connected layers" (complete bipartite graphs) and "max-pooling". Fully connected layers of n neurons from m inputs involve multiplying a vector of length m through an $n \times m$ matrix — for every pixel in your image, if you're doing something convolutional.

Telephone networks originally worked using fully connected switchboards: any jack could be patched to any other jack by the operator by inserting a wire into both jacks. But this scaled poorly with the size of the network, so they changed to crossbar switching.

A crossbar switch is a small permutation matrix; it might have four inputs and four outputs, although in the Telephone network all these connections were initially bidirectional. Each of the 16 crossing points could be closed (1) or open (0). In this way, any input can be connected to any output, and indeed every input can be connected to some output at once when the matrix is 75% sparse.

The trick is that multiple layers of many of these crossbar switches can produce any permutation of the inputs on the outputs. (A sorting exchange network is the special case of this where the crossbars are 2-input 2-output switches.) Consider a layer of four input 4×4 crossbars, each with one output connected to each of four output 4×4 crossbars. This is sufficient to connect any of the 16 inputs to any of the 16 outputs, but it cannot produce all possible permutations — you cannot have two inputs on the same input switch connected to two outputs on the same output switch. All circuits are busy.

Introducing an intermediate "hidden" layer of four more 4×4 crossbars, with a similar fully-connected arrangement, does, I think, give you all possible permutations. (A simple counting argument shows that it *could*, while no arrangement of eight 4×4 crossbars can — $4! = 24$, and $24^8 = 110{,}075{,}314{,}176$, while $16! = 20{,}922{,}789{,}888{,}000$, but $24^{12} = 36{,}520{,}347{,}436{,}056{,}576$; but clearly there are many ways to get the same output permutation by routing circuits through different hidden-layer crossbars.) There is some savings here: a single-stage 16×16 crossbar would contain 256 contact points, while this three-stage network contains only $12 \times 16 = 192$ contact points. Adding further crossbar-switch stages to the network increases the economy.

What I have in mind here is using 4×4 matrices as differentiable crossbar switches. Applied pixelwise to four input images, such a

crossbar produces four output images as linear pixelwise functions of the inputs, and so a few layers of such stages can substitute more economically for the fully-connected layers in traditional artificial neural networks.

The output of these stages is then fed to some set of fixed-function units attached in a fixed topology to the crossbar outputs: lerp, multiply, average, max, min, Gaussian blur, shift, and so on. In this way, the continuously-differentiable "crossbars" substitute for the discrete connection topologies we might otherwise have to search through.

We can apply an optimization penalty for weights that are far from 0 or 1, perhaps later on in the training, as is conventionally done in topological optimization. In this way we can give the crossbars an incentive to sparsify, and thus we can stop computing functions that aren't contributing to the results.

Topics

In a world with ubiquitous surveillance, what does politics look like?

Kragen Javier Sitaker, 2014-04-24 (11 minutes)

There's been a lot of discussion about how the NSA is spying on everyone in the world, which is a huge civil-rights problem. If everything you've ever written to a lover or a friend, every file you've ever downloaded, and a minute-by-minute record of your whereabouts as determined by your phone, is archived forever, someone who wants to blackmail you can probably find something in the archive to offend any particular group they want you to offend, prosecute you wherever you happen to live, and therefore make demands on you. So the NSA, or anyone with access to their archives, will have extensive influence on world politicians throughout the next half-century, even if their spying were to stop tomorrow.

Beyond the utility of such an archive as a means of coercion *per se*, it can be used to amplify more traditional means of coercion. It's common for political leaders and even rich people to employ bodyguards and keep their whereabouts private in order to frustrate kidnapping and assassination attempts. But, even if your cellphone records from your teen years don't reveal where you're likely to vacation with your family, even if your bodyguards are always at your side day and night, and even if the NSA and their Russian counterparts don't have the political backing to blow you up with a Hellfire missile or shoot you with a quadcopter-mounted rifle, the social-graph data of who you know will provide ample "soft targets" --- a tactic that has already been used, for example, in the FBI's malicious prosecution of US journalist Barrett Brown. (They raided his mother's house and filed trumped-up charges against her: "obstruction of justice," for not knowing he had his laptop at her house. She pled guilty.)

Any government agency armed with a computer-generated list of your thirty-two closest friends and family members can surely find one of them who is poorly guarded and easy to kidnap or threaten. By this means, social-graph data amplifies traditional means of coercion.

You might think that regulatory and administrative oversight of the NSA can solve this problem. I don't think it will, for two reasons.

First, the NSA is already in a strong position to retaliate against any politicians that attempt to rein it in, using blackmail.

Second, if the NSA doesn't do it, other agencies will. Current computer security is extremely poor; a smart kid in Iran can probably manage to download cellphone location records of people in the US en masse; and the relevant social-graph information is largely public on Facebook. As has been the case with bombings for over a century, our best defense is that not very many people want to do it badly enough to dedicate the necessary part of their life to the problem.

However, as with bombing, given that this is now a viable path to geopolitical power, there will be no shortage of organized groups (such as the US military and the militaries of other countries) who organize themselves to take advantage of the opportunity. Indeed, any entity seeking power through coercion who fails to take advantage of these opportunities will probably be subverted by entities that do.

There's still the matter of budget. If we stipulate that the information is readily available, how much does it cost to turn it into a Big Brother Database that enables this kind of coercion? (Let's assume the agency pays for this itself instead of using storage it borrows without permission.)

There are currently about 7 billion people. Each of them might have 1000 social connections that matter; these social connections range over several orders of magnitude of importance. If you need 16 bits to represent the importance of the relationship and 40 bits to identify the person who is related to, that's 56 trillion bits, or 7 terabytes. A one-terabyte disk currently costs about US$100, so storing the world social graph — if you could get hold of it — would cost you about US$700 of disk. You'd probably want to store it with some redundancy to handle disk failures, indices might add another factor of 2, and motherboards, power supplies, etc., might add another 25%. All in all we're talking about a budget of some US$2500 to store the world social graph, assuming you can get hold of it.

However, disk prices are still in an exponential fall, halving every 15 months. In three years, in 2016 or so, the price will be around US$600, and three years from then, US$150.

The next step, presumably, is the location information database. The Earth is about 20 million meters from pole to pole, so locating someone on Earth's surface to within one meter — good enough for targeting a bomb — requires about 50 bits of information. Let's round up to 64, 8 bytes. Collecting this information every ten minutes for a typical 30-year lifetime adds up to 12.6 megabytes per person, or 88 petabytes in all: about US$9 million of disk storage at present, shrinking to US$1 million around 2020 and US$1000 around 2035 (although the population will be somewhat higher then, so the cost will be slightly higher). But this information is highly compressible, so these cost numbers might be high by an order of magnitude or so.

However, even if you collect this much information originally, you don't have to retain it all for it to be useful. If you summarize to the three most common places in which a particular person can be found, then instead of 12.6 megabytes per person, you only need 24 bytes per person, or 170 gigabytes for the world population, about US$20.

Collating multiple sources of location information might inflate the 12.6 megabytes by a factor of three or so. For example, license-plate cameras, gait recognition from security cameras, public-transit card tracking, ticket-purchase information from airlines and long-distance bus lines, private jet flight plans, and location information on social-media postings can all provide supplementary location information. Disagreements among these sources might point to cases where someone's trying to hide something. For example, if someone's cell phone jumps from one city to another at over 200 kilometers per hour, they've probably taken a plane, and there should be a flight record unless they're traveling under an assumed name.

This kind of information also supplements the online-gathered social graph with information about who you move from place to place with (perhaps you're riding in their car) and who you lend your car to.

So what about the Blackmail Communications Database? This is more difficult. A basic version might be some ten 100-byte SMS messages per day per person, some 1000 bytes. 1000 bytes per day per person over 30 years is about 11.0 megabytes per person, similar in size to the location information database; you also need the metadata of who the message was to and when it was sent, which perhaps inflates it by 10% or 20%, so let's say 13 megabytes. If you also include IM and email, you might have another order of magnitude on top of that, or 130 megabytes. (Some people send more, some send less.) That brings us up to almost 900 petabytes for the world: US$90 million.

Any spy agency in the world with a billion-dollar budget has surely already done these calculations and has been running this program for years.

At some point in the near future, perhaps around 2020, DNA sequencing will be as inexpensive as license-plate scanning is today. Whenever you touch an object that isn't yours, such as a doorknob or hot-water tap, or release bodily fluids such as urine into a public receptacle, you'll be taking the chance that it's sampling your skin cells. Combined with the location database, this will rapidly provide a clear picture of the genetic ancestry tree of living humans, which can be added to the social-graph database to provide a more complete picture, including corrected biological paternity data. That is, you'll be able to infer whose biological father, grandfather, or great-grandfather is someone other than who it's conventionally assumed to be.

In its full form, the genome of a human being is 3 gigabasepairs, or 750 megabytes, although this is highly compressible. But you don't need 750 megabytes to uniquely identify a person; you only need a judiciously chosen 66 bits to provide a unique identifier with high probability. Reliably inferring biological relatedness probably requires more bits, but I don't think very many more.

This data, interchangeably with other biometric surveillance data such as facial recognition and gait recognition, will eliminate anonymity and pseudonymity for anyone who traverses public spaces, at least from the agency or agencies that have access to it.

This seems to point to a fairly dystopian future, one in which a friendship with, or relatedness to, a powerful person could get you kidnapped or murdered to coerce their compliance, and in which access to a relatively small and easily copied database is sufficient to provide this ability to the NSA --- and whoever manages to steal a copy of it from them. What are our options for avoiding this?

You could try to keep your part of the social graph secret: don't post it on web sites, say. But that doesn't help if the other people you know, or the people who see you together, go ahead and post their photos online.

Today there are companies whose business is to deanonymize web visitors http://www.forbes.com/sites/adamtanner/2013/07/01/heres-some-co
ompanies-who-unmask-anonymous-web-visitors-and-why-they-do
-it/ and to sell listings of sightings of a given license plate

http://www.forbes.com/sites/adamtanner/2013/07/10/data-broker-offers-new-service-showing-where-they-have-spotted-your-car/ so you can tell where a car has been. These services will get more comprehensive, cheaper, and more numerous, as improving fabrication and analysis technology progressively drives down the cost of providing them.

Topics

Caustics

Kragen Javier Sitaker, 2018-08-18 (updated 2019-11-08) (8 minutes)

A group of students at EPFL have started a company called Rayform to make objects with customized caustics (both reflection and refraction), using materials such as PMMA, aluminum, and glass to form their "caustic generators". This turns out to be a centuries-old Japanese art form known as "magic mirrors"; the originals were copper mirrors hand-scraped to produce customized caustics under the influence of copper's elasticity and a varying material thickness, legendarily so that underground Japanese Christians could escape religious persecution; the principle by which these "diaphanous mirrors" or "makkyo" worked was not understood until the 1960s, although they had been manufactured in Japan and also China for thousands of years. This prompted a couple of thoughts from me tonight. First, what about sunlight automicroscopy? Second, how about cheaper fabrication technology?

See also files Gauzy shit (p. 2985) and Caustic simulation (p. 1454).

Sunlight automicroscopy

Sunlight reflecting off a convex surface projects a magnified image of whatever colors or patterns are on the surface; this is easily seen with, for example, a Red Bull can. Sunlight is about 100 kilolux, while sunlight shadows can vary but are typically around 3–10 kilolux. So even if the reflected light is spread over a 20× larger surface than the reflective convex object, it still has brightness comparable to the ambient light. And the projected image is substantially larger than the patch on the surface that it is projected from — in the aluminum-can case, all the spread is in a single dimension, so it can be on the order of 20× larger, while the correspondingly bright reflection from a sphere would have a linear magnification factor of only 4–5 before becoming undesirably dim.

However, the caustic-shaping technique can potentially rescue the method — in the geometrical-optics approximation with point-source illumination, it can focus the light from an arbitrary area onto a point or line of zero area, thus achieving infinitely bright illumination. Rayform's demo videos seem to show focusing of more than one order of magnitude.

Point-source light reflecting from a surface whose normal varies over some angle θ will in turn vary over the angle 2θ. The sun subtends about half a degree, so the reflection from a curved surface patch will subtend about half a degree more than the curved surface, which blurs the projected image somewhat. However, this is a limit on the angular resolution of the microscopy method, not its spatial resolution. And narrowing the sunbeam by passing it through a pinhole that subtends less than half a degree from the point of view of the generator.

The nonzero angular size of the sun also provides the limit on the brightness increase available by focusing: the projected focus spot will have, at minimum, the same angular size as the sun, as viewed from

the point on the generator that is generating it.

Another limit on this technique is the diffraction limit: as the concave facet producing the focused spot becomes smaller, the produced beam has a larger divergence. I think that roughly to achieve half a degree divergence — the best you can do with a half-degree-wide sun light source — you need a facet of diameter roughly $\lambda/(\frac{1}{2}\sin(\frac{1}{2}°))$. This works out to about 126 μm for 550-nm light. This *is* a spatial resolution limit.

The facet can be a concave paraboloid section, in which case it produces a point caustic, but if it is less concave in one direction, it will spread out its light to produce a line caustic subtending some arbitrary angle at some arbitrary rotation.

Setting the spatial and angular resolution limits equal, maybe we would like 126 μm to subtend about ¼° of the curve of the convex surface, which gives us a radius of about 29 mm. Spheres or cylinders with a diameter smaller than 58 mm will have an unnecessarily coarse angular automicroscopy resolution limit, larger than ¼°, imposed by diffraction of light from their facets; those with a larger diameter will have an unnecessarily coarse spatial automicroscopy resolution limit, larger than 126 μm, imposed by the apparent size of the sun.

If we want these 126 μm facets to project pixels at about 72dpi — 350 μm, a lower limit for comfortably readable text, although older computer terminals and printers used a slightly coarser 8 vertical pixels per 6lpi line, giving 530 μm — then we want 350 μm to subtend ½° as seen from the surface of the mirror. This gives a projection distance of 40 mm, which seems rather small to me, so maybe 100 mm would be better, which gives blurry 870-μm projections. Since each facet can project an arbitrarily oriented line, rather than just a point, you only need about, say, 5 of them per letter. This means our 29-mm-radius shiny sphere with its 10500-mm² surface area, holding about 850 000 facets of 0.0124 mm² each, can project about 170 000 letters, about 40 or 50 pages' worth of text.

This may not be reasonable — the pixels may be too crowded together. Consider that if the facets are all just directly pointed away from the center, the spots they project will be ½° apart as seen from the surface — which is to say, they will all kind of blur together, unless you stack some of them on top of each other, which is of course what Rayform does.

How bright are they by default? Consider a mm² in the center of the sunbeam, which is reflecting its light onto a screen in shadow positioned a negligible angle to the side of it. This mm² subtends 34 milliradians (1.98°) and so its projection will subtend 78 mrad (4.0°), which means that at 100 mm it covers an area 6.9 mm × 6.9 mm, which is 48 mm². So it will be 48× dimmer than the direct sunlight: 2100 lux, visible on a 10 kilolux shadow background but far from overwhelming. But areas illuminated by several times this 126-micron-wide minimum will be considerably brighter than the shadow.

How much spatial precision do we need to make the surface reflect like this? Suppose we're willing to tolerate ¼λ deviations. Well, at 555 nm, that works out to 139 nm (≈ 5.5 micro-inches). This is a relative radius error of 4.8 parts per million. Regular ABMA bearing balls of grade 100 have a surface finish smoothness of 5.0 micro-inches (127 nm), but to get roundness of 5 micro-inches, you have to go

down to grade 5. That isn't even the lowest grade, but grade-5 bearing balls don't come as large as 58 mm; they only go up to 2 inches, which is 50.8 mm.

Turning a spherical surface with a radius of 29 mm into a 127-micron-wide paraboloidal facet with its focus at 100 mm requires changing the curvature radius from +29 mm to -200 mm. At +29 mm, the middle of the facet would be 69 nanometers proud of planar; at -200 mm, it would be 10 nm below. This seems like I must have some kind of calculation error, since it seems inconceivable for such a small difference to produce a precise focus.

Cheaper fabrication technology

First off, what about using sugar glass instead of PMMA or soda-lime glass for the refractive pieces? Instead of polishing it with rouge, you could polish it with water.

Second, how about using electropolishing to remove tiny, precisely controlled amounts of metal, leaving a smoothly varying surface, while leaving a mirror finish? Electroplating at around 1000 A/m² deposits chromium at something like 100 nm/minute, so it seems like thickness control down to the level of less than a monolayer should be feasible. This is also potentially useful for making large mirrors out of invar or similar, then aluminizing or silvering them.

Topics

A failed attempt to make squares cheaper to compute

Kragen Javier Sitaker, 2019-07-09 (updated 2019-07-11) (4 minutes)

Could you cheaply generate a lazily filled square table using the method of differences?

The squares are the sums of the odd naturals; $5^2 = 25 = 1 + 3 + 5 + 7 + 9$, for example. The simplest way to tabulate the sequence of squares $y = x^2$, or for that matter any polynomial function, is using the method of differences, here using three columns because the sequence is quadratic:

```
2   1   1
2   3   4
2   5   9
2   7  16
2   9  25
2  11  36
2  13  49
```

Here each number is the sum of the number to its left and the number above it, except the first row.

This is not a very appealing way to find 28^2, though.

Consider, though, if we want to tabulate $y = (3x)^2 = 9x^2$. We can tabulate this sequence with the same algorithm, but starting from a first row that is multiplied by 9:

```
18   9   9
18  27  36
18  45  81
18  63 144
18  81 225
18  99 324
18 117 441
18 135 576
18 153 729
```

This has gotten us somewhat more quickly to $27^2 = 729$, which is very close to knowing 28^2; specifically, the difference is $2 \cdot 27 + 1 = 55$, so $28^2 = 729 + 55 = 784$. We could have gotten there even faster by tabulating $y = (9x)^2 = 81x^2$ in the same way, starting with a first row multiplied by $9^2 = 81$:

```
162   81   81
162  243  324
162  405  729
```

That is, this setup allows us to leap ahead as we tabulate squares, skipping eight out of every nine items. We could leap by tens, or by hundreds, or by 27s, or by any other number. And once these squares are tabulated, we can keep them tabulated and not recalculate the ones needed to get us close to our objective.

This suggests a general procedure for finding your way to a given square with a logarithmic number of N-digit additions: first leap by the largest power of 3 less than the number, twice if necessary; then the next-largest power of 3, twice if necessary; and so on, until you're leaping by 1s. So, for example, to square 451, first find the square of 243, then 324 (leaping by 81), then 405, then 432 (leaping by 27), then 441 (leaping by 9), then 450, then 451. This involves 12 three-digit additions, which is worse than the standard partial-products approach, but only by a factor of 4. And it's the same 12 operations we'd need to find the 451st term of an arbitrary quadratic sequence, not just $y = x^2$. And maybe some of those values would be already tabulated if this isn't the first number we're squaring.

However, there's a missing piece here. When we arrived at 27 leaping by 9s, our computational state said:

```
162   405   729
```

To leap by 1s again instead of 9s, we need to somehow get from that to this:

```
  2    55   729
```

The 2 is easy — it's the same 2 on the first line of the leaping-by-1s square sequence — but where do we get the 55 from?

I'm pretty sure that the second column we need to slow down by a factor of 9 is a linear function of the previous computational state (162, 405, 729), and I think you can derive that linear function from Newton's divided-differences form for the underlying polynomial.

But I suspect it really only depends on the 405 in this case. 55 is 55 because it's $2x + 1$. 405 is 405 because it's $18x - 81$. So 55 is $2(405 + 81)/18 + 1 = 405/9 + 10$. Extending this to arbitrary quadratic functions (which potentially have a different second-order difference) might involve taking the 162 into account: it's the 2 of our original second-order difference multiplied by the square of our speed.

This is kind of shitty, though, because we were hoping to avoid multiplying a two-digit number by itself, and to get there we ended up dividing a three-digit number by 9, which is *harder*.

Topics

- Math (p. 3564) (78 notes)
- Facepalm (p. 3450) (24 notes)

A filesystem design sketch modeled on Lucene

Kragen Javier Sitaker, 2007 to 2009 (43 minutes)

IN PROGRESS:

- finish writing
- describe segment directory structure
- list awesome features

I've been thinking about using something like Lucene's index file structure to build a filesystem suitable for lots of small files, similar to Reiserfs's goals.

I've been using Reiserfs for a project recently, and I've been pretty annoyed at its lack of performance predictability for interactive use; for example, when I `find | wc -l` on a certain Reiserfs directory tree that contains 18GB in about 500 000 files (see the section "murdererfs problem" for exhaustive detail), it takes 9 minutes. So I was thinking about how to avoid this.

To some extent, lack of performance predictability is an unavoidable feature of disk-based filesystems; access to a random byte that's in the buffer cache costs maybe 100ns, while access to a random byte on a spinning-rust disk unavoidably costs somewhere around 8ms, which is 8 000 000 ns. Since the disk is (generally) much bigger than the buffer cache, an adversarial program can always choose its next requested byte from a block that's not in the buffer cache, thereby receiving terrible performance, 80 000 times worse than if it were to read data that was already in memory.

So it's not possible to design a filesystem to have consistently good performance for all possible access patterns, even for read-only access; read/write access complicates things even more. Instead we should design a filesystem that has consistently good performance for the *common* access patterns. As explained above and in the "murdererfs problem" section, reiserfs fails at this in the case of `find -print`.

This is an exploration of how to do better, and why; but first, why not.

Spinning Rust is Obsolete

Now, it's possible that Flash SSDs will make this all irrelevant, as they become large and inexpensive. I suspect that spinning-rust disks will remain relevant for a while longer, for the following reasons.

First, there's Moore's Law and its effect on pricing. For the last 15 years, disks have doubled in capacity every 15 months, while chips have doubled in capacity only every 18 months. Flash has improved its density relative to feature size quite a lot during that time (that is, bits per square lambda; lambda is now 45nm, I think) and has become economically important enough to get access to the latest fab technology and have almost zero NRE expenses per unit. However, I am guessing that from now on it won't improve any faster than process sizes do. At present 8GiB of Flash costs about US$20 (US$2.50 per GiB) and comes on a single smallish chip (that fits in a Micro-SDHC card). At the standard Moore's Law rate of 18 months

per lambda-halving, we should see 32GiB of Flash for US$20 in a Micro-SDHC form factor around 2012, 128GiB for US$20 around 2015, and 512GiB (550GB) for US$20 around 2018.

By contrast, 500GB disks currently cost US$80, or US$0.16 per GiB. If they continue on their 15-month exponential growth curve, then 2018 will be 7 doublings in the future for them, and so we should expect to have 64TB disks selling for US$80 in 2018, or US$0.00125 per gigabyte. That's a 30× lower price per bulk gigabyte, as against 15× today.

More surprisingly, though, in 2018 we should expect to have 4TB or 8TB disks selling for US$20, just as we have 40GB disks selling in that price range today. Today buying a US$20-$40 disk instead of a US$20-$40 flash card costs you perhaps a factor of 5 in capacity; in 2018 it would cost you a factor of 8 or 16, if these predictions hold. This reduces the market available for small Flash devices.

Second, in the time between now and 2018, even the existing 15× price premium for Flash is too much for many applications. It's likely that Flash will spend many years (in systems bigger than your thumb) occupying a middle layer of the memory hierarchy, in between DRAM (US$10/GB, 100ns) and spinning rust (US$0.16/GB, 8 000 000ns). Right now Intel's 80GB SSD has access times of about 130 000ns, according to Bonnie++, and it costs about US$550 (according to a Google search; all the other prices here are from Pricewatch), which is about US$6.70/GB. Presumably this price will come down soon; I'm mystified that people are buying these things already. Maybe database journaling?

Anyway, so Flash will be filling the price/latency tradeoff gap between RAM and disk, much as disks and drums used to fill the price/latency gap between core and magtape. (Again, this is in computers bigger than your thumb. In small systems, Flash makes things possible that were simply impossible before: in the past it made possible MP3 players, cellphones that shoot video, and in the future it will make things possible we haven't thought of yet.)

The third reason that disks will remain relevant for a while is that [a lot of the current SSDs actually use a lot of power] [TH], which neutralizes one of Flash's big potential advantages. Hopefully this will get better soon.

[TH]: http://www.tomshardware.com/reviews/ssd-hard-drive,1968.html "there is indeed one Flash SSD that beats the living daylights out of any hard drive now"

murdererfs problem

This section should contain enough information to either reproduce my performance problem or figure out why I have it and you don't. It probably contains too much detail to be interesting otherwise.

I have a 30GB-max Reiserfs 3.6 filesystem (using ordered data mode, size 8192) in an 18GB sparse file on ext3fs, which I defragmented by using `cp` to make a fresh (presumably sequential) copy of the file before running this test. I'm running a fairly stock Ubuntu 8.10 Intrepid Ibex system with its standard 2.6.27-9-generic i686 (32-bit) kernel, on a 1.6GHz Celeron E1200 system with an Asus P5KPL-AM motherboard, 2GiB of RAM, and a Western Digital

WD50000AAKS-75A7B0 500108-MB hard disk, attached via SATA (although the kernel claims it's configured for UDMA/133).

Running Bonnie++ on the underlying ext3 filesystem, once with 12240M and once with 10240M of data, I got about 64000K/sec block writes with 30% CPU usage, 75000K/sec block reads with 16% CPU usage, and 102 random seeks per second with 1% CPU usage.

I mounted the filesystem, for the first time since reboot, and ran `find reiserfs-mount-point/ | time wc -l` (actually I used a different piece of software that's slightly different from `wc -l`, but that shouldn't matter.)

It took 9:07.38 elapsed time to get the 538 055 filenames from the reiserfs filesystem the first time. (Almost half of the files are directories; there are 100 directories at the top level, each of which has around 2000 subdirectories; most of these subdirectories contain a single file, but some contain several.) Repeating, it took 0:08.54 elapsed time, 64× as fast. I was running `iostat 5` in another terminal while doing this; during the first read, `iostat` reported between 100 and 220 "tps" during the first read, and between 800 and 1300 "Blk_read/s". There was also write traffic during this time, but it stopped shortly after the test was over, so I am guessing it is due to reiserfs.

The system was generally idle otherwise, with under 2% CPU usage.

The pathnames totaled 25.8 megabytes.

As I have said, no filesystem can have good read performance for every access pattern, and it can't have consistent read performance for every access pattern unless it does so by always hitting the disk (i.e. not caching). However, I do not think `find` is such an unusual access pattern that it does not need optimizing for, especially since murdererfs's raison d'être is explicitly to handle the "lots of small files" pattern better than ext2fs.

It *does* handle this pattern a lot better than ext2fs (or ext3fs anyway), which needs something like six times as much time to do the same `find` (on a different machine!), and additionally wan't any faster the second time around.

According to `iostat`'s man page, a "block" is 512 bytes, so 1000 blocks is 500kiB. So reiserfs is running a 75MB/s disk at 0.5MB/s, less than 1% of its maximum speed. The useful data it retrieved was 538 055 filenames and inodes. An inode is about 128 bytes, so there were about 69 megabytes of inodes, and the filenames totaled 4.9 megabytes, for a total of 74 megabytes of *useful* data being read. (The filename and inode data would be entirely sufficient to answer the `find` query, although directory entries might have another 5 megabytes of inode numbers, which serve only as pointers to find the inodes.) But 0.5MB/s for 9:07 (547 seconds) is 270 megabytes, so only about 30% of the data Reiserfs read from the disk was actually useful for answering the query.

If all of that data were stored sequentially on the disk, it would take 1.0 second to read the filenames and inodes, or 3.6 seconds to read all of whatever other useless data Reiserfs decided to read. If it were scattered in 1000 pieces, requiring 1000 random seeks, it would take 11 or 14 seconds to read it. But apparently Reiserfs managed to require about 55 000 separate disk transactions: roughly one transaction (and 0.5–1 seek, since it was getting 100–200 transactions per second) per

ten files!

ext3fs manages to do substantially worse.

Common Access Patterns

A filesystem should have *fast*, *consistent* performance for *common access patterns*, as well as providing a way for applications to "escape" from the filesystem's tradeoffs by providing *predictable* performance for applications that want to roll their own --- by storing their data inside a big file they structure as they please. Here's a list of the common access patterns I think are important to be consistently fast:

- Sequentially reading a large file that was sequentially written. (Almost all filesystems do well at this.)
- Sequentially reading a large file that was randomly written. (Most filesystems do well at this; LFS implementations, and ext3fs during part of the 2.6 series, sometimes fail badly.)
- Sequentially writing a large file.
- Sequentially creating and writing a large number of small files in the same subdirectory tree. (None of the filesystems I know do well at this.)
- Sequentially reading a large number of small files in the same subdirectory tree.
- Statting or opening a file, given a path from the root of the filesystem.
- Reading the metadata of a large number of files in the same subdirectory tree.
- Random read/write access within parts of a single file.

I think an approach based on Lucene's index structure can provide reasonably good, but above all consistently not bad, performance for all of these access patterns, while providing the usual POSIX filesystem semantics.

The Design

The filesystem is a bag of variable-length (key, timestamp, data) triples. There are three kinds of key: pathnames, inode numbers, and extent numbers. The data associated with an *extent id* is an inode number and a list of variable-length (offset, contents) pairs, each of which gives a block of data stored at that offset in the file to which it belongs. The data associated with an *inode number* is a list of (variable-length) pathnames that point to that inode. The data associated with a *pathname* is more complicated, but usually it is the file's metadata --- basically, most of the results of `stat()`.

Writing to the filesystem consists of adding more triples to the bag with a newer timestamp than the triples already in the bag. To get data about a file, you look up its pathname in the bag, and (if you want its contents) look up its extent ids in the bag too. To list the contents of a directory, you search for triples whose keys are pathnames in a certain range.

When there are multiple triples in the bag with the same key, you just get the triple with the latest timestamp and ignore the older ones. Eventually they will be removed by "merging", as explained below.

This is not a great match to traditional Unix filesystem semantics, since the file data is actually stored with the directory entry. So when a pathname is a hardlink to an already-existing file, instead of storing

the usual metadata, it just stores a *broken heart* containing just the inode number of the file. When we try to get a file's metadata and come up with a broken heart, we look up the inode to find out what pathnames point to that inode; the first one of them has the actual file metadata. So we look up that pathname to get to the file.

The tricky thing in such a system, which pretends to be a normal Unix filesystem where file contents and metadata are separate from a bunch of interchangeable directory entries, but actually stores them as a single unit, is how to handle deletion of the "primary" directory entry for a file with multiple hardlinks. To handle this case, we add three triples:

XXX we'd need less write bandwidth if we separated inodes, atimes, and filesystem paths.

- a new version of the inode triple with its first entry removed;
- a new version of the new primary directory entry, updated to contain the file metadata instead of a broken heart;
- a new version of the triple for the old primary directory entry, which has a special "deleted" marker instead of file metadata or a broken heart.

This isn't the simplest approach, but it's not fiendishly overcomplex.

Segments

The triples are stored in segments. A segment is a sequence of triples sorted by key, compressed with LZF, and stored in a contiguous sequence of disk sectors. Each segment is normally around one megabyte in size compressed, which should decompress to around 2.5 megabytes of data. This size is chosen because reading less than about a megabyte of data from disk is pretty much a waste of a seek, but decompressing the segment (necessary to read its contents) might take 5–10ms of CPU time. XXX not sure; compress in subsegment chunks? The sort order sorts the different kinds of keys apart: inode numbers are all together, not interspersed with pathnames, and extent ids are all together, not interspersed with inode numbers and pathnames.

I'm not sure whether the slots the segments are in should be fixed-size; presumably that entails a certain amount of wastage.

There's a segment directory, which sort of like the superblock. For each live segment, it lists the key of the first and last key in the segment, the timestamp of the newest triple in the segment, and the segment's location on disk. My new 500GB disk might be expected to contain up to around 500 000 segments at any given time; keys are probably mostly around 64 bytes; timestamps are perhaps 8 bytes; disk addresses are perhaps 8 bytes. Consequently this is about 80 bytes per segment, or about 40 megabytes for the entire disk. This can easily be kept in RAM.

To look up a single key:

- Find all the segments `seg` in the segment directory for which `seg.first_key <= key <= seg.last_key`. Sort them by timestamp, with the newest segments first. Start with `found_timestamp = 0`, where 0 is a value that is less than any real timestamp.
- For each segment in the list of candidate segments, if its

`latest_timestamp` is earlier than `found_timestamp`, bail out of the loop; neither it nor any of the later segments can contain newer data for that key. Otherwise, load the segment from disk (if necessary) and decompress it (if necessary) to see if it contains the key with a timestamp greater than `found_timestamp`. If so, remember the triple and set `found_timestamp` to the timestamp of the found triple.
• Now you have in hand the latest triple for that key.

The algorithm for finding all of the keys in a range (a "range query"; e.g. files in a directory) is similar, but can't bail out of the loop early. It is built on an algorithm for finding the first key after a given key ("get next"); there is an analogous algorithm for finding the last key before a given key ("get previous").

Segment Merging

Clearly the efficiency of this scheme, for read, largely boils down to not having very many overlapping segments, so that you don't have to consult very many segments for each query. If the segments are perfectly nonoverlapping, then only a single segment ever needs to be fetched and examined to answer a single-key probe. So opening and reading a single-extent file (in the common case where it has only one directory entry) would require a single seek to fetch its metadata, then a single seek to fetch its contents. (Although see below under "extents and extent ids".) Listing the contents of a directory (and `stat()`ing all of the files) would simply involve reading the one or more segments covering that directory's key space.

However, when we add new triples, we do it by putting them into a new segment. So whenever we write to the filesystem, even to update an atime, we create overlapping segments.

The solution is to periodically --- well, more or less constantly --- *merge* segments. A merge takes some number of segments that partly overlap --- say, between 5 and 10 --- and turns them into roughly the same number of non-overlapping segments by merging them into a single sorted sequence of keys, while dropping outdated triples, and then chopping that sequence up into some number of new segments. This merge operation should take around 400ms: say, 70ms to random-seek to 8 overlapping segments, another 110ms to read their 8MB of data, another 100ms (overlapped) to decompress it into 20MB, 6ms to merge it into 20ms of sorted data, another 200ms to recompress it into 8MB, and another 110ms (overlapped) to write it in a sequential run of new segments. The segment directory on disk need not be updated until that is necessary for some other reason.

On a machine with 8 idle processors (which should be most desktop computers most of the time, starting with desktop computers sold in 2011), the recompression step could be done in 8-way parallel, cutting the time by 175ms.

Merging should clearly be done preferentially to fairly young segments that cover a comparatively large part of the keyspace, since they will tend to cause slowness to many more queries; and it is preferable to do it to segments that are already in the buffer cache and decompressed.

So one strategy that might work reasonably well for merging under load would be to merge after answering a query that required too many seeks --- specifically, merge exactly the set of segments that were needed to answer that query. In the above scenario, this would

leave only the 6ms of merging, the parallelizable 25–200ms to recompress, and the overlappable 110ms of writing --- so maybe 30ms on CPU and 110ms talking to the disk.

That isn't optimal in general, though; it's pretty wasteful to merge a very young, sparse, and therefore wide segment with a bunch of old, dense, and narrow segments; in the limit, you end up with the first half of the young segment, followed by the first half of an older segment, followed by the contents of a yet older segment, followed by the second halves of the other two segments. It's much more profitable to merge segments that are similar in density.

Every byte written to a new segment, if not superseded, must be eventually read, merged, and written into some new, denser, merged segment N times before reaching its final resting place in a very dense segment. This would seem to mean that only 1/N of the disk write bandwidth is available for writing new data! This may be ameliorated somewhat by churn, but it remains the case that to get a disk from empty to mostly full with this scheme, the data must be copied N times.

For 500 000 total segments and average 8-way merges, randomly distributed data would need to go through 6 or 7 steps to reach a totally non-overlapping state.

However, the inode numbers are not natural keys; it's probably possible to keep their segments from overlapping very much by the simple expedient of allocating them sequential serial numbers from a space large enough that it never gets exhausted. (Again, atime may play havoc with this.) (This may be a reason to go back to the traditional Unix approach of storing the inodes separately --- you'll have new filenames inserted between old ones much more often than you'll have inodes inserted.) And for all but the smallest files, it's the extents that really matter. My original data set, for example, had 538 055 files totaling 18GB; their filenames totaled 4.9MB; their uncompressed inodes presumably were 67MB. So the presumably mergey metadata is only 1/300 of the total, while the hopefully nonmergey file contents data is the rest. See below about "extents and extent ids" for how those are handled.

Extents and Extent ids

XXX need special case for merge and/or lookup; should extent id include length?

An extent id consists of an (inode number, offset) pair; so retrieving the extents associated with an inode consists of a range query. The data associated with it in the filesystem is some number of bytes of the file contents, starting at that offset. So to retrieve the contents of a file, you just retrieve all extents identified with that file's inode number; to retrieve file contents starting at an arbitrary offset, you use the "get previous" algorithm to find the last few extents that start at or before that offset, and take the most recent one that is long enough to reach that offset.

Extents have a maximum length, which I think should be around 32kiB, for several reasons:

• If extents had unbounded length, potentially into tens of gigabytes, they would create segments of unbounded length; this would impede random reads from the middles of files.

• If extents had unbounded length, then reading the byte at location 28920620234 in a file would require reading all of the extents starting at locations 0 to 28920620233, of which there could be up to 28920620234, because until you have read them all, you never know if the next one is going to be a 28920620234-byte long extent that's newer than all the ones you've seen so far. Limiting extents to 32kiB means you only have to fetch the extents beginning in the previous 32kiB in order to make sure you have the current version of the file data.

Making the maximum length too small, however, will impede compression efficiency and impose too much scatter-gather overhead.

In a contiguous bulk write, you'll lay down segments consisting entirely of new extents (already in order) and new versions of the file's metadata with updated size and mtime. These segments, after a merge, will keep the extents in order, and the extent segments will be "dense" in the sense that no other segment will ever overlap their key space, except when it updates that same file. Consequently that data will never need to be moved on disk unless it's updated.

When partial updates of a file are written to disk, they will be in a separate segment from the original data; the file is "fragmented". Eventually a merge will bring all of this data together; this is "online defragmentation".

Continuous snapshotting

If you don't delete triples with timestamps older than some cutoff date, you can roll time back to any arbitrary point in the past since that cutoff date, at some overhead cost to reads of data that's been updated since then. This permits a variety of useful features:

• online database table backup (you don't have to worry that the file is being modified while you back it up, giving you inconsistent backups; just backup a snapshot)
• file undeletion
• general undo
• filesystem change auditing ("what did that install just change?")

View Updating

If you have some application that wants to maintain some data that's dependent on the data currently in the filesystem --- say, locate (although with this filesystem, `find` might be fast enough to render `locate` obsolete), or a full-text indexing system, or something like `make` that runs instantly, that application currently has to function as follows. At startup, it scans the entire filesystem, or the subtree it cares about, in order to compare them to its database. As it scans, it uses `inotify` to request notifications of any future changes. Then it has to sleep, waiting on notifications, for a long enough time to justify the expense of its startup scan.

Things like these are analogous to "materialized views" in a database: a full-text index is a "view" of the files it indexes, a compiled program is a "view" of the source code that makes it up. So I'm calling this general problem "view updating", because it's analogous to the problem of updating materialized views in a database.

By contrast, on this filesystem, the segment directory contains a latest-timestamp for each segment, and most segments should contain data written during a fairly narrow time window, so you can efficiently retrieve all of the segments containing data written since some recent timestamp (if it hasn't already been merged with older segments.) And all of the data is efficiently reverse-mappable: if you find that there's a new extent, you can look up what pathnames it belongs to quite quickly, even if the mtime in the inode hasn't been updated (e.g. because the update followed another update in the same second).

This allows you to run view-updating programs on-demand, or from `cron`.

Filesystem Resizing

Resizing a mounted filesystem is fairly straightforward, either to expand or shrink it, and doesn't pose any special risks due to power failure at the wrong moment. Expanding the filesystem just adds some free space for use in new segments; the segment directory needs to be expanded correspondingly. Shrinking the filesystem requires relocating (and, if possible, merging) the segments in the region to be freed.

Use as Full-Text Index

If you have a bunch of (word, document_id, position) triples, you can encode them as pathnames: "#{word}/#{document_id}/#{position}", and create those paths as empty files. The efficiency of operations such as bulk-create, merge, and lookup should be comparable to the efficiency of the same operations in Lucene, although the inodes will make it somewhat worse.

Directories with Lots of Descendants

There is a case where this design is slower, even unpredictably slower, than standard designs like ext3fs: `ls`, when you're in a directory that has a lot of descendants. In theory, `ls` at the top level of the filesystem has to scan through the entire space of pathnames (you could potentially have billions of pathnames) in order to extract the dozen or so immediate descendants of the root directory. However, there's a reasonably efficient way to solve this problem.

Let's assume for the moment that the pathname segments are nicely merged so that there are no overlapping segments. Now we need to consult the segment directory to find out which segments might contain pathnames under, say, the root directory. But the first-and-last-key information we get this way actually contains enough information to know that most of these pathname segments don't contain any transitions between children of the root! For example, a segment that begins with `/usr/lib/perl/5.8.8/regcomp.h` and end with `/usr/lib/python2.6/xdrlib.py` doesn't contain any children of / or even `/usr` --- only descendants of `/usr/lib`.

So this gives us a worst-case bound: at worst, we only need to fetch one segment per child of the directory we're listing. For the `ls /` case above, this amounts to 120ms. However, only large subtrees (over around 100 000 pathnames) will actually push the next child

into another segment, so at worst, we only need to fetch one segment per *large-subtree* child of the directory we're listing.

out of place XXX

The advantage of storing things in this way is that it optimizes for the common case:

• statting a file through its primary directory entry (and most files have only primary directory entries) only requires a single contiguous read from the relevant spot in the segment. (How you find that spot is explained later. It may require a few seeks.)
• reading or creating and writing the entire contents of a subdirectory tree

Intra-segment structure

The data inside a single segment will normally decompress to around 2.5MB, but if the compression ratio is good, it could be considerably larger, like 20MB or more. But decompressing even 2.5MB of data will totally blow your cache, so you don't want to do that too much if you can avoid it.

So the segment is compressed in about 16 "sub-segments" of about 64kiB of compressed data, each prefixed with an (uncompressed!) length field and the first key in that segment, and containing some number of whole triples. This allows you to find the right sub-segment (assuming you only want one sub-segment) in about 16 cache line fills, and then decompress it into your L2 cache.

There is a SHA-256 sum of the uncompressed data inside of that data, at the end of the subsegment, in order to make it possible to verify that the data hasn't been corrupted by memory, disk hardware, or the decompression/compression step.

For large extents, the sub-segments will contain about two maximal-size extents each. Small data, such as inodes and pathnames, might be 128–256 bytes per triple uncompressed (and 40–80 bytes per triple compressed). So a sub-segment might contain a thousand or more triples. To allow finding the right triple quickly, the triples inside a sub-segment form a sort of skip list --- each one is prefixed not only with its own length but with some number of "pointers" giving the length of the sequence of 2, 4, 8, etc., triples that it begins. These pointers can be two bytes (they need to be fixed-width so they're easy to backpatch) and they add about 2000 bytes to the uncompressed data. This allows search inside a sub-segment to use at most about 20 probes.

XXX now we have three levels of index using different structures: the segment directory (which presumably has some in-memory indices of its own), the sub-segment directory, and the skip list. Maybe these could be unified?

XXX Are segments placed in fixed "segment slots" implying slack space at the end? Like what, 10k?

At the end of the segment, there's a SHA-256 of the SHA-256s of the sub-segments to allow early detection of serious segment corruption.

fsck

The filesystem has some internal invariants: the list of pathnames at an inode should match the set of pathnames that have that inode number; the size in the inode should match the

Details

stat() (inode number, mode, number of hard links, uid, gid, size or device ID, atime, ctime, mtime)

Durability

It's highly desirable that the filesystem not lose committed data when the machine loses power suddenly.

Group commit. Micro-segments. NVRAM/network backup/Flash.

Why Transparent Compression

Segments are compressed before being written to disk and decompressed when read from disk. This has several benefits.

First, it decreases the amount of disk bandwidth needed. The disk interface is still fairly slow; the machine I'm testing this on can sequentially write to the disk, as I've said a zillion times already, at 64 megabytes per second. Also, if I understand correctly, it can do 1600 million 32-bit memory transactions per second, which is 51.2 gigabytes per second.

Second, it means we don't care very much about the representation efficiency of the on-disk data structures. We don't have to agonize over how many bits to allocate to the inode field or how to represent numbers efficiently or how to represent pathnames efficiently or how to avoid storing a separate uid, gid, ctime, and mtime for each small file in a large directory.

As an example, I made a list of 2000 pathnames from my Reiserfs partition. (Yes, that did take 20 seconds.) Raw, the pathnames are 57kB (or 121kB if you include the mount point at the beginning of the paths instead of starting them with "./"). Compressed with the `frcode` program from GNU findutils 4.4.0, a program carefully designed to compress precisely such sequential lists of pathnames, it is reduced to 18kB. But compressed with LZF (a general-purpose high-speed compressor), it's 13kB, and compressed with gzip, it's 10kB. The compressed numbers don't change significantly depending on whether the pathnames include the mount point.

So the effort that went into writing `frcode`, and its corresponding decompressor, and the ";login:" article about it, and maintaining it over the years, has been wasted, and actually counterproductive; if `findutils` used `gzip` (or originally `compress`) instead, it would have been getting better compression all along. Still, `frcode` is only about 200 lines of code; but there are many carefully optimized but very specific representations like that, scattered around the system.

I tried another example that wasn't so successful: compressing an executable versus *compressing a hexadecimal dump* of the executable --- two hex digits in place of each byte in the original executable, with no newlines or offset numbers. The original executable was 34kB; hexdumped and compressed with LZF it was 28kB; compressed with LZF without hexdumping, it was 21kB. (`bzip2` was able to compress it to 16kB, which is perhaps a reasonable estimate of the actual entropy

of the data. It did only slightly worse on the hexdump version, like 17kB.) So hexadecimal-encoding frustrated LZF's ability to extract redundancy to some degree, but LZF was still able to remove more redundancy than the hexadecimal encoding added.

As a third example, I made a *list of 1533 numbers* that were the sizes of files in a directory tree. Encoded as ASCII decimal numbers separated by newlines, they took 6.6kB; binary-encoded as 32-bit integers (none of the files happened to be 4GiB or over), they took 6.1kB; the list of ASCII numbers compressed with LZO was 4.5kB. (LZO-compressing the numbers in their binary form put them at 4.9kB.) So there's no gain in space-efficiency in this case from using binary encoding, and actually a substantial loss --- barely a win over just using ASCII decimal numbers. (Note that ASCII decimal numbers also have no arbitrary size limit.)

Third, compressing segments means that groups of small, similar files in the same directory can be compressed as a group, as can groups of inodes. Long experience with the `.tar.gz` format has shown us that this is a win, often a significant one. Storing the file contents in extents separate from the metadata means that both the files' contents and the metadata are intermixed with a minimal amount of foreign data in a different format.

Fourth, by removing, in most cases, the need for applications to compress their data files, it can make those data files easier for users to reverse-engineer, modify, and use for other purposes. For example, Gnumeric's data file format is `gzip`ped XML. I made a simple spreadsheet with 45 cells; the Gnumeric data file was 2kB, the uncompressed XML was 13kB, and the LZF-compressed data file was 3kB. Clearly the 6× reduction in bulk is worthwhile; but a Gnumeric designed to run on an LZF-compressed filesystem, where `gzip` saves only 30%, could reasonably have chosen a different path. (Obviously Gnumeric is pretty far down this path already by using XML to store spreadsheets.)

Fifth, it effectively increases the size of the buffer cache substantially, increasing the buffer cache hit rate dramatically.

Sixth, it effectively increases the size of the disk substantially for no effort on the part of the user or their application software. (This is the traditional reason for filesystem compression, but I think it's less important than the previous four.)

Seventh, it may improve the security of whole-disk encryption. This is a disadvantage too --- it makes data recovery more difficult.

Compression Isn't Too Costly

Despite all the advantages I ascribe to transparent disk compression in the previous section, we wouldn't want to use it if it made the filesystem too slow. It turns out that it shouldn't.

`gzip -1c` on one of the CPUs of the 1.6GHz Celeron E1200 gets about 3.3× compression at about 20MB/s input. At `-9` it gets about 3MB/s and gets (on a test file, i.e. my reiserfs image) 3.9×, i.e. about 15% smaller. `gzip -d` decompresses at about 60MB/s output, and is about 15% faster when the input data is compressed with `-9`.

These are pretty expensive, CPU-wise. But if we assume that typical computers in the next few years will have more CPU cores than hard disks, and that compression and decompression of independent segments can be partitioned into embarrassingly-parallel

tasks, it might be reasonable.

However, `gzip` (LZ77) is not one of the fastest lossless compression algorithms out there. There are other algorithms around that beat the living shit out of `gzip` for speed.

LZW is no longer patented, and supposedly [compresses at 50 kilobytes per second on a 386] [TIFF], which runs maybe 12 million 32-bit instructions per second --- maybe 250 instructions per byte --- so you'd expect it to run on one of the Celeron's CPUs at about 50 megabytes per second. However, Thomas and Woods's `compress` program only compresses at about 10 megabytes per second on the same machine, half as fast as `gzip -1c`, and only compressing by 2.7×. LZW was discovered in 1984; it's about 2K of code to implement.

LZO and LZF are two new LZ77 variants with, like LZW, low CPU usage. LZF is being used in the kernel for suspend to disk.

LZO was supposedly a third the speed of `memcpy` at decompression on a Pentium 133 --- 20 MB/sec to `memcpy`'s 60MB/sec --- and four times slower for compression. On my same machine that I was testing with earlier, `lzop` 1.02rc1 (using liblzo 2.03-1) compresses at 64—100 megabytes per second and decompresses at 143 megabytes per second. It compresses by about 2.7×, slightly better than `compress`'s LZW. Apparently [in 2002 on a 200MHZ Pentium] [ACT] `lzop` 1.00w took 1.47 seconds to compress (and 0.99 to extract) a corpus that `gzip` 1.2.4 (with no options, which seems to be about `gzip -6` these days, 10 megabytes per second) took 15.57 seconds to compress.

That means LZO used to be 10.6× as fast as `gzip -6` at compression at the time, but now it's only 3–5× as fast. Maybe `gzip` got better, but also it seems that LZO's performance has gotten relatively worse --- it was 20MB/sec on a Pentium 133 and only 3–5× faster on a 1.6GHz four-issues-and-retires-per-cycle processor, which does 1600 million memory transactions per second instead of 66 million?

LZF is even faster. It only compresses by about 2.4×, but it compresses on the same hardware at 77—138 megabytes per second and decompresses at 138—199 megabytes per second. (I built liblzf-3.4 with `./configure && make -j3` with GCC 4.3.2-1-ubuntu11, which used `-g -O2 -O3 -funroll-all-loops`.) I don't understand the algorithm well enough to see if it could be optimized more or run on a GPU, but the current version is a small amount of C and seems to include CRC32 checking.

[TIFF]: http://www.fileformat.info/format/tiff/corion-lzw.htm "The TIFF LZW Compression Algorithm" [ACT]: http://compression.ca/act/act-text.html "Jeff Gilchrist's ACT Text Compression Test"

Topics

The TWI and I²C buses and better alternatives like CAN and RS-485

Kragen Javier Sitaker, 2018-06-28 (updated 2018-07-05) (24 minutes)

Most AVRs support "TWI", their slightly bastardized version of Philips I²C. In theory, this should allow you to hook up any number of AVRs (and maybe other devices) on a shared two-wire bus (SDA and SCL), or up to 113 of them, anyway, and communicate at 400 kbps. It even supports address-recognition wakeup, even from deep sleep modes that don't run a clock to the TWI interface — it uses the bus clock itself for wakeup!

In particular, the ATMega328P used in Arduinos supports it.

Actually, it turns out that only slave devices need addresses; because masters can initiate both reads and writes, they do not need addresses. So you could connect an infinite number of masters to the bus. Fat lot of good that'll do you, though, if they can't talk to each other!

What I was thinking with this is that if you want a bunch of GPIO pins, more than the Arduino has, or want to control more power than the Arduino can, it might make the most sense to add some more chips on a TWI bus in order to add those GPIO pins. This could potentially also give you modularity — you can plug boards together with just four wires, as long as you don't have slave address conflicts, which will probably happen around 10 devices without some mechanism to assign addresses dynamically.

I don't think the I²C bus deals with chips running at different voltages.

Fatal problems with I²C

As I dig more into this, it seems increasingly impractical as a way of building a modular system that is easy to extend, for the following reasons:

• Address assignment: aside from the occasional awful IC with a fixed I²C address or only a few bits assignable, even randomly assigned addresses over the whole 7-bit space will give you collisions after 11 devices, on average. To avoid this, you have to intervene to assign addresses by hand, but even then, Elliot Williams says he's never seen as many as 20 devices on an I²C bus. Some kind of daisy-chaining arrangement like JTAG uses would be a lot better.
• Voltage incompatibility: I²C high is 0.7 Vcc or above, which means that the 3.3 V power rail isn't high enough to be high on a 5V system. There's a standard technique to work around this with a discrete MOSFET per pin.
• Speed: 400kbps is pretty slow. This results in part from the single-ended nature of the I²C bus.
• Fanout: the drive capability you need to make your I²C bus work depends on the speed you're running it at and the total capacitance of the devices attached to it. If you have too much capacitance attached for the drive current of a device, you can't make it all the way down to 0.3 Vcc (its low level) in time. 5 mA should be fine for a couple of

devices but maybe not 10 devices, depending in part on signal routing, and the standard specifies 400 pF as the maximum. And you need stronger pullups if you have more devices, which of course requires still stronger pin drive.
• Bus hangs: there's a known bug in I²C where resetting a master in the middle of a transfer can leave a slave hanging the bus indefinitely. SMBus fixes this with timeouts.

Addresses

You would think that with 7-bit addresses, you would get 128 devices, but the address 0000 000 and the 8 addresses 1111 xxx are reserved, so you actually only get 119 devices. And actually 0000 xxx are reserved for other purposes, though Atmel doesn't document this, so you only get 113. 0000 000 is for broadcast. The bus arbitration algorithm provides strict priority among slave addresses; the broadcast address is the highest-priority possible address.

Bit rates

An address packet is 9 bits long, and following an address packet, you can transmit any number of 9-bit data packets, each bearing 8 bits of data. There are an additional two bit-times at the beginning to indicate the START condition and two more at the end to indicate the STOP condition. This ought to mean that you can transmit 20000 one-byte packets per second, or up to 44000 bytes per second in large transmissions.

The AVR implementation supposedly supports clock stretching, and indeed depends on it in order to give interrupt handlers time to respond.

The bit rate is set by the TWBR register to (CPU clock frequency)/(16 + 2 · TWBR · prescaler), which puts a maximum bit rate of 1/16 of the clock speed. For clock speeds over 6.4 MHz (including the maximum internal RC oscillator speed of 8 MHz) this should not be a consideration, but systems that use lower clock speeds to get better power consumption might be limited. (And apparently the CPU clock needs to be at least 250 kHz for TWI to work at all). In theory this only affects communications that include the slow chip.

The possible prescaler values are 1, 4, 16, and 64.

Electrical limitations

For reliable operation, the AVRs' 20 mA drive needs to be able to discharge all of the input capacitances on the bus at well over 400 kHz — say, in a microsecond. Worse, the pullups need to be able to charge them, and the drive needs to be able to fight the pullup. This suggests that only a couple of thousands of pF of input capacitance on the bus can be tolerated.

However, some other devices have smaller drive capabilities.

Chip support

Bit-banging I²C or TWI seems very challenging, due to requirements of bidirectional open-collector pins with slew rate limiting and spike filtering. It seems like something you could do with an external chip, but that's kinda what we're trying to avoid

here.

AVRs have interrupt support for TWI, but the interface involves one interrupt per byte transferred, and occasionally more. At 400 kbps and an 8 MHz CPU clock, you have at least 180 cycles between successful complete byte transfers.

The slave address register TWAR can be set to whatever address you want.

The ATmega328P, like its smaller variants the ATmega48A, ATmega48PA, ATmega88A, ATmega88PA, ATmega168A, ATmega168PA, and ATmega328, supports a single TWI bus on pins 27 and 28, or balls 4B and 4A in its UFBGA incarnation. The ATmega48/88/48PB/88PB/168PB supports a single TWI bus on pins 27 and 28. The ATmega16U4 used on the Arduino for its USB interface, and its larger version the ATmega32U4 (also the core of the Adafruit Feather), support a single TWI bus on pins 18 and 19. The ATmega8A supports a single TWI bus on pins 27 and 28. The ATtiny20 has a TWI bus for slave mode only on pins 6 and 3 (out of 14), pins 12 and 15 of its 20-pin VQFN, balls 2B and 2B of its UFBGA, or balls 3C and 5C of its 12-ball WLCSP. The ATtiny40 has a TWI bus for slave mode on pins 16 and 13 (out of 20), or 11 and 14 in VQFN.

The ATmega328P comes in a 4 mm square VQFN and a 4 mm square, 0.6 mm thick UFBGA, but no smaller packages. This is smallish but even the UFBGA is 8 times the size of the ATtiny20 12-ball WLCSP mentioned above.

The obsolete ATtiny2313's USI claims to support TWI, but without slew rate limiting and spike filtering, and it sounds like you pretty much have to implement the protocol in software. It is not clear to me that this will work, and definitely it is not interrupt-driven.

The ATtiny25/45/85 and ATtiny13/ATtiny13V do not support TWI, just SPI. (I think the 25/45/85 may have a 2313-like USI.) The ATTiny4/5/9/10 don't support either TWI or SPI.

More detail on the ATtiny20

The ATtiny20, despite being slave-only, is especially appealing for adding I/O lines to a distributed system linked by a TWI bus because its WLCSP incarnation is 1.56 × 1.40 mm and 0.54 mm thick, and its UFBGA (like the VQFN for the ATtiny40) is 3 mm square. Even its TSSOP and VQFN are only 5 mm square.

The ATtiny20 additionally supports 10-bit extended addresses and address masking, although that isn't useful without a similarly capable master to communicate with.

This tiny size still has a substantial current drive capability, though; at a drop of 0.8 volts, it can sink or source the usual 20 mA per pin at 5 V or 10 mA at 3 V, except on its reset pin. Running at lower voltages lowers the possible current substantially.

Digi-Key sells ATtiny20s in most packages from 56¢ in quantity 1, but the WLCSP costs 92¢.

Non-AVR chip support

Many other things nominally support I²C, although apparently compatibility problems are not unusual. Many EEPROMs support I²C — this is the main use of I²C actually — and the popular Cypress

CY7C68013A/CY7C68014A/CY7C68015A/CY7C68016A EZ-USB FX2LP 8051 supports 100 or 400 kHz I²C, for example, but only as a master; it can use this for booting from an EEPROM at startup. The popular ultra-low-power TI MSP430G2x53/MSP430G2x13 microcontroller supports I²C; not sure how much of the rest of their family does.

As an example of EEPROMs that support I²C, consider the AT24C32/64, with 4096 and 8192 bytes, respectively, 5 mm × 4 mm in SOIC or 3 mm × 4.5 mm in TSSOP. These use 3 of their 8 pins to set the I²C address of the EEPROM to 1010xxx (so you can gang up to 8 of them on a bus) and support the 400 kHz rate at 5 V. They support writes of up to 32 bytes at a time, or longer if what you want is a 32-byte ring buffer.

These EEPROMs have their own internal charge pump for erasing, so they need only a single supply. They can drive 5 mA and have 8 pF of input impedance, which works out to 50 kΩ at 400 kHz, so in theory support fanout of about 50. This is much less than the total of 119 from the address limits.

The other microcontrollers I'm most interested in are the STM32 family, the LPCxxxx family, and the ESP8266/ESP32 family, just because they seem to be the most popular at the moment (other than PICs, which I would prefer to avoid entirely).

The STM32F0 does support I²C, including 10-bit addresses (and some even have two I²C interfaces), and it's even functional in "low-power stop modes", which I guess means it can turn the chip on. I think it's only 3.3 volts, though, which seems like it could pose interoperability problems. The cheapest STM32 at Digi-Key is the STM32F030F4P6, which goes for US$1.30, down to 59¢ in quantity. The cheapest STM32 with CAN is the STM32F042F4P6, which is US$2.18 down to US$1.07.

The LPC1769 naturally supports I²C, and actually supports more than one bus per chip, I think.

The ESP32 supports I²C.

The TI DRV8830 is a 6.8V 1A H-bridge chip controlled over I²C.

Other peripherals? ADCs probably don't make sense (the AVRs have ADCs built in, and higher-speed ADCs are too fast for the I²C bus; they would need to just be high-precision, low-speed ADCs) but things like LCDs, DACs, and high-power switches ("drivers") might make sense. Also radios, of course. H-bridges or ESCs would be super nice. RAMs might be useful too, even if a bit slow. How about other radios, including LoRa and BLE?

The ONSemi NCP5623 is a linear I²C RGB LED driver that can drive three LEDs at up to 90mA on 2.7 to 5.5 V using current mirrors, but with only 32 PWM levels. I can't figure out how its address is determined or what its PWM frequency is.

The ONSemi LV8498CT is a voice-coil motor driver IC with I²C control; it's basically a current-mode 10-bit DAC running up to 150 mA at 5 VDC. Its slave address is 0110011, so you can only use one of them on a bus. I can't figure out how fast or slow it is.

The ONSemi LV5236V is a 24-channel 5V I²C LED driver with 5-bit PWM and/or up to 50–100mA per LED, or maybe 30 mA per LED controlled by a DAC, I can't tell. It's 5.6 mm × 15.45 mm. It has five address pins, so you can set its address to any 10xxxxx. Digi-Key will charge you US$3 for one, which works out to 12.5¢ per

LED.

Maxim has an LM75 I²C temperature sensor with three address pins to configure its address to any 1001xxx address.

Alternatives to I²C

SMBus

SMBus is a slight tweak on I²C which adds a few requirements to prevent hung or powered-off components from screwing up the bus, but it doesn't solve the fundamental problems.

CAN

The CAN bus sort of seems to be designed as an answer to some of these problems, but for some reason CAN bus drivers are expensive, and anyway they don't solve the problem of address assignment.

JTAG

JTAG has the desirable attributes of being daisy-chained and thus partly avoiding the problems of address assignment and fanout. It uses four or five wires, not counting power supplies: TCK, TMS, TDI, TDO, and optionally TRST*; you chain TDO of one chip to the TDI of the next, but you run TCK and TMS to all the chips, thus still potentially having fanout limits.

TMS is "test mode select", which clocks in a sequence of bits to drive the JTAG controller state machine. In particular, the sequence 11111 will always drive the state machine to its reset state, where it will remain as long as it gets more 1 bits; from there, the introduction of strategically placed zeroes into the TMS data stream can navigate it to other states, five of which are stable on 0 (i.e. have 0-edges to themselves). TMS bits are clocked in on the rising edge of TCK, and then the resulting states can cause TDI bits to be clocked into things on the falling edge of TCK.

The reset via TMS is somewhat fault-tolerant in the sense that a single spurious 0 is not sufficient to transition the state engine to take any action; three more 1s in succession will successfully drive the state machine back to the reset state.

At times, depending on the state of the JTAG state machine and the "current instruction", TDI is clocked directly to TDO, converting a whole chip into just a single clock delay. At other times, a shift register is interposed between TDI and TDO, but which one depends on both the JTAG state machine and the current instruction — it can be the current-instruction register or a data register determined by the current instruction. Two of the aforementioned five stable states, Shift-DR and Shift-IR, are the ones that interpose shift registers.

The instruction register is required to be at least 2 bits because there are 4 required instructions: BYPASS (all 1s), EXTEST (once, all 0s, but then they decided that was a bad idea), PRELOAD, and SAMPLE, which may be the same as PRELOAD.

The TDO line is supposed to be "set to its inactive drive state except when the scanning of data is in progress", which turns out to be when the chip is in Shift-DR or Shift-IR state. This allows you to share TDI and TCK between chains and wire their TDO lines together, using a separate TMS for each line to select which one will

be active.

Two other optional states in the rather complicated (16 states!) state machine permit either overwriting the shift register from data held elsewhere (i.e. moving data from an internal register into the shift chain) and overwriting data held elsewhere from the shift register.

Although the state machine is complicated, the standard actually includes a circuit diagram showing that you can implement it with 32 NAND gates and 8 D flip-flops, under 200 transistors.

I like this idea of using a sequence of bits to maneuver a state machine around, and I like the idea of bucket-brigading a bunch of bits through a daisy chain, but I don't like the fanout of TMS and TCK, even though they're always driven by the bus master, and so in the worst case just need a couple of big Darlingtons. I really like the idea of altering the bucket-brigade topology at runtime by bypassing some devices in order to prevent latency from the bucket brigade. I don't particularly like the separation of TMS and TDI, which seems unnecessary — JTAG ends up needing 6 wires if you include power, while CAN and I²C make do with only 4.

A hypothetical super-JTAG

What if you could redesign JTAG?

Bit-stuffing nonsense

To unify TMS and TDI, a very simple kind of bit-stuffing could use a sequence of 5 1 bits as a magic resynchronization/reset sequence, and when transferring data, send nybbles of 4 arbitrary bits preceded by a non-optional 0, thus preventing the magic sequence from occurring regardless of the data being transmitted, at only a 25% overhead.

Following the magic reset sequence, or indeed following a single 1 following a nybble of data, we could maybe have a variety of different states.

To provide addressing of individual slave devices, one possibility is to have a state that decrements a fixed-width little-endian hop-count address field, for example of 8 bits; if the borrow is set at the end, it means it rolled over from 0 to 0xFF, which means you're the intended destination! This entitles you to overwrite whatever payload data may follow, so that when it eventually gets shifted around to the master again, it contains your reply.

If we want to get rid of the clock line too, we might want a different kind of bit-stuffing that ensures frequent transitions.

A better, connection-oriented approach for unicast

Consider a simpler approach in which the bus master repeats these three steps repeatedly: 1. establish a connection to a slave node; 2. communicate with it; 3. terminate the connection. In a daisy-chain topology, step 1 could be as simple as sending a time-to-live count byte, or even an unary-encode count, which gets decremented on its path through the chain; intermediate nodes would change to a "passthrough" state and forward the data, bit by bit or byte by byte. Steps 2 and 3 could then be distinguished using, for example, HDLC bit-stuffing, constant-overhead byte stuffing, or SLIP framing.

Hmm, I guess that isn't very different from the previous approach, actually, except that in this approach, I was thinking not to forward packets that didn't need to be sent on further. Instead, the slave

addressed would simply send reply data back to the master. It would be more like a traditional serial connection than a packet-switched network, or SPI, or the ISA bus.

You could also have a special broadcast address for addressing all slave nodes at once, or within network latency anyway; and barrier synchronization of the master waiting on all slaves could be achieved by yet another kind of packet which a slave only passes to its successor if it is in "waiting" state.

At 1Mbps, which should be easy to reach, and one byte of buffer per node, the latency of a 256-node daisy chain would be 2048 bits: 2 milliseconds. This might be too much latency to replace Fabnet. 10Mbps should be electrically easy to reach but might be a larger computational load.

If the data is differentially encoded on a single twisted pair — as in RS-422 or RS-485 — a dc bias on this pair can be used to provide power from one board to another. This is more suitable for board-level connections than chip-level connections. As I've argued previously in Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642), a constant-current supply meshes nicely with daisy-chaining boards together, and allows the use of thinner wires. Consider using 24AWG copper phone-line wire, as I suggested there, 510μm in diameter, 1.8 g/m, 84mΩ/m, with a constant-current supply running up to 48V at the 3.5 A maximum for that kind of wire. This gives you a maximum of 168 W for the overall system, although of course anything that needs more power than that could use a separate power supply.

This allows you to do both power and data on just two wires from each board to the next, but the final board needs a final pair of wires to be brought back around to the master to complete the circuit. So each board needs four terminals in the end. You could reduce the number of cables by 1, and potentially isolate and partly tolerate connection errors, by using a four-wire cable from each node to the next, two of which are merely the return path and are wired straight through from the downstream socket to the upstream socket.

Differential bus connections like the CAN bus require only two connections per board, but can't also supply power at the same time. So CAN bus boards end up requiring at least four terminals, too.

RS-485

RS-485 is the basis for the Fabnet bus used in Peek's dissertation. It's a multidrop version of RS-422, which is a differential version of RS-232. Because they use terminating resistors and balanced transmission lines, RS-422 and RS-485 can reach data rates of tens of megabaud over short distances. RS-485 can be used either in a two-wire "party line" mode or a four-wire "master-slave" mode, but I think neither version has an arbitration algorithm for when multiple devices attempt to transmit at the same time.

USB

Topics

Notes on running QEMU on Debian Etch

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

I'm running QEMU with kqemu on my old 700MHz laptop.

User-mode stuff is slowed down only slightly. This command line:

```
time for x in $(seq 10000); do :; :; :; :; done
```

takes 1.17 1.19 1.20 1.22 user seconds in emulation and 1.13 1.13 1.14 1.14 user seconds outside QEMU.

However, it takes about 100ms of system time in place of about 10ms. (The `-kernel-kqemu` flag may solve this; haven't measured.)

I had some kind of keyboard problem when I ran QEMU 0.8.2-4etch1 with `-snapshot`. Like, the keyboard just didn't work. That problem went away when I built QEMU 0.9.1 from source and started using that, but I still can't use `-snapshot` and `-loadvm` together.

Networking: `tap`

This was a bad idea (for me).

By default, QEMU uses `user` networking, which proxies network connections through normal sockets, like `slipknot` or `slirp` or `term`. (In fact, it uses `slirp`.) I thought this didn't give me a way to talk to it over the network (for example, if I'm running a web server on it).

So I thought `-net tap` could help with this, but it has some drawbacks. It requires running QEMU as root, and then the network interface on the emulated machine needs to be configured statically, e.g. in `/etc/network/interfaces`, since `-net tap` doesn't provide DHCP by default. And then you have to set up IP masquerading, more or less as follows:

```
qemu -net nic -net tap,script=ifup "$image"
```

In file `ifup`:

```
set -e
/sbin/ifconfig "$1" 172.20.0.1
echo 1 > /proc/sys/net/ipv4/ip_forward
/sbin/iptables -t nat -A POSTROUTING --source 172.20.0.0/24 -j MASQUERADE
```

This does actually work, but you have to configure the network stuff inside of QEMU: IP address, netmask, default gateway, and worst of all, DNS server. And I think it might allow other people on your LAN to masquerade through you.

What would be ideal would be bridging the virtual interface to my real Ethernet interface, but I never got around to doing this.

Networking: `-redir`

It turns out there's an easier way. I can use the default `user` networking, and if I have a web server on the emulated host on port

8080, I can say

```
qemu -redir tcp:8000::8080 "$image"
```

and connect my web browser to http://localhost:8000/.

This works beautifully. The one downside I've found is that if you're using `qemu -loadvm`, the inner virtual machine has to re-request DHCP before the redirection works.

Startup: `-loadvm`

Bootup takes an annoyingly long time. But, if you don't regularly have any permanent changes you want to save, you can use the `savevm` command to save an image of the virtual machine state after a boot, and then use `qemu -loadvm` to start QEMU in the already-booted state.

Topics

Only a constant factor worse

Kragen Javier Sitaker, 2013-05-17 (16 minutes)

I read somewhere that the "optimal" approach to buying a money-saving appliance that you're not sure how much you'd use is to keep track of how much you waste by not having it; when the total of waste reaches the cost of the appliance, you buy the appliance. This way, your worst-case expenditure is twice the cost of the appliance, and your best-case expenditure is nothing. And, with this policy, it's very likely that you'll buy the appliance if it will save you money, and you won't if it won't.

This actually works for any constant factor of the cost of the appliance. You could buy the appliance when your total potential savings reach 75% of its cost, or 200%; the underlying principle is the same. Depending on your priors (how likely it is you'll keep doing what you're doing) and your time preference for money, it might make sense to adjust the factor.

Presumably whatever benefit you'd be getting more cheaply with the appliance is more valuable than the amount you're wasting by not having it --- say, having a washing machine might save you $25 a week in laundry-service costs, but having clean clothes to wear is presumably worth more to you than the $25; and having a camper bus might save you $100 a night in hotel-room fees when you travel, but presumably if traveling isn't worth $100 a night, you wouldn't be doing it before buying the camper bus.

Some other possible strategies have, in some sense, an unlimited downside. "Never buy" can cost you an unlimited amount of money --- $100 a night for all eternity, say --- and while "buy just in case" won't cost you an unlimited amount of money, the ratio between the benefit you get and the cost is unlimited. For example, you could spend $40000 on a camper bus you never use. If you use it for just one night, you'd have gotten better value for your money by spending $10000 on a really nice hotel room. (Not that this is a reasonable strategy.)

The buy-when-costs-reach-predetermined-multiple-price strategy omits a couple of significant factors, though: the cost of owning the appliance, and its lifetime. The cost of ownership can be substantial if you have a small house and move frequently, or if it requires a lot of maintenance. (This is much on my mind at the moment, because I'm living in a small apartment --- effectively an efficiency with a storage room --- and I've moved six times in the last seven or eight months; and my refrigerator and bicycle need some serious maintenance.) These are not too hard to add in to the model, though, and you still have a strategy that guarantees you a worst-case expenditure of a constant factor of the cost of the appliance.

RAID

Another case where a constant-factor extra cost gets you something valuable is error-correction coding. For some constant factor in coding expansion, you can reduce the probability of storage errors in your data to an arbitrary degree. The simplest realization of this is "disk mirroring", where you store the same data on both disks. If one

disk dies, the other still has your data. (In theory. Right now, some of my data is on a RAID where one disk has died, and I haven't gotten around to replacing the dead disk, so I could still lose my data at any moment.)

Food buying

It's well known that you can buy sufficient nutrition for a dramatically lower cost than a normal diet. On August 16th of last year, after the national statistics bureau had created a furor by deciding on a poverty line of about AR$6 per day, I went to Carrefour on Independencia in San Telmo to price out some food and calculate the lowest-priced macronutrient-balanced diet. Here's what I came up with. None of the prices are sale prices.

	Soybeans	Salt	Sunflower oil	Flour	Total
g/day	200	5	33	430	668
kcal carbohydrates/day	54			1238	1292
kcal protein/day	280			155	435
kcal fat/day	420		297	39	756
kcal/day	754		297	1432	2483
AR$/kg	7.98	6.30	5.10	2.48	
AR$/day	1.60	0.03	0.17	1.07	2.86
US$/day (at AR$4.50)	0.36	0.01	0.04	0.24	0.64

(The whole spreadsheet, in Spanish, is at http://canonical.org/~kragen/comida.gnumeric. Due to rampant inflation, Argentine prices have gone up since then.)

The idea is that you boil the soybeans with a little salt, or maybe sprout them, and use the flour, rest of the salt, and the sunflower oil to make what are called "tortas de parrilla", a sort of unleavened flatbread which is commonly for sale in the streets here, cooked over charcoal in metal pans on shopping carts. You can see that the result is hearty; what may not be obvious is that the soybeans provide enough fiber and omega-3 fatty acids to avert what could otherwise be serious nutritional imbalances, and that their protein is of an especially high biological value, i.e. its amino-acid mix is close to optimal.

There are a couple of obvious questions about this diet:

- What about vitamins, minerals, other micronutrients?
- Don't you have to be rich to bulk-buy to get prices this good?
- Won't the lack of variety really suck?
- Isn't this going to be a lot of work?
- Isn't so much gluten going to be bad for you?

There's also the question of how much space you need for food storage.

What about vitamins, minerals, other micronutrients?

The flour (whose extremely low price is, I think, a result of government subsidies) is fortified with a variety of vitamins as required by law. But all-in-one multivitamin pills cost about US$0.02 per day, or AR$0.09 at the time, and provide all the

micronutrients we're known to need. So it's possible to solve the micronutrient problem very cheaply.

Don't you have to be rich to bulk-buy to get prices this good?

Bulk buying is indeed necessary, although all the prices above are for units of one kilogram or less. (You might be able to get better prices if you buy in *real* bulk.) It turns out, though, that even if you're actually so poor that you can't ever afford to buy US$5 of food at a time, you can work up to bulk buying with a constant-factor-worse strategy. If you're buying sunflower oil, say, by the 250mℓ bottle and getting a 20% worse price as a result, you can gradually build up a stock of sunflower oil by buying an average of, say, 10% more than you need. 250mℓ might last you 7½ days, so buy a new bottle every 6¾ days on average, rather than every 7½. Every week, more or less, you'll accumulate an extra 25mℓ of oil; after about six months, you'll be able to buy the 500mℓ bottle instead of the 250mℓ bottle, and your sunflower-oil investment will start paying dividends. Another year later, you'll be able to buy a liter at a time.

You could argue that this is not a realistic view of life in poverty; more typically you have no money coming in for a long time, like a month or six months, and then you finally get some, which you can use to buy things you've been putting off; and the critical thing to focus on is not efficiency but resiliency, i.e. making sure you have some way to get some food when you need it. This actually just happened to me, and I went to the supermarket and bought two kilograms of rice, some mayonnaise, butter, spaghetti, rice, *pears*, and so on, after living for much of the last weeks on whatever dried foods I had stored up and whatever my girlfriend bought for herself. But I'm getting ahead of myself.

Won't the lack of variety really suck?

The lack of variety is a more serious problem. It tempts you to go off-budget and eat an AR$5 hot dog or something, because you just can't face the thought of another lunch consisting of soybean pancakes. And there may be health problems caused by such a monotonous diet, even if they don't come from deficiencies of known macro- or micronutrients; for example, there might be a pesticide used on the soybeans that your body can tolerate if you're eating 200g of dry soybeans once a week, but not every day, or you might be getting some vitamin in a form that your body finds particularly hard to absorb.

The same constant-factor-worse strategy applies, though. If you can manage to buy 10% more soybeans than you're eating, then after a short time, on some shopping trip, you can buy another nutritionally similar low-priced food instead --- around here, that would be lentils, split peas, or garbanzo beans, or possibly polenta or whole-wheat flour. As long as you can keep buying a constant factor *more* than what you're eating, whether it's 5%, 10%, 50%, or 100% more, the variety of foods stored in your pantry will continue to increase, and therefore so will the variety of your meals.

(Practically speaking, you might also want to spend some of that constant factor on things other than macronutrients: spices, MSG, onions, sesame oil, herbs to plant, and so on --- things that go a really

long way to rendering otherwise unpalatable dreck edible. Tonight, for example, I ate about 100g (dry) of boiled split peas, which were AR$8.50/kg and therefore cost about AR$0.85; I added an AR$1 packet of dried soup stock, which is mostly MSG and salt but also had some basil and garlic flavor, and it made a huge difference. Ajinomoto has a line of mixed-condiment packages that are similar but even cheaper. The one I used earlier today came in a package of 12 5-gram packets for AR$4.85, I believe, so each packet costs AR$0.40.)

This also gradually reduces the risk of hunger shocks where you have nothing to eat for a few days, or weeks, because of an unexpected expense, delay in getting paid, or jump in prices. That is, if applied assiduously, the constant-factor-worse strategy eliminates some serious risks to your food budget.

Building up a stock in this way also increases the probability that you'll be able to buy food when it's on sale or even gratis.

This ignores, though, the cost of storage and the limits of lifetime --- as the constant-factor-worse appliance-buying strategy does. If you eat 200g of soybeans a day and buy 100% more, 400g (or, more practically, buy 2kg of soybeans every five days), your stock of stored soybeans will grow at 200g per day. If you somehow followed this strategy for a year, you'd have 73kg of dried soybeans stored. How on earth are you going to store 73kg of dried soybeans? And stored flour will eventually go rancid, especially if it's whole-wheat flour.

This *is* a real limit, but it's not as bad as it sounds, unless you're bouncing from one temporary accommodation to another, as I am. (In that case, maybe you should ask a friend to store your soybeans and stuff in their house.) See below about space requirements.

Independent of how consistent or inconsistent your food buying and income is, you can adjust how much of your constant factor is going into building up a stock for the future and how much is going into buying "luxury" foods that are more expensive than the bare minimum. I think the optimum fraction for building up a stock for the future, in the absence of storage and lifetime considerations, would be about half of the total. As shown above, this is feasible if you're spending more than about US$1.28 per day on food. The total investment needed to build up a one-year stockpile would be about US$234.

Isn't this going to be a lot of work?

No, soaking and boiling soybeans and frying up griddle cakes is not a lot of work. It requires planning and discipline, which can be difficult, but it doesn't take much time or toil.

Buying 10% or 20% or 100% more food than you would have been buying otherwise and bringing it home and putting it away is more work, but it's not much more work. It's about 10% or 20% or 100% more work. It's only a small constant factor worse.

Isn't so much gluten going to be bad for you?

Most people digest gluten well. Some people don't. Some people are so sensitive to it that they have to avoid it entirely or face serious health problems. Avoiding gluten increases the cost substantially, and because of the vitamin fortification, increases the risk of micronutrient shortages. From the same spreadsheet, here's my

cheapest gluten-free version:

	Polenta	Brown rice	Soy beans	Salt	Sunflower oil	Total
g/day	240	180	200	5	33	658
kcal carbos/day	691	562	53			1306
kcal protein/day	108	58	280			445
kcal fat/day		45	420		297	762
kcal/day	799	665	753		297	2514
AR$/kg	5.56	5.17	7.98	6.30	5.10	
AR$/day	1.33	0.93	1.60	0.03	0.17	4.06
US$/day (@AR$4.50)	0.30	0.21	0.36	0.01	0.04	0.90

That is, cutting out gluten increases the price of the minimal diet by about 42%, to almost a dollar a day.

Won't food storage take a lot of space?

Earlier, I said that by buying a constant factor more than what you eat, you will gradually build up a stockpile, which will allow you to buy food only when it's on sale, buy food in bulk for better prices, and keep a wide variety of stored food on hand to avoid dangerous dietary monotony. But such a stockpile takes up space. Is it an unreasonable amount of space?

I'll investigate this, plus the question of managing stored food lifetimes effectively, in depth in another post. For now, the outline is this:

These foods will last at least a year in storage. At the 600 to 700 grams of stored food per day described above, a year's supply is about 237 kilograms. That's a small enough amount of food that you could store it under your bed, in your coffee table, or possibly in shelves that already exist. So if you're not moving around a lot, it won't take up an unreasonable amount of space.

240 kilograms is enough to have about four kilograms each of about 60 different foods, so it can provide plenty of variety.

Topics

- Pricing (p. 3646) (89 notes)
- Household management and home economics (p. 3504) (44 notes)
- Strategy (p. 3734) (10 notes)
- Health (p. 3496) (3 notes)
- Nutrition

Double ended log structured filesystem

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

I use a sort of log-structured filesystem for my notebooks. I fill the notebooks in chronological order (more or less) from the second page to the last page. (The first page is left blank at first.) Everything is under some heading; the current heading is repeated at the top of every page, with the date, but sometimes there are several headings on a single page. The headings are underlined so they're easy to see looking at the page.

So I can find things by paging through the recent pages and looking at the headings. When that gets to be too much, I append a new "table of previous contents" section, under a heading just like everything else; it lists all the headings, with dates, since the last "table of previous contents". The first page contains a list of tables of previous contents, with their dates, so that I can find them relatively quickly. This allows me to find my notes more quickly by reading through the few pages that are full of tables of previous contents, rather than leafing through all the pages in the book looking for headings.

If I were a disk, which I'm not, this would be a reasonably efficient scheme for writes: regardless of how much stuff I have to write, I could append it all in a single write to the end of the currently-written data, possibly including a new table of previous contents, then update the "superblock" on the first page with a pointer to the new table. So writing any amount of data less than a notebookfull requires a seek to the end of the previous ToC, possibly a read of data following it, a write of the new data, and possibly a second seek and a second write to the superblock. Two seeks. Finding something in a notebook with three ToCs requires at most four seeks: one to each ToC, then another one to the data; if it's not listed in any ToC, you can sequentially scan for it after the last ToC.

With this scheme, there's a tradeoff (for either humans or for disks) between the amount of sequential scanning you may have to do (due to still-unrubricated items) and the number of ToCs you may have to seek to and read.

Beatrice pointed out the other day that it would be easier for a human to write the notes sequentially from the beginning of the book, while writing the ToC entries sequentially from the end of the book. This way, all the ToC entries are in a single sequential chunk, the tradeoff between maximum sequential scan length and ToC fragmentation is eliminated, and writing still requires only two seeks.

Of course she is correct, and this might be a reasonable strategy for log-structured filesystems too, although there are usually more levels of indirection: from superblock, through various levels of inodes and directories, to the actual file extents on disk. You could probably do a reasonable job by putting a B-tree of pathnames at a fixed location of the disk, and putting the inodes and data extents contiguously somewhere else. `/var/cache/locate/locatedb` is a reasonable approximation of the contents of this B-tree; on my current laptop, it's 5.3MB,

indexing 95GB of files using 596 662 inodes (i.e. 596 662 files, although `sudo locate / | wc -l` only finds 494 488 files.).

Repacking a 5–20MB B-tree when it got too large and loose would take a significant fraction of a second on a modern disk, but on my laptop would take perhaps 10–20 seconds, due to the slowness of on-CPU disk encryption. So it might be better to defragment the tree incrementally.

Topics

• Filesystems (p. 3455) (8 notes)
• Notebooks

Piano synthesis

Kragen Javier Sitaker, 2015-09-17 (updated 2017-07-19) (6 minutes)

Looking at this "grand piano" synth sample, I see that toward the beginning (19 cycles just after the attack has settled down a bit):

there's a fair bit of white (or rather pink) noise between the harmonics;

its fundamental is at 443 Hz at -13dB,

its second overtone at 883 Hz is 6.8dB lower,

its third overtone at 1326 Hz is the same (well, only 1.1dB lower),

its fourth overtone at 1775 Hz is a bit lower at 16.1dB below the fundamental,

its fifth overtone at 2223 Hz is substantially depressed at 23.1 dB below the fundamental (almost as low as the second subharmonic),

its sixth overtone at 2678 Hz is about 13 cents sharp (twice the inharmonicity of anything so far) and is 19.3dB below the fundamental;

its supposedly undesirable seventh overtone at 3129 Hz is almost 16 cents sharp and is 23.4dB below the fundamental;

its eighth overtone at 3597 Hz is 25.9dB below the fundamental, and inharmonicity continues to increase sharply to 26 cents;

its ninth overtone at 4072 Hz is 31dB below the fundamental;

higher overtones (which probably don't matter) continue to diminish in amplitude exponentially with their frequency.

Overall this spectrum looks a lot like an impulse train has had its fifth harmonic attenuated by about 6dB, has been low-pass filtered at about 9 dB per octave with a cutoff below the note frequency, and has had its overtones stretched out a bit, by about 9 cents per octave.

(It seems strange that amplitude would diminish *exponentially* with frequency, though. 9 dB per octave would give you amplitude diminishing *cubically* with frequency. Might be hard to tell the difference with this much noise, though.)

This suggests that my attempt to synthesize a piano with triangle waves didn't sound very piano-like because it was *too* low-pass filtered, not the opposite, which I guess I should have figured out just by looking at the waveform.

Looking at it about 500ms later, the peaks are much more separated (much less pink noise in between), everything is quite attenuated (the fundamental, which has moved down to the more correct 441Hz, is down to -30.9 dB, a decay of 27.9dB, indicating a half-life of about 55ms or about 24 cycles) and the rolloff is sharper: the second overtone is now 13.8dB below the fundamental, and the third 25dB below the fundamental. This is consonant with my understanding of the low-pass-filtering nature of the KS model, but it is being generated from a very realistic synthesized sample.

Looking at a somewhat noisy and MPEG-4-artifacted sample from an upright piano shortly after the attack, I see peaks at

119 Hz -46.7dB
247 Hz -19.2dB
370 Hz -28.4dB
491 Hz -34.0dB
614 Hz -40.5dB

739 Hz -32.1dB
866 Hz -34.2dB
986 Hz -41.6dB
1110 Hz -47.2dB
1236 Hz -44.5dB

It also has a notably high peak at 1867Hz at -45.7dB.

(Those are from a 2048-sample window. Later I got better numbers with a 4096-sample window but didn't update it, but in particular the first peak really peaks closer to 124Hz.)

These are roughly harmonics of about 123.47 Hz, which would be B2 in A440 pitch (although as I recall, the guy was tuning the piano a little flat) and which is nearly *missing* from the sound, which is probably why Débora says her upright piano sounds shitty compared to a grand.

If we figure that the two octaves from the second overtone 247Hz at -19dB up to 986Hz at -42dB represent the normal falloff, that's about 11.5dB per octave. The fifth overtone at 614Hz is attenuated some 8dB below this line, just like in the synthesized sample, and the seventh overtone at 866Hz is not particularly attenuated at all.

Looking at it after a second or two of decay, there's less noise between the overtone peaks, everything is of course much quieter (the first overtone is down to -61.8dB and the second down to -38.3), and the third and seventh harmonics have gotten much stronger relatively — stronger, in fact, than anything else! And there's a second subharmonic peak at 55Hz at a barely-detectable -82dB, presumably due to nonlinearities in the instrument. This is after about 850ms, or about 105 cycles. If we figure the overall attenuation is about 12dB, that's a half-life of about 26 cycles, which is quite similar to the half-life of the synthesized piano signal.

The initial attack is about 35ms long, and seems to be pretty similar across all the frequencies; it doesn't show the phenomenon I saw in the synthesized sample where the high frequencies start later.

So, overall, our piano recipe is:

• an attack time about four cycles of the fundamental;
• a half-life of about 25 cycles of the fundamental, shorter for higher overtones;
• about 9 to 12 dB per octave of low-pass filtering in the initial spectrum (compared to an impulse train);
• attenuate the fifth overtone by about 6dB over and above the basic low-pass filter;
• overtone tuning stretched by about 9 cents per octave from perfect harmonicity (the Railsback curve, although that curve shows that it's not linear) or perhaps significantly less for a better piano;
• beating among different oscillators for a given note at about 1Hz;
• to sound like a cheap upright piano, also high-pass the thing such that stuff below 250Hz or so is subject to a vicious 25dB/octave rolloff.

There's also some stuff about sympathetic strings and energy transfer among different modes of vibration, but that stuff doesn't really show up in my analyses. You can definitely hear it at times in the highest strings on the actual physical piano.

Topics

Nomadic furniture optimization

Kragen Javier Sitaker, 2019-12-15 (2 minutes)

Reading Papanek and what's-his-name's *Nomadic Furniture* I'm astonished by how much they've optimized to minimize the difficulty of cutting their furniture designs out of plywood sheets with hand tools, rulers, and T-squares. (Many of the designs use other materials as well.) If you're using laser-cutting or another automated sheet-cutting process like CNC plasma cutting or Maslow-style CNC routing, measuring is free; only the material and cut length, and sometimes the cut curvature and angles, add cost.

And, cutting prototypes out of cardboard with a box cutter, I find that indeed measuring takes an enormous amount of time, as does precision in cutting. An imprecise cut can be made in cardboard by hand in a fraction of a second, while a precise cut may require a minute. This totally kneecaps the otherwise amazing advantages of cardboard. (See Cardboard furniture (p. 742).)

So we should expect nomadic-furniture designs optimized for digital fabrication to look very different from those designed for 1970s manual construction. Maybe they would use a great deal more material, but cheaper material (though the original *Nomadic Furniture* or maybe its sequel were early champions of Frank Gehry's laminated-corrugated-cardboard furniture) or in thinner sheets. Maybe they would look like the fully-interlocking designs Mark Pauly's group at EPFL have been publishing at SIGGRAPH over the last few years.

Presumably, though, you'd like to use some kind of mathematical optimization algorithm to search for the lowest-cost design that fulfills some kind of requirements. The cost of fabrication might be one part of the objective, while others might include weight, maximum load, impact energy to withstand, rigidity, and fabrication time.

Topics

Simple system language

Kragen Javier Sitaker, 2013-05-17 (7 minutes)

Reading about Rust, and having just written a raytracer that doesn't use the heap and is almost trivially statically safe, it occurs to me that you could probably do a lot better than OCaml efficiency-wise, especially on modern hardware, without much more compilation complexity.

The ML type system has basically one kind of immutable container, which is like a variant record. Typically this is implemented as a type tag followed by a pointer for each value. Many types only have one variant. Types can be parametric and recursive.

Here are the most basic type definitions from my raytracer:

```
typedef float sc;               // scalar
typedef struct { sc x, y, z; } vec;
typedef vec color;              // So as to reuse dot(vv,vv) and scale
typedef struct { color co; sc reflectivity; char texture; } material;
typedef struct { vec cp; material ma; sc r; } sphere;
```

In memory, a `sphere` occupies nine contiguous 32-bit words:

```
cp.x
cp.y
cp.z
ma.co.x
ma.co.y
ma.co.z
ma.reflectivity
ma.texture
r
```

(`texture` is one byte, but on almost all platforms, it will be padded out, basically to align the `r` that follows it. If you used parallel arrays you could avoid this memory bloat.)

We can translate this same type definition into, say, OCaml:

```
type sc = float
and vec = Vec of (sc * sc * sc)
and color = vec
and material = Material of (color * sc * char)
and sphere = Sphere of (vec * material * sc)
;;
```

The semantics are very similar, but with this definition, a `sphere` is actually a graph of objects on the heap:

```
Sphere
vec ---------------------------------- Vec
material -------- Material             sc -- float
sc -- float       color -- Vec         sc -- float
                  sc -- fl sc -- float  sc -- float
                  char     sc -- float
```

That's a total of 11 heap allocations. If OCaml had unboxed floats, it would be only 5, but even 5 is a lot. If the boxed floats don't need a separate in-memory type tag (I haven't looked), while the others do, then this is a total of some 24 words, more than twice the count for the C approach. With in-memory type tags, the total is 32 words!

Now, this is clearly a problem with memory efficiency, but it gets worse. Half of those 24 words are pointers, and if the object survives until a garbage collection, the garbage collector has 12 pointers to traverse, where it could have had zero. And they're likely to be in different cache lines (perhaps segregated by size in different pages, to cut down on allocator space overhead), so if you have a lot of spheres (I don't), you miss out on cache locality.

In a dynamically-typed language, these allocations really have to be separate — there's no telling when you could replace the `Material` with an integer or something. And you might want some of the parts to be shared with other objects, either because they're mutable or because they're large. But these objects are neither large, nor mutable, nor do they vary in type (we might say "mode", to use the Algol-68 term for in-memory representation). And the cost of the implementation strategy is clearly unacceptable.

"Sum types", that is, types with multiple possible representations, more or less clearly need to be implemented with pointers:

```
type 'a mylist = Mycons of ('a * 'a mylist) | Mynil ;;
```

both because they can be recursive (while being finite and acyclic) and because their elements can vary in size.

There are, as I said, cases where you actually *want* aliasing semantics, either because of mutability or because of size; the C and Pascal approach to this — a type constructor like ^ such that `x^` means "pointer to an x" — seems to me entirely harmonious with an ML-style type system.

(You could lump mutability in with explicit pointerhood, since if you're going to be changing the color of a sphere frequently you're quite likely to want to be able to do that with a single operation instead of three; that would be useful if you wanted a language with a minimal number of features, like C; but in theory the two are orthogonal features.)

This is similar to the approaches taken by Golang and Rust, but Golang doesn't have full type inference.

By eliminating the majority of pointers from your heap, you eliminate the majority of work for the garbage collector, reduce your memory consumption, and probably dramatically improve locality of reference. This makes garbage collection substantially cheaper. It also makes other memory-management disciplines substantially cheaper — for example, single-ownership inherited deletion, or reference counting, are much cheaper when you have fewer pointers.

Golang's approach to arrays and pointer arithmetic seems basically sound, but it seems like it would work better if you could hoist array bounds checks in more cases.

So what's the simplest compiler for a reasonably complete language along these lines? We can dispense with Schemeish simplifications

that simplify the language at the cost of complicating the compiler: automatic closure of free variables, automatic tail-call detection, and call-with-current-continuation. OCaml further dispenses with polymorphic arithmetic and implicit arithmetic type conversion, showing that this is practical. Scheme and Forth dispense with syntax above the level of tokens, showing that this is practical. If you're going to have ML-style variant constructors, you can dispense with conditional statements, but you still need loops if you're not going to do automatic tail-call detection; and you still need a stack if you're going to support recursion that isn't tail-recursion. (And not supporting recursion seems to me to go beyond just "not complicating the compiler" into "turning implementation shortcuts into subtle bugs".)

So you need functions, arithmetic, recursion, pattern-matching (with iteration), type declarations, (mutable) pointers, and perhaps variables; and you probably also need arrays, even if you don't use them much, so you need array indexing; loops; assignments; and sequences of statements.

The simplest way to get loops, other than tail recursion, is to have a pattern-matching construct that repeats unless you break out of it. For example:

```
while (listnode) {
case Cons(a, b) ->
    listnode := b;
    total_weight += weight(a);
case Nil -> break;
}
```

Not sure that really compares well with C:

```
for (; listnode; listnode = listnode->next) {
    total_weight += weight(listnode->item);
}
```

Hygienic macros — i.e. rewrite rules — may be a convenient way to implement a fairly rich compiler fairly quickly.

Topics

Underwater energy autonomy

Kragen Javier Sitaker, 2019-11-25 (9 minutes)

Suppose you have built an undersea or other underwater habitat, like Jacques Cousteau, Tom Swift, or many popular science articles from the 1950s; but it's autonomous rather than having an umbilical to a surface support station. How do you get energy to run and repair your base?

The usual suspects: fission power, solar, fuel, buoys; and their drawbacks

Fission power is clearly the best option, but suppose you can't use fission power for political reasons.

If you have sunlight, you can of course use solar panels, although the water will absorb some of it. (See Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) for notes on the absorption spectra of different kinds of water.) If you're under enough water to make it difficult to see you from the surface, or from space, you will get only a tiny amount of sunlight energy.

You can periodically receive shipments of fuel, but burning fuel requires an oxidizer such as oxygen, so you need shipments of oxidizer too. As described in The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271), the best option to me seems to be sodium chlorate, but even so, you end up with only 5 MJ/kg when you include the weight of the sodium chlorate rather than the usual 43 MJ/kg we're used to on land.

By floating a buoy on the surface, it's possible to gather surface solar energy, and also some amount of wave power, as well as sucking in air for human respiration and perhaps fuel combustion. But the buoy is visible, which may be undesirable, and also exposed to damage from heavy seas, ship collisions, oxygen, and sunlight.

Underwater "kite" wind power

I think the best option in many places is analogous to kite wind power, using a reconfigurable-geometry buoy floated some distance below the surface but above the bed; when configured for high-drag geometry, it pulls on its tether, which is gradually let out, generating power; when the tether is nearly exhausted or it's too close to the surface, it's reconfigured for a lower-drag geometry and reeled back in at a much lower energy cost than what it generated when it was being let out. This is much easier than the equivalent task in the air because the velocities are lower, the forces are much higher, the rope length and therefore snapback potential energy is lower, and it's easy to reconfigure the "kite's" buoyancy for a particular altitude under the water.

(I think passive altitude control via buoyancy control is much more difficult in water than in air; the density of air varies sufficiently with pressure for a balloon to hover within tens of centimeters of a constant height as the rubber holds its own volume relatively constant; on the other hand, in a submarine, water density varies only

very slightly with pressure, but higher pressure will tend to collapse your swim bladder and reduce your buoyancy further.)

Cables under tension can carry an amazing amount of power. Consider gel-spun UHMWPE, with its 2.4 GPa yield stress (see Dyneema (p. 123)). At 10 m/s, a snappy but not insane speed (22 miles per hour, in medieval units), 2.4 GPa is 24 GW/m², which is 24 kW/mm². According to Induction kiln (p. 2352), AWG20 [copper] wire can safely carry 5 amps and is 0.812 mm in diameter (not counting the insulation), or 9.7 megawatts/volt/m²; so, reaching the same 24 GW/m² with it requires 2500 volts. At 100 m/s, the UHMWPE cable carries 240 GW/m² = 240 kW/mm², which requires 25 kV in the electrical wire. Copper weighs 9 g/cc, about 9× what UHMWPE weighs.

Lower cable speeds require proportionally more tension and thus more cable thickness to deliver the same power.

Regular air kites

A regular air kite might be better in some sense, particularly at those underwater sites, such as those at the bottom of medium-sized lakes, that have no significant water currents. It could be made of a hydrophobic material, floated to the surface of the water by slight inflation, and then floated to kite height by further inflation with hydrogen or helium. Once in the air it can expand substantially to a size much larger than the underwater structure.

In The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271), it is proposed to store some 500 MJ of energy in half a tonne of Li-ion batteries to provide a month's worth of 180-watt autonomy without access to air, in a habitat equipped with, among other things, 25 kW of winches. Suppose that the kite pulls 100 kilonewtons, which is a bit over ten tonnes (42 mm² of UHMWPE, or maybe 100 mm² = 1 cm² to have a safety factor), and rises to a height of only 200 m in order to avoid interfering with aircraft; and suppose that the wind at that height is generally 20 m/s. That provides a megawatt or two of power, enough to fully charge the batteries in five to twenty minutes, although it requires some kind of apparatus in the lake habitat capable of storing megawatts of power.

So, like a ball python, this underwater habitat could lurk unobserved at the bottom of a still, dark lake, reaching out to replenish its energy from its environment once or twice a month; but instead of swallowing a rat, it silently flies a kite for a few minutes in the middle of the night.

Hovering submarine assemblages

As I described previously in "hovering kite assemblages" (?) a flying machine large enough to simultaneously be at altitudes with winds in different directions can use those differing wind directions to maintain tautness in the tethers between its various otherwise disconnected parts, to control its direction of movement, and to generate energy to power it, in particular keeping it from falling out of the sky. Similarly, a group of tethered submarines at different depths could harness the differing directions of deep-sea currents at those depths to generate energy and control their direction of movement. (Lift in that case is unnecessary.)

In a sense, that's what a sailboat is doing: harnessing energy from the relative movement between the air and water to move in any direction, including tacking upwind.

Lift/drag calculations

The hydrodynamic force of a fluid flowing past a body can act on it in almost any direction; in general it is proportional to the density of the fluid, the square of the impact velocity, and the cross-sectional area of the body perpendicular to the direction of the flow, and has an additional factor which I think is actually the "drag coefficient". Conventionally it's resolved into a scalar in the direction of the fluid flow called "drag" and a two-dimensional vector perpendicular to it called "lift".

In a constant flow in the absence of any other applied force, drag gradually accelerates the body to the velocity of the flow, causing the hydrodynamic force to disappear. This leads to an interesting phenomenon where the power produced at zero velocity and maximum force is zero, and the power produced at zero force and maximum velocity is also zero; maximum power is somewhere in between, specifically at one third of maximum velocity. (Unless the drag coefficient changes, which it does.)

Because drag is complicated --- the coefficients vary with flow speed and viscosity, and not even continuously or monotonically --- I hesitate to pronounce anything too pompous about this, but *very roughly* this suggests that water currents tend to produce about a thousand times as much force and power as wind of the same speed; for wind to provide the same force, it needs to go 32 times as fast, but to provide the same power, it only needs to go 10 times as fast. Or, changing a different variable, wind needs a thousand times as much area to press on to be equivalent to a water current of the same speed.

Topics

• Independence (p. 3520) (63 notes)
• Energy (p. 3438) (63 notes)

Air conditioning

Kragen Javier Sitaker, 2007 to 2009 (21 minutes)

Why This Is Important

According to the EPA's figures, 75% of residential energy use in the US in 2001 was for heating things up and cooling them down in fairly brute-force fashions --- of 9.86 quadrillion BTU, only 2.40 quadrillion were spent on anything else. [0] And essentially all of that is unnecessary.

Only about a fifth of total energy consumption in the US is residential [1], so this 75% is only about 16% of the total; but I think it is only slightly exaggerated from usage patterns in the US economy as a whole. I estimate that 50% of overall US energy consumption is spent on indoor climate control.

I don't have good numbers for the rest of the world. I assume they're roughly similar.

As global warming increases the amount of severe weather people must cope with, the importance of effective indoor climate control will increase; and fighting global warming will require the reduction of energy consumption. Additionally, sufficiently effective indoor climate control can render uninhabitable regions habitable and, if scalable to large buildings, enable the cultivation of a wider range of crops than the local climate would normally allow.

Argentine Air Conditioning

Here in Argentina, air conditioners are rated in "frigorías". One frigoría is one kilocalorie per hour, which is about four BTUs per hour. They recommend 50 frigorías per cubic meter. (This is a little strange, since you'd think you would pick your air conditioner capacity by the amount of heat that comes into your house per hour, rather than the amount of heat your house can hold. But I digress.)

Our apartment is 80 m², and about 2.75 meters in height, which means it contains about 220 m³, and therefore should need about 11000 kcal/h of cooling capacity. This is a bit of a problem for two reasons:

• Air conditioners are expensive more or less in proportion to their cooling capacity, and a 2250-frigoría unit costs $1550 ($0.69 per frigoría), while a 3000-frigoría unit costs $2040 ($0.68 per frigoría). The largest off-the-shelf unit I found was 8000 frigorías, and it cost $5000 ($0.63 per frigoría). At this rate 11000 frigorías would cost us $6930 (US$2200), which seems like a lot of money to me. For example, it's more than three months' rent.
• Air conditioners also use energy more or less in proportion to their cooling capacity, with variations in the neighborhood of 30%. The 8000-frigoría air conditioner uses 3540 watts of electrical power, which would be 16 amps if it's properly power-factor corrected. (I know that's not the amount of heat it's removing, because a kcal/h is only about 17% bigger than a watt. I'm a little confused because I thought that it normally cost more than a watt to remove a watt of heat, but all the air-conditioning units' datasheets claim that they

remove a little over two watts of heat per watt of power used.) So 11000 frigorías should be 4868 watts of electrical power, or a little over 22 amps, more than half of the total of our main circuit breaker. If we assume a duty cycle of 40% during the summer, that's 1947 watts on average during the summer. If we were paying US$0.10 per kilowatt-hour, which is in the neighborhood of what the price would be if residential power weren't heavily subsidized, 4206 kWh would cost us US$420.60, or AR$1324. The current residential electrical rates are about $0.04 per kWh, which would bring the price to only $168.20, but that's still enough of a cost to want to minimize it.

Water Tanks to Sink the Heat During the Day: What Size?

So I've been thinking some more about my 'desert cool and water economics' kragen-tol post from more than a year ago. Suppose we wanted to be able to sink 11000 kcal/h of electrical power for 40% of each day during the summer into some kind of heat storage tank (say, full of water) and then radiate it into space at night. How big of a tank would we need, and how much area? How much surface area would we need for heat-exchanger coils to suck 11000 kcal/h of heat out of the air during the day?

The first question is the easiest one. In the last few days, the highs have been around 28°C and the lows have been around 18°C. Suppose we want to lower the temperature of the air from 32°C to 22°C, which is 10 K of difference. So the water must start out cooler than 22°C and end up cooler than 32°C. Suppose we can start with water at 16°C and end with it at 30°C. Then we can get 14 kcal of heat out of the air per liter of water that flows from the cold tank through the heat exchanger into the hot tank. So 11000 kcal/h would be 790 liters per hour. I've hypothesized needing this cooling power for 9.6 hours (40%) out of each day, which means that each tank would need to hold 7580 liters of water, which is 7.58 metric tons and 7.58 cubic meters.

If we kept these 7.58 metric tons of water indoors, and didn't need any extra space for insulation, we would need 5.5 square meters of floor space for them, or about 2.3 meters square. That's about 7% of our total floor space, and that 7% should be no more and no less constant across dwellings than the figure of 50 frigorías per cubic meter.

Water Tank Size Is Less Sensitive To Temperature

The same amount of water can sink more heat, and sink it more quickly, when the difference between the air temperature and the cool-water temperature is greater. If the cool-water temperature stays at 16°C and I want to cool the air from 28°C, I can only sink, at most, 12 kcal per liter of water. But if the air temperature is 40°C, I can sink 24 kcal/L, twice as much heat for the same volume of water.

So while a conventional air conditioner needs to be sized for the amount of heat it needs to extract on the hottest days, and therefore cares a lot about how hot they are, the design of such a reservoir device should be relatively insensitive to the maximum temperature.

So, even if the 16°C number is correct, I might be able to get by with a much smaller amount of water, simply because the 11000-frigorías number is only relevant on the very hottest days --- if even then.

The Exhaust Heat Exchanger

Cooling the water at night can happen in essentially three ways: evaporation, conduction, and radiation. I think evaporation is limited in its applicability (it won't help colonize the Grand Erg Occidental, and it will get expensive if water does) although being able to use brackish water might broaden its applicability.

This leaves conduction and radiation. Radiation has the major advantage that, on dry nights, the heat sink is the cosmic background radiation at a temperature of about 3 kelvins, so it might be possible to cool the water quite cold, perhaps even to freeze it. (This probably involves using an intermediate heat transfer fluid with a lower freezing point, but it's tractable.)

The basic radiation law is Stefan's Law or the Stefan-Boltzmann law:

```
Q/t = e{sigma}A(T_hot^4 - T_cold^4)

e is the emissivity, a number between 0 and 1 indicating how close
to an ideal black-body radiator the object in question is (at the
relevant wavelengths).  I think it's (1 - albedo) at the relevant
wavelengths.
Sigma is the Stefan-Boltzmann constant: 5.67e-8 W/m²/K^4
Q is the heat transferred.
t is time.
T_hot and T_cold are the temperatures of the hot and cold bodies,
measured in kelvins.
```

So if our e is, say, 0.75; T_hot is at least 16°C (289 K); T_cold is insignificantly small; t is, say, 8 hours; and we want the Q to be 100Mcal (11000 kcal/h * 9.6 h); how much emitting area do we need?

100Mcal is 420 MJ, so the required heat dumping rate is 14.5kW; T_hot^4 is 7.0e9 K^4; so we have

```
14.5kW / (0.75 * (5.67e-8 W/m²/K^4) * 7.0e9 K^4) = A
m² * 14.5e3 / (0.75 * 5.67e-8 * 7.0e9) = A
m² * 14.5e3 / (0.75 * 5.67 * 70) = A = 49 m²
```

So I'd need about 7 meters square of roof space to beam the heat from my little apartment back out into space at night, plus enough pipes and aluminum to keep it all warm. And on cloudy nights, it wouldn't work. Note that 49m² is more than half the floor area of the apartment.

Conduction, on the other hand, requires little extra equipment (just a fan and some flaps to direct air from the outdoors through the same heat exchanger used to cool the indoors) and is known to be feasible.

Cost of Space

We can estimate the "cost" of losing the space at 7% of our rent: for us, $150 per month, or $1850 per year. So it costs us $40 per month during the life of the thing, and as long as the electrical rates are so deeply subsidized, it would actually cost us almost all of the $150 amount. We're not in a particularly expensive area of Buenos Aires, but areas outside the Capital Federal are cheaper still. So for people in areas that cost less per month per square meter, it might save them money each month. Even for us, it might cost less up front than a $7000 refrigerative cooler, at least if it doesn't break the floor and can be adequately insulated without building anything really expensive.

Heat Exchanger Area

I don't really have a clue what size of heat exchanger I would need. Presumably it would be a few times bigger than the heat exchanger that an air conditioner would use, because the temperature difference between the coolant and the air is smaller.

Tank Insulation

The hot water tank must retain its heat until it can be exhausted into the great outdoors at night, if it is inside; and the cold water tank must retain its cool until it can be exhausted into the indoors during the hot day, if it is outside. In the winter, their roles would be reversed. (Beatrice suggested reversing their roles during the winter. I need to think more about this.) Insulating materials like fiberglass typically have thermal conductivities around 0.04 W/m/K. If a cubical tank contains 5.28 cubic meters, it's 1.74 meters on a side, and therefore has 18.2 m² of surface area, so that's around 0.72 m W/K, and I've been hypothesizing about a 20 K temperature difference, so that's 14.4 m W. We need to divide that by enough centimeters of insulation that the resulting uncontrolled heat flux is small compared to the average 4400 kcal/h (=5100W) heat flux that the thing is designed to control. If "small" means 5%, that's 510W, which means we need 2.8 cm of insulation.

3 cm of insulation around cubical tanks seems like it might be a much more reasonable proposition than welding up some giant dewar flasks for the thing, which would presumably need to be round instead of some more convenient shape. (Right?) That's roughly a cubic meter of fiberglass (or whatever: cork, cotton, felt, "mineral wool", styrofoam, are all in the same ballpark; silica aerogel could cut the space by half, and straw would double it) to insulate all ten cubic meters of reservoir.

Phase Change Materials Instead of Water

Water has a very high specific heat, so using most other materials for the hot and cool reservoirs in place of water (air, say, or iron, or concrete) would increase the weight required, rather than decrease it. (And solid materials have the additional problem that they're kind of inconvenient to move between the warm reservoir and the cool reservoir.)

However, materials that change phase in the relevant temperature range --- say, solid to liquid, or liquid to gas --- have a much higher effective specific heat than any substance. If we were trying to cool

the apartment from by heating water to 1°C from -1°C, for example, we could dump half a calorie per kilogram of ice going from -1°C to 0°C, then 80 calories per kilogram as it melted, then a calorie per kilogram of water going from 0°C to 1°C. This allows you to reduce the weight of water you need by more than a factor of 40. If previously you would have needed five metric tons, now you only need 123 kg.

(Air conditioners in the US are often rated in "tons", which are tons of ice melted per day. Before refrigerative air conditioners were invented, you would cut ice out of frozen lakes in the winter and store it in an "icehouse", insulated by straw, all year round. So this idea of using phase-change materials as heat reservoirs is nothing new.)

(Note that this works even if you want to cool the air from 25°C to 20°C.)

But this only works if the relevant reservoir can be induced to keep its temperature at the melting or boiling point of the reservoir fluid, or oscillate back and forth across it. If you can dependably get access to a reservoir of below-freezing cold, you can use this approach with water; otherwise, you might have to use a material whose phase change temperature is somewhere more convenient. (If it can be actually inside the range where you want to maintain the air temperature, you don't need tanks and stuff. You can just put the stuff out where it can conduct heat to the air, like in the wall or on the coffee table or whatever.)

There are all kinds of research going on on phase-change materials as lower-mass heat reservoirs. See, for example, J.R.Gates's phase-change material home page:

http://freespace.virgin.net/m.eckert/index.htm

They also have this nice thermostatic property, which has been used to calibrate thermometers for centuries, that they tend to heat up things that are colder than their phase-change temperature, and cool down things that are warmer.

This is Nothing New; Why Was It Rejected Before?

Obviously, indoor climate control with insulated heat reservoirs, passive solar design, phase-change materials for reservoirs, and so on, is nothing new. Adobe houses, stone castles, dugout houses, soddies, caves, iceboxes using ice stored since the winter in the icehouse, walls shared between houses to prevent heat loss, and so on, all go back generations if not millennia or longer. But they were all abandoned in the developed world over the last few decades. Why were they abandoned, and what makes me think they're worth recovering?

They were abandoned for several reasons:

- Fashion. In "Pioneer Pride", I seem to recall reading my great-uncle abandoned his Western New Mexico dugout because his new wife didn't feel that it was a "proper house." It wasn't what she was used to, what she'd grown up with. This is the problem the Viridians are tackling; green design has to be hip in order to get uptake.
- High cost of labor. The traditional techniques involve a lot of heavy material hauling, and most of it isn't easily automatable. (Adobe

bricks are still mixed by hand, made by hand in wooden frames, then stacked by hand to dry.)
• Inflexible construction techniques. You can build a wooden balloon-frame house anywhere you can haul the wood, but you can only build a cave house where there's a cave.
• Low cost of energy. As we've eagerly depleted the Earth's fossil-fuel endowment without much concern for the environmental effects, we've kept the cost of that energy quite low, so it's seemed sensible to use cheaper construction techniques and save space rather than spend money to save electricity. And electricity subsidies like those in Argentina, while they might be helpful for development, make this worse.

I think there are a number of new developments that make these approaches relevant again:

• Fashion. Energy conservation, and green design in general, are trendy in much of the world. Hopefully they'll stay that way for a few decades.
• More extreme weather. As global warming increases the extremes of heat and cold we must survive, even people who cling to 1950s climate-control technologies will benefit by complementing them with more efficient approaches.
• Better materials. Even glass and steel are major improvements (in cost and flexibility) over many traditional materials, but we also have stainless steel (making large Dewar flasks possible), silica aerogel, PVC, fiberglass, styrofoam, fiberglass-epoxy composites (making large transparent water tanks possible), low-emissivity glass, stainless steel, aluminum, and copper pipes and fins (making heat exchangers efficient), rammed earth, straw bales, and on and on. In this case, the large amount of thermal mass (in the form of water in a PVC tank insulated with styrofoam) occupies a tiny fraction of the size of adobe walls.
• Better designs. Dewar flasks, window overhangs, double-paned windows, and so on.
• Less-labor-intensive construction techniques. We can now excavate and build earth berms with small-scale earthmoving equipment.
• Inexpensive computer control. It's now inexpensive and reliable to have a microcontroller turn a water pump or two on and off a few times a day, monitor some pressure sensors, control some fans, open or close some air control flaps, open or close curtains or louvers to regulate the amount of solar heat gain, and raise an alarm in case of pump failure or pipe leakage. This means that it should be straightforward to control air and radiant temperature indoors as effectively as with a traditional climate-control system.
• A possibly rising cost of energy. Certainly the cost will rise here; it may double or quadruple or more.

Related Buzzwords

Traditional adobe construction. Traditional dugout construction. Earth berms. Trombe walls. The German Passivhaus program. Seasonal thermal stores. Earth-berm construction. Straw-bale construction. Dewar flasks. Earthships. Thermal energy storage in general, in particular, "full storage systems" that run the air conditioner chillers only at night. John Hait's Passive Annual Heat

Storage. Water walls. Isolated solar gain. Annualized geo-solar. Superinsulation. Heat recovery ventilation. Ground source heat pumps. The SHPEGS Solar Heat Pump Electrical Generation System.

References

[0] "Energy Consumption and Expenditures RECS 2001", from the US Department of Energy Residential Energy Consumption Survey:

http://www.eia.doe.gov/emeu/recs/recs2001/detailcetbls.html#total In particular, see page 3 of the total consumption PDF: ftp://ftp.eia.doe.gov/pub/consumption/residential/2001ce_tables/enduse_consump2001.pdf

Of the 9.86 quadrillion BTU, 2.21 quadrillion are spent on air conditioners and refrigerators, which someone might argue aren't really part of "climate control".

[1] US Department of Energy Energy Information Administration (DOE EIA) Monthly Energy Review, December, 2007

http://www.eia.doe.gov/emeu/mer/consump.html In particular, the third page of section 2, "Energy Consumption by Sector", p.27: http://www.eia.doe.gov/emeu/mer/pdf/pages/sec2.pdf

Currently this shows a 9-month total for 2007 of 16.43 quadrillion BTU for the residential sector, out of 76.16 quadrillion overall; there's another 13.84 quadrillion attributed to the "commercial" sector, which has roughly similar seasonal and source energy usage patterns, suggesting that a large part of its energy consumption is also devoted to indoor climate control.

However, that total also includes 30.87 quadrillion BTU of energy used by the "electric power sector" --- 40% of the total --- and presumably that usage is roughly proportional to total usage of electricity.

So my best estimate is that (0.75 * (16.43 + 13.84) / (76.16 - 30.87)) = 50% of US energy consumption is devoted to climate control.

Topics

Immutability-based filesystems: interfaces, problems, and benefits

Kragen Javier Sitaker, 2019-02-08 (updated 2019-03-19) (23 minutes)

Git provides a sort of immutable Merkle-graph filesystem which saves all previous versions of all files; every version of every file ("blob") is identified with its SHA-1 hash. It's hierarchically structured, with a similar property for subdirectories ("trees"): iff two subdirectories, or two versions of the same subdirectory, are recursively identical, their hashes will match. This is true across time and across hosts.

You could imagine a variety of functionality based on a filesystem with such an interface. Git provides some of it: for example, it eagerly eliminates duplicate files, so that storing a hundred identical versions of a subdirectory tree in Git costs no more than storing a single one, and storing a hundred nearly-identical versions costs only slightly more. (Storing a hundred nearly-identical versions of a file is optimized using a different, heuristic, mechanism). It provides rapid access to previous snapshots of the filesystem (though, unfortunately, usually through a stateful interface.) It provides relatively efficient synchronization of Git repositories over the network, using the hashes to identify which data needs to be transferred, and what data the receiver already possesses and can thus use to compress the data being sent.

Possible uses

But in addition to duplicate-elimination, historical snapshot access, and efficient network synchronization, such an interface can be used for a variety of other purposes, including consistent-snapshot backup, software downgrade, lazy file reading, userland hashing, polling for generalized change notification, build systems, and maybe even transactional filesystem updates.

Consistent-snapshot backup

Consider backing up a database server, for example. Database servers are designed so that, if the power fails, after a reboot, they will not have lost any data — under some reasonable assumptions about write ordering which are sometimes violated by actual disks, resulting in permanent data loss. So you would think that backing up the files or raw disk partitions they store their data in (sometimes called "tablespaces") would enable you to recover to the point in time when the backup was taken.

Unfortunately, backups are not taken at a point in time; it takes a certain amount of time for the backup process to read from one end of the tablespace to the other, during which time the database server may be writing data. So the data at the end of the tablespace will be of a more recent vintage than the data at the beginning, producing inconsistencies. This can result in useless backups, a fact discovered by many novice database administrators, to their dismay, only after a data-loss disaster.

(SQLite and Berkeley DB have instructions in their documentation

providing an order in which active database files can be backed up safely, but for many databases no such order exists.)

One of NetApp's selling points in 1996 — at the point that their entire website still had a bright yellow background — was that you could take backups of databases. You put the tablespaces on one of their fileservers (then marketed as "FAServers", now "filers") and, instead of backing up the live database, you make the fileserver take a snapshot and then do the backup from the snapshot, not from the constantly-mutating firesystem.

Such a backup from a consistent snapshot would be a useful application of any snapshot-capable filesystem. If you can take the snapshot of the underlying block device rather than the filesystem, for example using AWS EBS's snapshot tools, that may work as well, but it's vulnerable to a certain very common class of filesystem implementation bugs — it's similar to recovering the filesystem after a system crash, and if the filesystem doesn't actually provide the sequencing guarantees it claims to provide, the tablespace could be inconsistent anyway. Also, successfully mounting the snapshot of the live filesystem often requires the ability to modify it to correct inconsistencies in the filesystem itself (e.g., replaying filesystem journals) which requires copy-on-write functionality if it is to both be fast and not violate the integrity of the snapshot.

Git doesn't provide this functionality because it doesn't offer an API that any database servers use to store their data.

Software downgrade

Upgrading software is a constant necessity, both due to the insane epidemic of software security holes and to provide new functionality users want. But upgrading a system you depend on is always somewhat risky: the upgrade may break it, in subtle or obvious ways. Being able to reset the whole filesystem to a previous snapshot can reduce this risk greatly, as it does on AWS EBS, and I think this functionality can be provided more cheaply at the filesystem level than the block device level.

Lazy file reading

When I open a large file in Emacs, it copies the entire thing into virtual memory — it doesn't necessarily have to fit into RAM, but at least into swap by way of RAM. This is necessary because, while I'm editing the file, something else may be modifying the file while Emacs has it open, and in such a case, Emacs wants to be able to detect the conflict and offer options for resolving it. If it only read the file into memory lazily, as parts of it were requested, it could not do this. Even if I'm just scrolling through the file, you could imagine a modification in the middle of the file that results in "torn reads" — where one data block is from before the modification, while the following data block is from after it, and their juxtaposition results in inconsistent results.

It is for this same reason that incorrectly upgrading active shared libraries — which are memory-mapped by ld.so — can result in core dumps.

If Emacs could, instead, open a *current snapshot* of the file, trusting the filesystem to not permit modification of any of the data in the snapshot instead of eagerly making a copy of the whole thing, it could

open files of any size instantaneously.

As described in "Proposal for the Implementation by Xanadu Operating Company of a Full-scale Semi-distributed Multi-user Hypertext System", 1984-04-25:

> Separate processes which request the retrieval of the same orgl at the same time are each given different berts which reter to the orgl. Associated with each orgl is a count of the number of berts which currently refer to it. If one of these processes then makes an edit change to the orgl, a new orgl will be created. The process's bert will be made to refer to the new orgl and the old orgl's reference count will be decremented. By this means, the other processes will not "see" the change, and their berts will still refer to the same V to I mapping as previously. Any information about the orgl's state which the other processes might have been keeping externally will not be invalidated by the one process's edit operation.

Also, Emacs depends on filesystem metadata modification to know if modifications have happened since it read the file; this is not entirely reliable, particularly on filesystems whose modification time granularity is 1000 or 2000 milliseconds. Checking a filesystem-maintained hash of the file would be considerably safer.

Userland hashing

If you want to know if any of the files installed by any of your Debian packages have changed, you can run the `debsums` command to compute secure hashes of them, comparing them against the hashes in the original packages. Unfortunately, this is slow, because it has to read all the file data. If the filesystem provided a trustworthy hash — in 2019, maybe we'd choose SHA-256 rather than SHA-1, among other things to preserve a 2^{128} security factor even in a postquantum future — this would be instantaneous, as long as the filesystem's hashes themselves were not out of date.

Additionally, though, if you wanted to maintain a list of the hashes of some files using some other algorithm (for example, BLAKE2B), you could associate them with the filesystem's file hashes rather than merely with filenames. When a file's filesystem hash changed, you would know that you needed to recompute its BLAKE2B hash. This is in a sense a special case of the "build systems" item below.

Alternatively, the filesystem itself could maintain potentially more than one hash for each file.

Polling for generalized change notification, like inotify

The inotify API in Linux allows a running process to be woken up asynchronously when a file is changed or when any file in a directory is modified; for example, `tail -f` uses this to display new lines appended to a logfile immediately, rather than polling once per second, and GUI file managers use inotify to keep their windows up-to-date with the filesystem — so that newly created files are displayed, files whose contents have changed get updated, and deleted files disappear. The only way to do this with the standard Unix API is to poll periodically, wasting energy in the case where nothing has changed, and still delaying change propagation by up to the polling period.

However, inotify is still somewhat limited. It only applies at a per-directory level, so starting to watch for changes throughout an entire subtree of the filesystem requires a recursive traversal of the subtree with a couple of system calls per directory. And it has no way to determine whether changes happened when the process wasn't

running — the GUI file manager has no way to cache its display in case it's restarted without a directory having been changed, but must laboriously re-examine the same data it examined on the previous execution.

By contrast, if there were a hash for the directory subtree and for each file within it, the file manager could validate its cached display against these hashes when it's restarted. And a filesystem watcher doesn't need to recurse through the entire tree to set up notifications — although it may need to recurse to track down the source of a notification.

Build systems

The `make` system does several things, but one big one is caching of previously-executed computations. Supposing that your compiler is deterministic (a difficult problem in modern systems, though one being tackled by Debian's Reproducible Builds project and by Nix and Guix), it will always produce the same output files given the same input files and options. `make` relies on filesystem timestamps for this, but a more reliable approach would use secure hashes of the file inputs instead, or other handles to immutable versions of the files. (And a filesystem that records the files accessed by a build step, by interposing a check on open() and similar system calls, can provide a more reliable dependency set, not depending on compilers and Makefile authors to specify the dependency set.)

Compilation steps can also potentially depend on the contents of directories, and this introduces a potential problem. For example, I just ran a compilation command with GCC which did, among other things, the following sequence of system calls, with others interspersed:

```
[pid 25870] open("../yeso/time.h", O_RDONLY|O_NOCTTY) = -1 ENOENT (No such file o
r directory)

[pid 25870] open("/usr/lib/gcc/x86_64-linux-gnu/5/include/time.h", O_RDONLY|O_NOC
TTY) = -1 ENOENT (No such file or directory)

[pid 25870] open("/usr/local/include/time.h", O_RDONLY|O_NOCTTY) = -1 ENOENT (No
such file or directory)

[pid 25870] open("/usr/lib/gcc/x86_64-linux-gnu/5/include-fixed/time.h", O_RDONLY
|O_NOCTTY) = -1 ENOENT (No such file or directory)

[pid 25870] open("/usr/include/x86_64-linux-gnu/time.h", O_RDONLY|O_NOCTTY) = -1
ENOENT (No such file or directory)
[pid 25870] open("/usr/include/time.h", O_RDONLY|O_NOCTTY) = 4
[pid 25870] fstat(4, {st_mode=S_IFREG|0644, st_size=13543, ...}) = 0
[pid 25870] read(4, "/* Copyright (C) 1991-2016 Free "..., 13543) = 13543
```

An interposition-based system that concluded that this compilation step depended on the contents and filesystem metadata of `/usr/include/time.h` would be correct, but it *also* depends on the *nonexistence* of `/usr/local/include/time.h`, among other things. If GCC had found `/usr/local/include/time.h`, it wouldn't have continued on to read `/usr/include/time.h`.

But it would be very unfortunate for the build step to be re-executed because the contents of `/usr/local/include` had changed, or worse, because `/usr/local`, `/usr`, or `/` had a change somewhere beneath them.

Fortunately, GCC didn't call `getdents()` (at least in this case), so we can automatically define the dependency more narrowly to *just* the files it specifically probed for — the rest of the directory's contents were not relevant.

Other systems whose results we might want to cache, such as the `updatedb` part of `locate`, might indeed walk an entire filesystem tree using `getdents()`. Such systems would need a bit more surgery to make their results usefully cacheable — they would need to somehow split it into separable transactions per subdirectory tree.

Transactional filesystem updates

ACID transactions are contagious; like other acids, they tend to splash on unexpected things and produce unexpected results on them. If your filesystem can provide ACID transaction updates, then you can expand the scope of whatever transactions you're doing in your program to include the filesystem. Some form of this is necessary if you want your transactions to be actually durable, rather than just "ACI", longing after the D. Providing processes in other transactions with a view of all the files as they were before your transaction began to modify them is very similar to simply providing them with snapshots from before the transaction, but it requires at least some kind of atomic validate-and-commit operation. If you're going to participate in two-phase commit at that level, it further requires the ability to lock the validation in place while you're waiting for other participants in the transaction, which means potentially denying other writers to the filesystem.

Accessing the same mutable data within transactions and also outside of transactions has some pitfalls! Beware!

Implementation issues

Given the above list of benefits, what problems do we need to solve to implement the system? We need to decide on the granularity of hashing, deal with the possibility of hash inconsistency, and figure out what to do about hashes being broken, foreign filesystems, and space exhaustion.

Granularity and splitting

As explained in more detail below, on Flash, unchanged data is necessarily copied from one place to another periodically, offering the opportunity to hash it, but maybe not an entire file at a time. Also, processes like database servers expect to be able to efficiently write individual sectors, or at least tracks (≈100 KiB), of a potentially much larger file; such an operation shouldn't have to reread the entire file to compute the new hash. So probably you want to hash data in chunks close to the size of a disk block, in the range of 512–262144 bytes, and build some kind of Merkle tree from that for larger files. BitTorrent does this, for example, hashing each "piece" of a torrent separately; the `btih:` schema in Magnet links then uses the hash of the torrent file as a whole.

For text files and other files in which blocks of data might move

around, it's desirable to be able to draw the boundaries of the Merkle tree nodes in a way that can recognize a mostly-unchanged file, even if something has been inserted or deleted. Silentbicycle's "jumprope" data structure used in his "Tangram" filesystem and the splitting system used by Avery Pennarun's "bupsplit" system are two variants on this theme.

How can the filesystem's internal hashes get out of date?

The filesystem's hashes can get out of date via bit-flip errors on the disk, via modifications of the disk's data that don't go through the filesystem (for example, if you edit the disk with a hex editor), or via bugs in the filesystem, so they need to be checked, as `git fsck` does.

If the filesystem is running on raw Flash that suffers from read disturb, all data needs to be copied to new Flash blocks periodically; this affords an opportunity to check its hashes in the process. For this to work, the data probably needs to be hashed at a finer granularity than an arbitrarily-large file. Similarly, on Flash, live blocks periodically need to be copied from mostly-empty pages into new pages so that the old pages can be erased, affording a similar opportunity.

Cryptographic agility

In retrospect, it looks like baking SHA-1 into every persistent structure in the Git universe may have been a bad idea — SHA-1 is too short to resist well-resourced attacks with Grover's algorithm, if that becomes feasible, and it has shown some worrisome signs of weakness in the last couple of decades, with some theoretical collision attacks published.

For most of the applications cited above, user processes don't need to access the actual hash values used to index the on-disk data; they can manipulate them through opaque handles like file descriptors, referring either to specific versions of files or to the time-dependent value of "the current state of such-and-such a filesystem entity" (from which the current version can be obtained) — although equality comparison must also be provided. Duplicate-data elimination, previous-snapshot access, consistent-snapshot backup, lazy file reading, and run-time polling for change notification could work with such opaque identifiers. However, you need some kind of stable serialization of these snapshot identifiers for software downgrade, change notification across process restarts, and build systems (across process restarts); for efficient network synchronization and userland hash verification, you additionally need to be able to get the results of a *specific* hash algorithm, because if your filesystem is giving you SHA-1 hashes and the server has upgraded to BLAKE2B, or your filesystem has upgraded to SHA-256 hashes but your package manager only uses SHA-1, you're in trouble.

There are a couple of ways to handle this. You could have the filesystem internally compute a non-constant set of hashes — perhaps the old version of a file has a SHA-1 Merkle tree hash with a blocksize of 8192, while the new version uses a blocksize of 131072, so that if you want to compare them you need to ask the filesystem to compute the other hash for one of the two, which it will store thereafter.

Alternatively, you could fix the algorithm for a given filesystem,

and allow userland processes to use the usual caching approaches to compute hashes using their favored algorithms when they need them, indexing them to individual snapshots if desired.

Compatibility with foreign on-disk filesystems

Suppose your operating system provides such an interface as its native filesystem interface. What happens when you plug in your USB pendrive with a VFAT firesystem on it? VFAT doesn't give you an efficient way to get the hash of a single file, much less an entire subdirectory tree.

I think the key is to use the previously described opaque-version-identifier interface as long as possible; as long as no userland process requests a serialized hash for a filesystem node, you can postpone hashing the file. You can still provide access to snapshots of old versions of the filesystem by buffering the old versions of modified filesystem pages in RAM, as long as some snapshot refers to them. And, once a hash is requested, the OS can retain the hash in RAM as long as the firesystem is mounted.

There may be some filesystems — btrfs, perhaps, or ZFS — which provide a reliable way to determine whether a file has been modified since the last time the filesystem was mounted. They might have some kind of generation count or journal timestamp when the file was last modified. On such foreign systems, you could either use journal-timestamp/inode pairs (or whatever version identifier you end up using) in place of hashes, or you could store a lazily-populated mapping from foreign version identifiers to secure hashes.

Space exhaustion

A serious problem with any kind of lazy copy-on-write system — whether Linux's memory-page allocation, NetApp's WAFL block allocation, Flash FTLs, or a log-structured filesystem — is that space usage is somewhat unpredictable and difficult to attribute to a single cause. Consequently, it becomes exhausted unpredictably, and then it isn't entirely clear how to respond. With the foreign-filesystem interface described in the previous section, the problem is potentially exacerbated by having to buffer old versions of filesystem data in RAM, and RAM may be much smaller than the filesystem.

This is particularly a problem when it comes to database transaction durability. A database executing a two-phase commit must check for all possible errors, especially including errors of disk-space exhaustion, during the PREPARE phase; once the transaction enters the COMMIT phase, it is too late to rollback the transaction because of being out of space. For this particular case, we can use a call to preallocate some space that we may want to write to later on, producing a conservative (pessimistic) failure indication if there may not be space available. If the filesystem can then optimize the actual data by, for example, noting that the disk block written is the same as some other disk block, so much the better — such a situation could snatch victory from the jaws of defeat, but never defeat from the jaws of victory.

Topics

Turning a delay line into a counter with a FSM

Kragen Javier Sitaker, 2018-12-10 (1 minute)

Suppose you have an FSM connected to a delay line. There should be a pretty simple transition table to make it into a counter; it just needs to somehow set its carry bit when it wraps around to the beginning again. One solution is for the FSM's initial state to be such that it emits a "start mark" or "framing mark" which doesn't otherwise appear in the initial state of the delay line.

Probably the simplest such scheme is the Internet-0 encoding scheme: 11 is the framing mark, 01 represents a binary 1, and 10 represents a binary 0 (or vice versa), and 00 is unused ("blank tape" or no transmission). This particular scheme can even work with odd-length delay lines, since once the counter sees a correctly-framed 00, it can go into a mode where it searches for the framing mark even at odd bits.

This FSM, as a Mealy machine, has seven states: carrying, waiting to carry, copying, waiting to copy, searching, framing mark 1 (the initial state), and framing mark 2. Its transition and output table is as follows:

(XXX TODO)

	0	1
A: carrying		
B: waiting to carry		
C: copying		
D: waiting to copy		
E: searching		
F: framing mark 1		
G: framing mark 2		

Topics

Toward a language for hacking around with natural-language processing algorithms

Kragen Javier Sitaker, 2016-09-08 (7 minutes)

To come up with non-words beginning with "sex" consisting of an "s" followed by a real word, in bash:

```
< ~/devel/wordlist cat | while read freq word
    do case "$word" in
    ex*) echo "s$word";;
    esac
done | grep -vf <(< ~/devel/wordlist head -15000
                  | while read freq word
                    do echo "^$word\$" done) | head -45 | sort
```

This is probably the wrong way to do it. I thought it might be easier in Python, but it isn't:

```
import itertools
words = [line.split()[1] for line in open('/home/user/devel/wordlist')]
common_words = set(words[:15000])
swords = ('s' + word for word in words if word.startswith('ex'))

sorted(itertools.islice((sword for sword in swords if sword not in common_words),o
o 0, 45))
```

Python's set type, lazy generator expressions, and implicit file-line iteration are useful here, but this still ends up being kind of a lot of code, even more than the bash version, in part because genexes are pretty pointful, which is in part because Python's methods are not very useful to pass to higher-order expressions like `map` and `filter`.

Another thing to keep in mind here is that I invariably write this kind of thing incrementally, looking at the results computed by intermediate versions, in order to decide what to do next. For example, I added the filter to eliminate existing common words when "sexist" showed up in the output, and increased the cutoff from 2000 to 15000 when it continued to show up. Traditional `function(arguments)` syntax kind of sucks for this, because usually you write the arguments left to right (not least because the cursor's implicit motion is to the right as you type), and then you have to move back to the beginning to add the `function` bit. This gets even worse when we're wrapping something in a list comprehension or a generator expression.

The ideal environment for writing stuff like this incrementally would not be implicitly imperative, so that it could safely evaluate intermediate expressions without fear of damage and evaluate lazily for responsiveness without fear of confusion; it would allow you to add functions on the right; it would use map and filter functions rather than list comprehensions or generator expressions; it would use CLOS-like generic functions, with ML-like implicit currying, instead of methods so that you could use them as map and filter

arguments; and it would allow you to write the equivalent of `f(a, g(b, h(c, d)))` without matching parens.

These requirements actually make it sound a lot like Forth! But I don't think we need to descend down the rabbit hole of typelessness and syntaxlessness to get these advantages.

Here's a hack at an alternative syntax that maximizes left-to-right typability with incremental feedback:

```
'/home/user/devel/wordlist' file, split each; 1 th each -> words
words, 'ex' startswith only;
    ('s' ++) each;
    , (words, :15000 th set) contains not each;
    :45 th sorted
```

I'm not sure that's right or even parseable, but here are the things I was trying:

- `f(a, b, c)` is written as `a, b, c f`
- `x = a` is written as `a -> x`.
- `f(a, g(b, h(c, d)))` is written, with syntax somewhat borrowed from Mark Lentczner's Glyphic Script, which in turn borrowed it with a semantic change from Smalltalk, as `c, d h; b g; a f`. I'm not sure that I have the parsing rules down correctly, and it might be better to use `c, d h | b g | a f`. Also I think both of these do the wrong thing as you type the `c, d h; b` part, because it tries to apply `b` to the results of `h`, though `b` will become merely an argument to `g`; this will result in misleading feedback in the middle. (The `;,` thing in the middle there makes me think that this is really wrong and I need to rethink it.)
- For cases where the `,,;,,;` nesting-free syntax is inadequate (because you need nesting in some parameter that isn't the first one)
- `map` is called `each`. Probably it would be better to default to `flatMap`, but this code is written with the opposite assumption.
- Sequence indexing `[]` is spelled `th` (as in `4 th`, `5 th`, etc.), and sequences are lazy by default. This may not be the best spelling.
- `:` is an infix operator with the same semantics as in Python slicing, except for indexes from the end: it returns a slice object that can be used with `th`.
- Functions are implicitly curried, making it easy to write functions like `1 th each`, which means λx.each(λy.th 1 y)x in the λ-calculus.
- There are no functions as separate from methods. If we need ad-hoc polymorphism, as for `th` and sequence iteration, we use CLOS-style generic functions.
- `++` is the sequence concatenation function as an infix operator, in order to avoid semantic confusion with the `+` operator for numeric addition, which probably ought to be implicitly lifted to apply over conformable functions and sequences. I thought about using OCaml `^`, SQL `||`, Perl/PHP `.`, or Lua `..`, but all of those already carry far too much semantic baggage already.
- `(x op)`, where `op` is an infix operator, is a Haskell-style "section", meaning λy.x op y.
- `not` is implicitly lifted to operate over not just booleans but boolean-returning functions.
- `open` is called `file` and implicitly opens read-only, as in Python
- Probably opening a file for write should use some kind of IO monad

or whatever.
• `in` is called `contains`, on the theory that `common_words contains` is a sensible phrase with the right meaning.
• `split` takes a string and optionally a separator. I need to figure out how to reconcile implicit currying with optional arguments, but I suspect OCaml labeled arguments have my back, and I don't even have to use their shitty syntax because I'm not backwards-compatible with ML.
• `filter` is called `only`, so that `words, 'ex' startswith only` returns the items from `words` that start with 'ex'.
• `sorted` sorts the provided sequence, which is necessarily eager (the last input item might be the first output) but because of lazy evaluation, this only needs to loop until the first 45 items are generated.

The above suggests that it actually is necessary to have some kind of separator between a function and its arguments; in this line

```
'/home/user/devel/wordlist' file, split each; 1 th each
```

it's totally ridiculous that `each` is being applied not only to `split` but to the return of `file`. We could use `.` to apply functions and call methods:

```
'/home/user/devel/wordlist'.file, split. each, 1.th. each -> words
words, ('ex'.startswith). only, ('s' ++). each,
    (words, :15000.th. set. contains. not). each.
    :45.th.sorted
```

Or alternatively we could use `;`, but that's kind of terrible, really, because the whole point of `;` is that its left and right precedence are very different:

```
'/home/user/devel/wordlist'; file, split; each, 1;th; each -> words
words, ('ex'; startswith); only, ('s' ++); each;
    (words, :15000;th; set; contains; not); each;
    :45;th;sorted
```

Topics

My very first toddling steps in ARM assembly language

Kragen Javier Sitaker, 2019-12-10 (updated 2019-12-13) (46 minutes)

It's long past time I learned to write a little bit of ARM assembly and machine code! So I spent two hours and was able to get hello-world running, and then a few more hours and learned a bunch of other things.

Basic tools

As it happens, although this laptop is an amd64, it has cross-compilation and transparent CPU emulation stuff installed, so this works:

```
$ cat hello.c
#include <stdio.h>

int main() { printf("hello, world\n"); return 0; }
$ arm-linux-gnueabihf-gcc-5 -static hello.c -o hello.arm
$ ./hello.arm
hello, world
$
```

Specifically I have these Linux Mint packages installed:

- gcc-5-arm-linux-gnueabihf
- gcc-5-arm-linux-gnueabihf-base (its prerequisite)
- binutils-arm-linux-gnueabihf
- libc6-armhf-cross
- qemu-user
- qemu-user-binfmt

(Warning: much of the following is quoted from glibc, a copyrighted work licensed under the GNU Lesser General Public License.)

Although I do have libc6-armhf-cross installed, running dynamic executables does not work. This makes disassembly kind of a pain:

```
$ arm-linux-gnueabihf-objdump -d !$
arm-linux-gnueabihf-objdump -d ./a.out

./a.out:     file format elf32-littlearm

Disassembly of section .init:

00010160 <_init>:
   10160:   e92d4008    push    {r3, lr}
   10164:   eb000092    bl  103b4 <call_weak_fn>
   10168:   e8bd8008    pop {r3, pc}

Disassembly of section .iplt:

0001016c <.iplt>:
```

```
   1016c:   4778        bx  pc
   1016e:   46c0        nop         ; (mov r8, r8)
   10170:   e28fc600    add ip, pc, #0, 12
   10174:   e28cca68    add ip, ip, #104, 20    ; 0x68000
   10178:   e5bcfe94    ldr pc, [ip, #3732]!    ; 0xe94

Disassembly of section .text:

00010180 <backtrace_and_maps>:
   10180:   2801        cmp r0, #1
   10182:   f340 8084   ble.w   1028e <backtrace_and_maps+0x10e>
   10186:   2900        cmp r1, #0
(97,167 more lines follow)
$
```

My first steps

We can see that this is Thumb-2 machine code, with some instructions 16-bit and others 32-bit.

Buried in there is the way system calls work on ARM Linux:

```
00010aa0 <__libc_do_syscall>:
   10aa0:   b580        push    {r7, lr}
   10aa2:   4667        mov r7, ip
   10aa4:   df00        svc 0
   10aa6:   bd80        pop {r7, pc}
```

At a guess, r7 selects the system call.

And exit(2):

```
00020648 <_exit>:
   20648:   b500        push    {lr}
   2064a:   4603        mov r3, r0
   2064c:   f04f 0cf8   mov.w   ip, #248    ; 0xf8
   20650:   f7f0 fa26   bl  10aa0 <__libc_do_syscall>
   20654:   f510 5f80   cmn.w   r0, #4096   ; 0x1000
   20658:   d810        bhi.n   2067c <_exit+0x34>
   2065a:   4618        mov r0, r3
   2065c:   f04f 0c01   mov.w   ip, #1
   20660:   f7f0 fa1e   bl  10aa0 <__libc_do_syscall>
   20664:   f510 5f80   cmn.w   r0, #4096   ; 0x1000
   20668:   d800        bhi.n   2066c <_exit+0x24>
   2066a:   deff        udf #255    ; 0xff
   2066c:   4b07        ldr r3, [pc, #28]   ; (2068c <_exit+0x44>)
   2066e:   ee1d 2f70   mrc 15, 0, r2, cr13, cr0, {3}
   20672:   4240        negs    r0, r0
   20674:   447b        add r3, pc
   20676:   681b        ldr r3, [r3, #0]
   20678:   50d0        str r0, [r2, r3]
   2067a:   deff        udf #255    ; 0xff
   2067c:   4a04        ldr r2, [pc, #16]   ; (20690 <_exit+0x48>)
   2067e:   ee1d 1f70   mrc 15, 0, r1, cr13, cr0, {3}
   20682:   4240        negs    r0, r0
   20684:   447a        add r2, pc
   20686:   6812        ldr r2, [r2, #0]
```

```
   20688:   5088        str r0, [r1, r2]
   2068a:   e7e6        b.n 2065a <_exit+0x12>
   2068c:   000589cc    .word   0x000589cc
   20690:   000589bc    .word   0x000589bc
```

I'm guessing that this is the actual exiting part:

```
   2065c:   f04f 0c01   mov.w   ip, #1
   20660:   f7f0 fa1e   bl  10aa0 <__libc_do_syscall>
```

So I tried putting this in a file and compiling it:

```
        ; Attempt to write an ARM assembly program that exits
        ; successfully.
main:
        mov.w r7, #1
        svc 0
loop:   b.n loop
```

But it seems like that is completely the wrong syntax. I asked GCC for a listing please:

```
$ arm-linux-gnueabihf-gcc-5 -static -Wa,-adhlns=hello.lst hello.c
```

It obliged:

```
  1                     .arch armv7-a
  2                     .eabi_attribute 28, 1
  3                     .fpu vfpv3-d16
  4                     .eabi_attribute 20, 1
  5                     .eabi_attribute 21, 1
  6                     .eabi_attribute 23, 3
  7                     .eabi_attribute 24, 1
  8                     .eabi_attribute 25, 1
  9                     .eabi_attribute 26, 2
 10                     .eabi_attribute 30, 6
 11                     .eabi_attribute 34, 1
 12                     .eabi_attribute 18, 4
 13                     .file   "hello.c"
 14                     .section    .rodata
 15                     .align  2
 16                 .LC0:
 17 0000 68656C6C       .ascii  "hello, world\000"
 17      6F2C2077
 17      6F726C64
 17      00
 18                     .text
 19                     .align  2
 20                     .global main
 21                     .syntax unified
 22                     .thumb
 23                     .thumb_func
 25                 main:
 26                     @ args = 0, pretend = 0, frame = 0
 27                     @ frame_needed = 1, uses_anonymous_args = 0
```

```
  28 0000 80B5          push    {r7, lr}
  29 0002 00AF          add r7, sp, #0
  30 0004 40F20000      movw    r0, #:lower16:.LC0
  31 0008 C0F20000      movt    r0, #:upper16:.LC0
  32 000c FFF7FEFF      bl  puts
  33 0010 0023          movs    r3, #0
  34 0012 1846          mov r0, r3
  35 0014 80BD          pop {r7, pc}

  37                    .ident  "GCC: (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0
 20160609"
  38 0016 00BF          .section    .note.GNU-stack,"",%progbits
DEFINED SYMBOLS
                           *ABS*:0000000000000000 hello.c
     /tmp/cc20XqWG.s:15    .rodata:0000000000000000 $d
     /tmp/cc20XqWG.s:16    .rodata:0000000000000000 .LC0
     /tmp/cc20XqWG.s:25    .text:0000000000000000 main
     /tmp/cc20XqWG.s:28    .text:0000000000000000 $t

UNDEFINED SYMBOLS
puts
```

Evidently r0 contains the argument for `puts` and also the return value for `main`.

Aping the syntax therein, I tried this:

```
        @ Attempt to write an ARM assembly program that exits
        @ successfully.
        .arch armv7-a
        .syntax unified
        .thumb
        .globl main
main:
        mov.w r7, #1
        svc 0
loop:   b.n loop
```

That does build successfully; `arm-linux-gnueabihf-objdump` on the resulting executable suggests that the requested instructions were emitted:

```
(322 lines omitted)
0001049c <main>:
   1049c:   f04f 0701   mov.w   r7, #1
   104a0:   df00        svc 0

000104a2 <loop>:
   104a2:   e7fe        b.n 104a2 <loop>
(87108 lines omitted)
```

However, upon execution, the program segfaults. So I guessed wrong about *something* but without a debugger or knowing how to print things it's hard to tell what still.

Let's try building a program that exits successfully with GCC:

```
$ cat return42.c
int main(int argc, char **argv)
{
  return 42;
}
$ make return42
cc -Wall -Werror -std=gnu99    return42.c   -o return42
$ ./return42
$ echo $?
42
$ arm-linux-gnueabihf-gcc-5 -static -Wa,-adhlns=return42.lst return42.c
$ cat return42.lst
...
  21                main:
  22                    @ args = 0, pretend = 0, frame = 8
  23                    @ frame_needed = 1, uses_anonymous_args = 0
  24                    @ link register save eliminated.
  25 0000 80B4          push    {r7}
  26 0002 83B0          sub sp, sp, #12
  27 0004 00AF          add r7, sp, #0
  28 0006 7860          str r0, [r7, #4]
  29 0008 3960          str r1, [r7]
  30 000a 2A23          movs    r3, #42
  31 000c 1846          mov r0, r3
  32 000e 0C37          adds    r7, r7, #12
  33 0010 BD46          mov sp, r7
  34                    @ sp needed
  35 0012 5DF8047B      ldr r7, [sp], #4
  36 0016 7047          bx  lr
...
```

Maybe bx is "branch indirect". Also maybe I should use some optimization:

```
  21                main:
  25 0000 2A20          movs    r0, #42
  26 0002 7047          bx  lr
```

That's more like it. Can I get that to build?

```
$ cat return42-arm.s
        @ Attempt to write an ARM assembly program that exits
        @ successfully.
        .arch armv7-a
        .syntax unified
        .thumb
        .globl main
main:
        mov.w r0, #42
        bx lr
loop:   b.n loop
$ arm-linux-gnueabihf-gcc-5 -static return42-arm.s
$ file a.out

a.out: ELF 32-bit LSB executable, ARM, EABI5 version 1 (GNU/Linux), statically li
```

```
onked, for GNU/Linux 3.2.0, BuildID[sha1]=6ddb42d20b6cff668f5c6ded33b82eeda0e3bec3o
o, not stripped
$ ./a.out
qemu: uncaught target signal 4 (Illegal instruction) - core dumped
Illegal instruction
$
```

Hmm, that's not what I was hoping for. Maybe some of the other assembly directives are needed to build a runnable executable?

```
        @ An ARM assembly program that exits successfully.
        .arch armv7-a
        .syntax unified
        .thumb
        .thumb_func
        .globl main
main:
        mov.w r0, #42
        bx lr
```

It turned out to be .thumb_func. The Gas manual explains, "This directive specifies that the following symbol is the name of a Thumb encoded function. This information is necessary in order to allow the assembler and linker to generate correct code for interworking [sic] between Arm and Thumb instructions and should be used even if interworking is not going to be performed. The presence of this directive also implies '.thumb'. This directive is not necessary when generating EABI objects. On these targets the encoding is implicit when generating Thumb code."

(The manual is apparently wrong about it not being necessary when generating EABI objects.)

A little more perusing of the manual allows me to reduce this to the following:

```
        @ An ARM assembly program that exits successfully.
        .arch armv7-a
        .syntax unified
        .thumb_func
        .globl main
main:   mov.w r0, $42
        bx lr
```

Adding this line before the return instruction converts the program into an infinite loop, as expected:

```
loop:   b.n loop
```

This program runs, prints "hello, world" as hoped for, and exits:

```
        .globl main
main:   push {lr}
        movw r0, #:lower16:hi
        movt r0, #:upper16:hi
        bl puts
        mov r0, $0
```

```
        pop {pc}
hi:     .ascii "hello, world\0"
```

$ doesn't work for the half-symbols. Note no .thumb_func, and consequently it generates non-Thumb code!

```
0001049c <main>:
   1049c:   e52de004    push    {lr}        ; (str lr, [sp, #-4]!)
   104a0:   e30004b4    movw    r0, #1204   ; 0x4b4
   104a4:   e3400001    movt    r0, #1
   104a8:   fa00120e    blx 14ce8 <_IO_puts>
   104ac:   e3a00000    mov r0, #0
   104b0:   e49df004    pop {pc}        ; (ldr pc, [sp], #4)
```

Sticking .thumb_func back in there corrects this:

```
0001049c <main>:
   1049c:   b500        push    {lr}
   1049e:   f240 40ae   movw    r0, #1198   ; 0x4ae
   104a2:   f2c0 0001   movt    r0, #1
   104a6:   f004 fc1f   bl  14ce8 <_IO_puts>
   104aa:   2000        movs    r0, #0
   104ac:   bd00        pop {pc}
```

Okay, let's try something with real computation:

```
#include <stdlib.h>

int main(int argc, char **argv)
{
  if (atoi(argv[1]) == 37) printf("whoa\n");
  return 0;
}
```

This compiles to more or less the following:

```
  21                main:
  24 0000 08B5          push    {r3, lr}
  25 0002 4868          ldr r0, [r1, #4]
  26 0004 0A22          movs    r2, #10
  27 0006 0021          movs    r1, #0
  28 0008 FFF7FEFF      bl  strtol
  29 000c 2528          cmp r0, #37
  30 000e 05D1          bne .L2
  31 0010 40F20000      movw    r0, #:lower16:.LC0
  32 0014 C0F20000      movt    r0, #:upper16:.LC0
  33 0018 FFF7FEFF      bl  puts
  34                .L2:
  35 001c 0020          movs    r0, #0
  36 001e 08BD          pop {r3, pc}
  38                    .section    .rodata.str1.4,"aMS",%progbits,1
  39                    .align  2
  40                .LC0:
  41 0000 77686F61      .ascii  "whoa\000"
  41      00
```

So it looks like it's calling strtol(argv[1], 0, 10), passing the args in r0, r1, and r2, and getting the result in r0. Why it's saving r3 I have no idea. I'm guessing the `ldr r0, [r1, #4]` syntax is for indexing 4 bytes off r1 and loading the result into register r0. The rest is the same.

Does it use this same register-passing convention for varargs functions? Let's see:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int n = atoi(argv[1]);
  if (n != 37) printf("whoa %d\n", n);
  return 0;
}
```

```
  21                main:
  22                    @ args = 0, pretend = 0, frame = 0
  23                    @ frame_needed = 0, uses_anonymous_args = 0
  24 0000 08B5          push    {r3, lr}
  25 0002 4868          ldr r0, [r1, #4]
  26 0004 0A22          movs    r2, #10
  27 0006 0021          movs    r1, #0
  28 0008 FFF7FEFF      bl  strtol
  29 000c 2528          cmp r0, #37
  30 000e 07D0          beq .L2
  31 0010 0246          mov r2, r0
  32 0012 40F20001      movw    r1, #:lower16:.LC0
  33 0016 C0F20001      movt    r1, #:upper16:.LC0
  34 001a 0120          movs    r0, #1
  35 001c FFF7FEFF      bl  __printf_chk
  36                .L2:
  37 0020 0020          movs    r0, #0
  38 0022 08BD          pop {r3, pc}
  40                    .section    .rodata.str1.4,"aMS",%progbits,1
  41                    .align  2
  42                .LC0:
  43 0000 77686F61      .ascii  "whoa %d\012\000"
  43      2025640A
  43      00
```

This looks pretty similar but it seems to be passing the format argument in r1, the int argument in r2, and the number of arguments in r0, to a function called `__printf_chk`. I can ape this pretty well:

```
$ cat you-arm.s
        @ Simple ARM assembly program to say "hello, Fred" when run with "Fred"
        .globl main
        .thumb_func
main:   push {r3, lr}
        ldr r2, [r1, #4]    @ argv[1]
        mov r0, $1
        movw r1, #:lower16:hi
```

```
        movt r1, #:upper16:hi
        bl __printf_chk
        mov r0, $0
        pop {r3, pc}
hi:     .ascii "hello, %s\n\0"
$ arm-linux-gnueabihf-gcc-5 -static you-arm.s
$ ./a.out Fred
hello, Fred
```

So based on the above, I can do the dumb Fibonacci benchmark program:

```
        @ Simple ARM assembly program to compute dumb Fibonacci
        .thumb_func
fib:    push {r3, lr}
        cmp r0, $0
        beq basecase
        cmp r0, $1
        beq basecase
        push {r0}
        sub r0, r0, $1
        bl fib
        mov r1, r0
        pop {r0}
        push {r1}
        sub r0, r0, $2
        bl fib
        pop {r1}
        add r0, r1, r0
        pop {r3, pc}
basecase:
        mov r0, $1
        pop {r3, pc}

        .globl main
        .thumb_func
main:   push {r3, lr}
        ldr r0, [r1, #4]    @ argv[1]
        mov r1, $0
        mov r2, $10
        bl strtol
        bl fib
        mov r2, r0
        mov r0, $1
        movw r1, #:lower16:hi
        movt r1, #:upper16:hi
        bl __printf_chk
        mov r0, $0
        pop {r3, pc}
hi:     .ascii "fib = %d\n\0"
```

This comes out as the following:

```
0001049c <fib>:
   1049c:   b508        push    {r3, lr}
```

```
   1049e:   2800        cmp r0, #0
   104a0:   d00e        beq.n   104c0 <basecase>
   104a2:   2801        cmp r0, #1
   104a4:   d00c        beq.n   104c0 <basecase>
   104a6:   b401        push    {r0}
   104a8:   3801        subs    r0, #1
   104aa:   f7ff fff7   bl  1049c <fib>
   104ae:   1c01        adds    r1, r0, #0
   104b0:   bc01        pop {r0}
   104b2:   b402        push    {r1}
   104b4:   3802        subs    r0, #2
   104b6:   f7ff fff1   bl  1049c <fib>
   104ba:   bc02        pop {r1}
   104bc:   1808        adds    r0, r1, r0
   104be:   bd08        pop {r3, pc}

000104c0 <basecase>:
   104c0:   2001        movs    r0, #1
   104c2:   bd08        pop {r3, pc}

000104c4 <main>:
   104c4:   b508        push    {r3, lr}
   104c6:   6848        ldr r0, [r1, #4]
   104c8:   2100        movs    r1, #0
   104ca:   220a        movs    r2, #10
   104cc:   f003 ff9e   bl  1440c <__strtol>
   104d0:   f7ff ffe4   bl  1049c <fib>
   104d4:   1c02        adds    r2, r0, #0
   104d6:   2001        movs    r0, #1
   104d8:   f240 41e8   movw    r1, #1256   ; 0x4e8
   104dc:   f2c0 0101   movt    r1, #1
   104e0:   f012 fa88   bl  229f4 <___printf_chk>
   104e4:   2000        movs    r0, #0
   104e6:   bd08        pop {r3, pc}
```

So I guess I can say that's the first program I've written in ARM assembly, since the others were mostly just slight modifications of GCC output. I'm still cargo-culting the saving of r3, and I probably should use a less-than comparison rather than two equal-to comparisons, and I don't know what order registers get pushed.

It segfaults if you feed it -1, and I think maybe this system is configured with apport to send the core dumps to Ubuntu or something.

A minimal -nostdlib program in ARM assembly

And here's a program that successfully invokes `_exit` via the SVC instruction instead of using the standard library, and thus can be linked with -nostdlib and doesn't make a humongous executable:

```
@ Attempt to write an ARM assembly program that exits
@ successfully with -nostdlib.  cf. return42.c.
.syntax unified
.thumb_func
```

```
        .globl _start
_start: mov r7, #1   @ system call 1: _exit
        mov r0, #42  @ exit return value?
        svc 0
loop:   b.n loop
```

This produces a reasonable disassembly:

```
$ arm-linux-gnueabihf-gcc-5 -static -nostdlib goodbyearm.s
$ ./a.out
$ echo $?
42
$ arm-linux-gnueabihf-objdump -d a.out

a.out:     file format elf32-littlearm

Disassembly of section .text:

00010098 <_start>:
   10098:   f04f 0701   mov.w   r7, #1
   1009c:   f04f 002a   mov.w   r0, #42 ; 0x2a
   100a0:   df00        svc 0

000100a2 <loop>:
   100a2:   e7fe        b.n 100a2 <loop>
$
```

So, that took a couple of hours to figure out, but it did eventually work.

Machine instructions seen thus far

Destination register always comes first.

- svc: supervisor call; in Linux we use svc 0 with the system call number in r7.
- b.n: branch always
- beq, bne: branch if equal or not equal
- bx: "branch and exchange" (not necessarily indirect; see below)
- bl: branch and link (i.e., call)
- push, pop: take sets of registers; can push lr and pop pc. Not sure how order is determined yet.
- mov: can load an immediate constant into a register or copy register to register
- movt: sets upper 16 bits of register to immediate constant
- movw: sets register to 16-bit immediate constant (or maybe sets lower 16 bits?)
- ldr, str: load or store registers to memory, supporting index-offset and I think decrement addressing modes
- cmp: can compare registers to immediate constants
- sub/subs, add/adds: can add or subtract registers, immediate constants, or both
- nop: nop.

Still mysterious: cmn.w, bhi.n, udf, mrc, negs, the whole .w and .n and "s" suffix thing, and what is this "ip" register?

Hmm, the Gas manual actually explains the "s" suffix: that means to set the flags. So presumably "add" just does an addition, while "adds" does an addition and also sets carry flags and whatnot.

write(2), and a -nostdlib hello, world

If we want to get output without stdlib, we need to be able to invoke the SVC for write(2); looks like maybe that's the system call with r7=4:

```
00021180 <__libc_write>:
   21180:   f8df c04a   ldr.w   ip, [pc, #74]   ; 211ce <__libc_write+0x4e>
   21184:   44fc        add ip, pc
   21186:   f8dc c000   ldr.w   ip, [ip]
   2118a:   f09c 0f00   teq ip, #0
   2118e:   b480        push    {r7}
   21190:   d108        bne.n   211a4 <__libc_write+0x24>
   21192:   2704        movs    r7, #4
   21194:   df00        svc 0
   21196:   bc80        pop {r7}
   21198:   f510 5f80   cmn.w   r0, #4096   ; 0x1000
   2119c:   bf38        it  cc
   2119e:   4770        bxcc    lr
   211a0:   f002 bb4e   b.w 23840 <__syscall_error>
   211a4:   b50f        push    {r0, r1, r2, r3, lr}
   211a6:   f001 f9d9   bl  2255c <__libc_enable_asynccancel>
   211aa:   4684        mov ip, r0
   211ac:   bc0f        pop {r0, r1, r2, r3}
   211ae:   2704        movs    r7, #4
   211b0:   df00        svc 0
   211b2:   4607        mov r7, r0
   211b4:   4660        mov r0, ip
   211b6:   f001 fa15   bl  225e4 <__libc_disable_asynccancel>
   211ba:   4638        mov r0, r7
   211bc:   f85d eb04   ldr.w   lr, [sp], #4
   211c0:   bc80        pop {r7}
   211c2:   f510 5f80   cmn.w   r0, #4096   ; 0x1000
   211c6:   bf38        it  cc
   211c8:   4770        bxcc    lr
   211ca:   f002 bb39   b.w 23840 <__syscall_error>
   211ce:   9d40        .short  0x9d40
   211d0:   bf000005    .word   0xbf000005
```

I don't know what all the extra stuff is in there for but presumably it covers up some impedance mismatch between the Linux system call and what the standard library behavior is supposed to be.

On this basis I achieved a stdlibless hello, world:

```
$ cat hellobarearm.s
        @ Attempt to write an ARM assembly program that hellos
        @ successfully with -nostdlib.  cf. goodbyearm.s
        .syntax unified
        .thumb_func
        .globl _start
_start: mov r7, #4   @ system call 4: write
        mov r0, #0
```

```
        movw r1, #:lower16:hello
        movt r1, #:upper16:hello
        mov r2, #(helloend - hello)
        svc 0

        mov r7, #1   @ system call 1: _exit
        mov r0, #0   @ exit return value
        svc 0
hello:  .ascii "hello, world\n"
helloend:
$ arm-linux-gnueabihf-gcc-5 -static -nostdlib hellobarearm.s
$ ./a.out
hello, world
$ ls -l a.out
-rwxr-xr-x 1 user user 964 Dec 11 02:56 a.out
$ arm-linux-gnueabihf-objdump -d a.out

a.out:     file format elf32-littlearm

Disassembly of section .text:

00010098 <_start>:
   10098:   f04f 0704   mov.w   r7, #4
   1009c:   f04f 0000   mov.w   r0, #0
   100a0:   f240 01b8   movw    r1, #184    ; 0xb8
   100a4:   f2c0 0101   movt    r1, #1
   100a8:   f04f 020d   mov.w   r2, #13
   100ac:   df00        svc 0
   100ae:   f04f 0701   mov.w   r7, #1
   100b2:   f04f 0000   mov.w   r0, #0
   100b6:   df00        svc 0

000100b8 <hello>:
   100b8:   6c6c6568    .word   0x6c6c6568
   100bc:   77202c6f    .word   0x77202c6f
   100c0:   646c726f    .word   0x646c726f
   100c4:   0a              .byte   0x0a

000100c5 <helloend>:
        ...
$
```

Note that `strip` or rather `arm-linux-gnueabihf-strip` seems to break objdump's ability to disassemble the code, but it still runs. Here's a dump of the stripped executable:

```
$ od -vbAn a.out
 177 105 114 106 001 001 001 000 000 000 000 000 000 000 000 000
 002 000 050 000 001 000 000 000 231 000 001 000 064 000 000 000
 034 001 000 000 000 002 000 005 064 000 040 000 002 000 050 000
 005 000 004 000 001 000 000 000 000 000 000 000 000 000 001 000
 000 000 001 000 306 000 000 000 306 000 000 000 005 000 000 000
 000 000 001 000 004 000 000 000 164 000 000 000 164 000 001 000
 164 000 001 000 044 000 000 000 044 000 000 000 004 000 000 000
 004 000 000 000 004 000 000 000 024 000 000 000 003 000 000 000
```

```
107 116 125 000 005 140 061 140 131 337 041 016 031 220 022 215
156 115 132 247 261 367 300 226 117 360 004 007 117 360 000 000
100 362 270 001 300 362 001 001 117 360 015 002 000 337 117 360
001 007 117 360 000 000 000 337 150 145 154 154 157 054 040 167
157 162 154 144 012 000 101 036 000 000 000 141 145 141 142 151
000 001 024 000 000 000 005 067 055 101 000 006 012 007 101 010
001 011 002 012 004 000 056 163 150 163 164 162 164 141 142 000
056 156 157 164 145 056 147 156 165 056 142 165 151 154 144 055
151 144 000 056 164 145 170 164 000 056 101 122 115 056 141 164
164 162 151 142 165 164 145 163 000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000 000 000 000 013 000 000 000 007 000 000 000 002 000 000 000
164 000 001 000 164 000 000 000 044 000 000 000 000 000 000 000
000 000 000 000 004 000 000 000 000 000 000 000 036 000 000 000
001 000 000 000 006 000 000 000 230 000 001 000 230 000 000 000
056 000 000 000 000 000 000 000 000 000 000 000 004 000 000 000
000 000 000 000 044 000 000 000 003 000 000 160 000 000 000 000
000 000 000 000 306 000 000 000 037 000 000 000 000 000 000 000
000 000 000 000 001 000 000 000 000 000 000 000 001 000 000 000
003 000 000 000 000 000 000 000 000 000 000 000 345 000 000 000
064 000 000 000 000 000 000 000 000 000 000 000 001 000 000 000
000 000 000 000
```

Although, doh, I seem to be writing to fd 0, not fd 1. QEMU confirms:

```
$ qemu-arm -strace ./a.out
30784 write(0,0x100b8,13)hello, world
 = 13
30784 exit(0)
```

read(2) seems to be syscall 3. open(2) syscall 5, and close(2) syscall 6, and maybe brk() 45 and getpid() 20.

The ARM-THUMB Procedure Calling Standard

ARM publishes this document, called the ATPCS for short. It explains the use of the registers: r0 and r1 are used for return values; r0 to r3 are used for arguments and are thus caller-saved; r4 to r11 are callee-saved general-purpose registers; r11 is also FP, the frame pointer; r12 is IP, the "intra-procedure-call scratch register"; r13 is SP; r14 is LR, the link register used by the `bl` instruction; r15 is PC. The stack grows downwards, and the stack pointer (which must be 8-byte-aligned when calling a public function) points at the last thing that was pushed, not the next thing to push. Interrupt handlers can execute on your stack, so if you have interrupts you can't depend on values you've popped staying put.

Newer versions are longer and of poorer quality, though covering more modern CPU features; fortunately I was able to find an older version, "SWS ESPC 0002 B-01" (B-01 being the version number), from "24 October, 2000", which is only 37 pages.

Original-THUMB instructions can only access r0 to r7 for most operands, so you only have 8 general-purpose registers, like the 8088;

more recent ones can access the other 8 registers but I think need longer instructions.

There are also some special uses of r10 ("SL", "stack limit" in stack-checked variants), r9 ("SB", "static base" for shared libraries and other uses of position-independent data), and r7 ("WR", "Thumb-state Work Register"), but I don't think these affect their use most of the time --- from the perspective of writing assembly code, they're mostly just more callee-saved registers, except that I guess if your assembly code is in a shared library it will use r9. More recent versions of the ATPCS eliminate SL and WR and give r9 an additional role, TR, the "thread register", for TLS I guess.

I think the definition of "IP" means that the dynamic linker is free to clobber r12 when it's doing lazy dynamic linking, so the callee may see random crap in r12 if it just got loaded. This also means it's caller-saved.

So, in summary, r4-r11 and SP (r13) are callee-saved, and everything else is caller-saved. (Except that r10 may get horked by "limit-checking support code".)

Parameter passing is what you'd expect: everything gets widened to 32 bits, except 64-bit values are split into two 32-bit values (not sure about endianness). The first four parameters go into r0-r3, and subsequent parameters are passed on the stack, first argument last. (So the argument-count passing used by `__printf_chk` above is nonstandard, which I guess is why it was calling `__printf_chk` and not `printf`.) Return values go in r0, r0-r1, r0-r2, or r0-r3, if they fit, while longer return values are returned "indirectly, in memory, via an additional address parameter." Not sure whether that parameter is passed in by the caller as a ghostly first parameter or what.

Floating-point parameter passing is different but uses floating-point registers.

The floating-point story sounds remarkably like the 8086-family story. The old FPA register set has 8 extended-precision registers (I think 12 bytes instead of the 8087's 10) that can be used as single-precision, which has recently been replaced by "VFP", "vector floating point", which has 16 double-precision registers that can be used as 32 single-precision registers instead. A difference is that they are mutually exclusive: while an Atom supports both 80387 instructions and SSE instructions, ARM chips support either FPA or VFP or neither, not both.

The assembly-language examples use a syntax closely resembling Intel syntax, with ; for comments and no %, but . to mean the current position:

```
MOV   LR, PC     ; VAL(C) = . + 8
MOV   PC, r4
```

GCC and Gas use something like this syntax by default except for the comments. The ATPCS sometimes says things like "-4[FP]" in the body text; it's not clear to me whether this is valid assembly syntax in ARM's mind, but Gas seems to be writing that as `[fp, #-4]`.

"BX" is "branch and exchange", not "branch indirect" as I thought; it uses the LSB of the address to determine whether to use ARM or THUMB instructions after the jump. There's a note: "In ARM architecture version 5T, a load (but not a move) to the PC also

restores the instruction-set state, allowing an inter-working return to be performed using LDR, LDM, or POP," which I guess means that before arm5t that wasn't the case.

Shared libraries and position-independent code

To get position-independent code, you have to use PC-relative references to all your read-only data, and you have to access read-write data by indexing off SB. This in particular means that no static data can point to any other static data without dynamic-linker intervention, even inside the same segment; and read-only data, to be sharable, can only point to read-write data by indexing off SB. I don't know how non-instruction read-only data can point to read-only data at all.

There is a delightful hack for shared libraries to find their read-write data segments in their entry trampolines: every shared library has a "library index" set at load time, and every shared library read-write data segment starts with has four pointers into a "process data table" which lists the data segments of the shared libraries. So you're supposed to do this, in THUMB code:

```
MOV LSB, SB             ; for some "low register" LSB
LDR LSB, [LSB, #my_segment]  ; 0, 1, 2, or 3
LDR LSB, [LSB, #my_index]    ; set by the dynamic linker
```

Surprisingly, section 5.8 is about Chez-Scheme-style segmented stacks (called "chunked stacks"). Too bad there's no explicit support for closures, although maybe the GCC-style trampoline hack is better than an explicit context pointer in the ABI, which would slow down every call. (Although the SB thing is pretty close to being exactly an explicit context pointer in the ABI...)

Stack unwinding

A surprising thing about the ATPCS is that it contains an unwinding spec to make it possible for zero-overhead exception handlers to safely unwind the stack, even through shared libraries, restoring all callee-saved registers just as if the functions unwound had returned normally. Even more surprising is that the recommended approach is to examine the binary code of the functions on the stack to identify their prologues or epilogues, and then either interpretively undo the effect of the prologue, or directly execute the epilogue. This is very clever, but for it to work, you need to not move the stack pointer during the body of the function the way my dumb Fibonacci code above does. If that requirement is satisfied, though, I think only a very minimal amount of auxiliary data is required to make the unwinding work.

The Gas manual leads me to believe that this is not the method currently used for unwinding, because it demands that you tell it what registers you're saving with a `.save` directive.

A bit more disassembly, exploring instruction set differences, and failing to figure out shared libraries

I wrote this C program on a different computer and compiled it

with `arm-none-eabi-gcc -S -mthumb -O`, with no `#include` files:

```
int main(int argc, char **argv) {
    printf("%d\n", atoi(argv[1]) << 3 | atoi(argv[2]));
    return 0;
}
```

The assembly code generated inside `main` looks mostly like this:

```
main:
        push    {r3, r4, r5, lr}
        mov     r4, r1
        ldr     r0, [r1, #4]
        bl      atoi
        mov     r5, r0
        ldr     r0, [r4, #8]
        bl      atoi
        lsl     r1, r5, #3
        orr     r1, r0
        ldr     r0, .L2
        bl      printf
        mov     r0, #0
        @ sp needed for prologue
        pop     {r3, r4, r5}
        pop     {r1}
        bx      r1
.L3:
        .align  2
.L2:
        .word   .LC0
        .size   main, .-main
        .section        .rodata.str1.4,"aMS",%progbits,1
        .align  2
.LC0:
        .ascii  "%d\012\000"
```

The `ldr` for the `atoi` arguments confirms that the #4 or #8 is a byte offset. The `lsl` and `orr` mnemonics were what I was really looking for, but I'm surprised not to see the left-shift incorporated into an operand, because I thought ARM supported a left-shift in every operand or something.

The `ldr r0, .L2` is presumably because the 32-bit constant address of the string at .LC0 is hard to fit into an instruction. The separate `pop` for the return address is presumably because if it used `r1` in the same `pop` it would have been popped in the wrong order (because in the instruction encoding this is surely some kind of bitfield or something, not a variable-length list of 4-bit register numbers); this also clarifies that the first thing in the `push` or `pop` list is the one SP points at within the push/pop pair: the last one to be pushed and the first one to be popped. But why didn't it just pop it into `pc` rather than using two more instructions? I suspect the answer is what I saw earlier in the ATPCS: old ARMs needed an explicit BX to ensure a switch in instruction encoding.

Previously I was compiling for `armv7-a` (by the default configuration of my toolchain on the other computer), and I wonder if that resulted

in using the freer-form Thumb-2 instruction format, in which you *can* access the high registers. Indeed, *all* of these instructions enter into 16 bits, except for the immediate operands of the call instructions:

```
$ arm-none-eabi-objdump -d shl.o

shl.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
   0:   b538        push    {r3, r4, r5, lr}
   2:   1c0c        adds    r4, r1, #0
   4:   6848        ldr r0, [r1, #4]
   6:   f7ff fffe   bl  0 <atoi>
   a:   1c05        adds    r5, r0, #0
   c:   68a0        ldr r0, [r4, #8]
   e:   f7ff fffe   bl  0 <atoi>
  12:   00e9        lsls    r1, r5, #3
  14:   4301        orrs    r1, r0
  16:   4803        ldr r0, [pc, #12]   ; (24 <main+0x24>)
  18:   f7ff fffe   bl  0 <printf>
  1c:   2000        movs    r0, #0
  1e:   bc38        pop {r3, r4, r5}
  20:   bc02        pop {r1}
  22:   4708        bx  r1
  24:   00000000    .word   0x00000000
```

We can see that the `ldr` to get the constant has been compiled as a PC-relative reference, presumably to support position-independent code --- although I'm not sure how that word is supposed to get the address of the string in it in a relocatable way?

It's not; if I instead compile with `arm-none-eabi-gcc -S -mthumb -fPIC -O`, I get this instead:

```
        ldr     r0, .L2
.LPIC0:
        add     r0, pc
        bl      printf
...
.L2:
        .word   .LC0-(.LPIC0+4)
        .size   main, .-main
        .section        .rodata.str1.4,"aMS",%progbits,1
        .align  2
.LC0:
        .ascii  "%d\012\000"
```

That is, instead of storing the address of the string, it stores the PC-relative offset from the place where the string's absolute address gets computed. The (static) linker can freely relocate the string because the .LC0 relocation will fix up the word at .L2 when the final executable or shared library is built.

With a non-Thumb non-PIC compilation `arm-none-eabi-gcc -S -O`

shl.c the code for main() is instead:

```
main:
        @ Function supports interworking.
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        stmfd   sp!, {r3, r4, r5, lr}
        mov     r4, r1
        ldr     r0, [r1, #4]
        bl      atoi
        mov     r5, r0
        ldr     r0, [r4, #8]
        bl      atoi
        orr     r1, r0, r5, asl #3
        ldr     r0, .L2
        bl      printf
        mov     r0, #0
        ldmfd   sp!, {r3, r4, r5, lr}
        bx      lr
```

(Again, all of this is without #includes; thus the literal calls to atoi and printf.)

These are 32-bit instructions, and it seems like it's using the stmfd and ldmfd instructions (rather than push and pop) to load and store multiple values; presumably the sp! addressing mode is some kind of magic autoincrement/autodecrement addressing mode. The fact that sp is an explicit operand makes it sound like r13 is just another register and its use as the stack pointer was just a convention, but I don't think that was really true --- I think even old ARM interrupt handlers used r13 to save the registers of the thread being interrupted. (Certainly the ATPCS documents this as a thing that could happen in 2000.)

Some of the instructions have three operands instead of two, and the built-in shift I thought I remembered does see to exist here: orr r1, r0, r5, asl #3. Also it's worth noticing that these instructions are missing the s suffix in the disassembly:

```
$ arm-none-eabi-objdump -d shl.o

shl.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
   0:   e92d4038    push    {r3, r4, r5, lr}
   4:   e1a04001    mov r4, r1
   8:   e5910004    ldr r0, [r1, #4]
   c:   ebfffffe    bl  0 <atoi>
  10:   e1a05000    mov r5, r0
  14:   e5940008    ldr r0, [r4, #8]
  18:   ebfffffe    bl  0 <atoi>
  1c:   e1801185    orr r1, r0, r5, lsl #3
  20:   e59f000c    ldr r0, [pc, #12]   ; 34 <main+0x34>
  24:   ebfffffe    bl  0 <printf>
  28:   e3a00000    mov r0, #0
  2c:   e8bd4038    pop {r3, r4, r5, lr}
```

```
  30:   e12fff1e    bx  lr
  34:   00000000    .word   0x00000000
```

The Thumb assembly generated by GCC didn't have the `s` suffix on the instructions either, but the disassembly did; it turns out that Thumb instructions always update the flags, except for `mov` and `add` instructions with high registers. Also note that the disassembly spells `stmfd sp!`, as `push`, just like the Thumb version.

What about position-independent mutable data? It turns out to use the same scheme as position-independent immutable data, contrary to what I had expected from the ATPCS. I compiled this C module

```
static int accumulator;

int octal_digit(int digit) {
    accumulator = accumulator << 3 | digit;
    return accumulator;
}
```

with `arm-none-eabi-gcc -mthumb -S -O -fPIC` and got this remarkable result:

```
octal_digit:
        ldr     r3, .L2
.LPIC0:
        add     r3, pc
        ldr     r1, [r3]
        lsl     r2, r1, #3
        orr     r0, r2
        str     r0, [r3]
        @ sp needed for prologue
        bx      lr
.L3:
        .align  2
.L2:
        .word   .LANCHOR0-(.LPIC0+4)
        .size   octal_digit, .-octal_digit
        .bss
        .align  2
        .set    .LANCHOR0,. + 0
        .type   accumulator, %object
        .size   accumulator, 4
accumulator:
        .space  4
```

And this disassembly:

```
00000000 <octal_digit>:
   0:   4b03        ldr r3, [pc, #12]   ; (10 <octal_digit+0x10>)
   2:   447b        add r3, pc
   4:   6819        ldr r1, [r3, #0]
   6:   00ca        lsls    r2, r1, #3
   8:   4310        orrs    r0, r2
   a:   6018        str r0, [r3, #0]
```

```
   c:   4770        bx  lr
   e:   46c0        nop         ; (mov r8, r8)
  10:   0000000a    .word   0x0000000a
```

So we have .L2 in the code segment, just after the end of the function, which contains the BSS address of the read-write variable `accumulator`, relative to the instruction at .LPIC0. So first the program does a PC-relative `ldr` to fetch that read-only datum, and then it adds PC to the fetched datum to obtain the address of .LANCHOR0, which is the part of BSS that holds this file's zero-initialized static variables. This doesn't seem like it could possibly permit sharing the code segment, since the data at .L2 would need to be modified according to where (that piece of) BSS is positioned relative to where this code segment is mapped --- it would need a fixup by the dynamic linker.

This code also shows that the `str` instruction has its destination field on the *right*.

Without the `static`

```
int accumulator;

int octal_digit(int digit) {
    accumulator = accumulator << 3 | digit;
    return accumulator;
}
```

we get a different piece of code that refers to a global offset table; it sure isn't the scheme described in the ATPCS:

```
octal_digit:
        ldr     r3, .L2
.LPIC0:
        add     r3, pc
        ldr     r2, .L2+4
        ldr     r3, [r3, r2]
        ldr     r1, [r3]
        lsl     r2, r1, #3
        orr     r0, r2
        str     r0, [r3]
        @ sp needed for prologue
        bx      lr
.L3:
        .align  2
.L2:
        .word   _GLOBAL_OFFSET_TABLE_-(.LPIC0+4)
        .word   accumulator(GOT)
        .size   octal_digit, .-octal_digit
        .comm   accumulator,4,4
```

I mean this still seems to demand that this code be mapped at a fixed memory location relative to the `_GLOBAL_OFFSET_TABLE_` if it isn't going to be fixed up at load time. So, I don't know.

Even still, it seems like a relatively heavy price to pay for code segment sharing that instead of accessing a variable by saying

```
ldr     r3, [pc, #12]
```

you have to say

```
ldr     r3, [pc, #12]
add     r3, pc
ldr     r2, [pc, #something]
ldr     r3, [r3, r2]
ldr     r1, [r3]
```

and also have a per-reference offset stored somewhere the static linker can fix it up; and so I wonder how often it is really worth it.

Costs of accessing variables allocated statically

An interesting thing about this way of referring to variables (or that described in the ATPCS) is that it reverses the traditional costs of referring to statically and dynamically allocated variables. From the 1940s through the 1980s, accessing a statically allocated variable was cheap: it was at a known, constant address in memory, which could be baked into the instruction; while accessing a variable allocated dynamically, for example on the stack, required indexing off the stack pointer or some other kind of base pointer, which itself had some extra cost to create and maintain. (Worse, until around 1970, there were a significant number of computers where an indexed memory access required self-modifying code, because they didn't have index registers.) But in this case we see that accessing two dynamically-allocated variables can be as simple as `lsl r2, r1, #3`, while accessing a single statically-allocated variable requires a five-instruction watusi.

At first blush this sounds like a straightforward case of architectural evolution, but it isn't really. RAM is just a bunch of registers, after all. There are only a couple of minor details of the ARM architecture that contribute to this situation: it has an efficient encoding for PC-relative addressing (like amd64, unlike i386); loading from a constant pointer requires three instructions (movw, movt, ldr) instead of one; and you only have 16 registers you can address directly, while everything else is much slower, because CPU speed has zoomed way ahead of RAM speed.

Rather than a change in architecture, though, it's mostly an evolution of the execution model. It's just a different way of using the machine that prioritizes different tradeoffs. You could totally use a PDP-10 or 6502 in such a way: mostly reserve the 6502 zero page for local variables and frame pointers and whatnot rather than global variables, and index all your "statically allocated" variables off one of those registers so that separate processes sharing an address space store their mutable state in separate "segments". And although the Cortex-A7 ARM in your cellphone might have gigabytes of RAM and a deep cache hierarchy, the Cortex-M0 in a small STM32 doesn't see a whole lot of difference between its speed of accessing CPU registers and accessing the on-die SRAM, except that it may need to run several instructions to compute an address into the on-die SRAM.

Reading other stuff

ARM published an "assembler user guide" in 2001 that explains the assembly language fairly comprehensively (354 pages!). Its chapter 4 is the ARM instruction set reference, and chapter 5 is the Thumb instruction set reference. It's marked as "superseded" on ARM's unusably bad website, but without a link (that I could find) to the superseding version. On the 15th page, it explains what ARM and Thumb are; on the 16th page, it describes the register-banking scheme used to separate user and supervisor (kernel) mode. It has a wealth of information about the historical development of the instruction set, including explaining literal pools and whatnot.

However, this version of the book lacks such crucial features as 32-bit-wide Thumb-2 instructions and if-then-else blocks.

There's a decent but unfinished 15-page tutorial by Carl Burch under CC-BY-SA; it explains the -s suffix on instructions, the absence of an integer division instruction (though not the existence of extensions that have it), the built-in shift, the `umull` instruction, the limitations on `mov` immediate constants, `mvn`, `{ldr,str}{b,}`, `{ld,st}m{i,d}{b,a}`, all the ALU instructions, conditional execution, all the condition codes, all the addressing modes (including examples of scaled-register-offset and immediate-post-indexed addressing), etc.

Despite this admirable level of comprehensiveness, it's imperfect; it seems to be unfinished, stopping after explaining the above but before describing function call and return, and it doesn't cover Thumb at all. Also, the "hailstone sequence" example program has a bug in it in which the `ands` instruction overwrites the accumulator, preventing the program from ever working, and at one point it erroneously says it's jumping to the beginning of an array of doublewords. And, unfortunately, the tutorial uses ARM's assembly syntax instead of Gas's.

Azeria Labs wrote an ARM assembly cheat sheet, though it's mostly focused on breakins, and they want to charge you for the full-resolution version; it's associated with a poorly-written error-filled tutorial with totally kool diagrams. The discussion of it in 2017 on the orange website links to a lot of better resources.

https://www.coranac.com/tonc/text/asm.htm?

http://www.davespace.co.uk/arm/introduction-to-arm/not-trivial.html?

Topics

- Programming (p. 3658) (286 notes)
- Instruction sets (p. 3526) (40 notes)
- Assembly language (p. 3328) (25 notes)

Plato was not particularly democratic; ἄρχειν is not "participating in politics"

Kragen Javier Sitaker, 2014-04-24 (5 minutes)

There is a common misquotation of Plato, which recently came up yet again on Twitter:

> One of [the] penalties of refusing to participate in politics is you end up governed by your inferiors.

Or, worse:

> The price paid by good men for indifference to public affairs is to be ruled by evil men.

This is a fake Plato quote. Plato did say something kind of like this, but there's a really major difference.

What did Plato really say?

Allan Bloom's precise translation says:

> and the greatest of penalties is being ruled by a worse man if one is not willing to rule oneself.

(You could misread this as "if one is not willing to control oneself", but the Greek text doesn't admit that reading; it clearly means "is not willing, oneself, to rule.".)

Where are people seeing this fake quote?

You can find both versions of this fake quote at misquote-laundering sites like Brainyquote.

You can also find this quote in misquote-laundering web sites and publications similarly unconcerned with correctness, such as those by the Brookings Institution.

Where did the fake quote come from?

The real quote in context, from Benjamin Jowett's loose translation , is as follows:

> Wherefore necessity must be laid upon them, and they must be induced to serve from the fear of punishment. And this, as I imagine, is the reason why the forwardness to take office, instead of waiting to be compelled, has been deemed dishonourable. **Now the worst part of the punishment is that he who refuses to rule is liable to be ruled by one who is worse than himself.** And the fear of this, as I conceive, induces the good to take office, not because they would, but because they cannot help --not under the idea that they are going to have any benefit or enjoyment themselves, but as a necessity, and because they are not able to commit the task of ruling to any one who is better than themselves, or indeed as good.

This is Socrates speaking to Glaucon about three-quarters of the way through Book I of the Republic, in what's usually known as section 347c. You will note the very large difference between "refuses to rule" and "refuses to participate in politics"; anyone who endorses a candidate, after all, is participating in politics, but only the leader of the government rules.

The relevant phrase in the Classic Greek original reads:

> τῆς δὲ ζημίας μεγίστη τὸ ὑπὸ πονηροτέρου ἄρχεσθαι, ἐὰν μὴ αὐτὸς ἐθέλῃ

ἄρχειν·

(Perseus has put parallel texts online.)

WikiQuote, the free world's answer to profiteering falsehood sites like BrainyQuote, has several paragraphs of discussion about this misquotation.

At one point I misremembered Plato as saying that this was the penalty for "not being the tyrant", but that's going too far to the other extreme, to the extent of inaccuracy. The Republic talks about several different forms of government, including absolute power by one person (the "tyrant", but without the modern derogatory connotation), and there's no indication that Plato is talking specifically about seizing absolute power; ἄρχειν just means "to lead", in a very general sense, but here referring specifically to governing.

But ἄρχειν definitely does *not* mean just "to not be indifferent to public affairs" or "to participate in politics".

The older version of the falsified quote seems to be this:

> The price paid by good men for indifference to public affairs is to be ruled by evil men.

I have a vague memory that this was invented by an American political group in the mid-20th Century. The phrase "indifference to public affairs" seems to have been stolen from Hannah Arendt, in her 1951 *The Origins of Totalitarianism*, near the middle of the book, which more or less directly contradicts the fake Plato quote:

> Indifference to public affairs, neutrality on political issues, are in themselves no sufficient cause for the rise of totalitarian movements. The competitive and acquisitive society of the bourgeoisie had produced apathy and even hostility toward public life not only, and not even primarily, in the social strata which were exploited and excluded from active participation in the rule of the country, but first of all in its own class.

Topics

- History (p. 3500) (71 notes)
- Politics (p. 3639) (39 notes)
- Facepalm (p. 3450) (24 notes)
- Philosophy (p. 3628) (2 notes)
- Plato
- Classics

Starfield servo

Kragen Javier Sitaker, 2016-08-30 (updated 2018-11-07) (13 minutes)

You can use a camera to measure deep subpixel movements by using moiré patterns generated by geometrical optics, permitting extremely inexpensive sensing and servomechanisms with several degrees of freedom all the way down to submicron scales.

Suppose you have two sheets of A4-size acetate transparency film printed almost entirely black on a 1200-dpi printer, with occasional randomly placed transparent pixels piercing the otherwise uninterrupted darkness. Specifically, about one of every 2048 pixels is transparent; the other 2047 are opaque black. This means that out of all 139 million pixels on either sheet, only about 68000 are transparent.

Let's lay one of the sheets on top of the other with some integer X and Y pixel offset between them, but no relative rotation. The number of possible ways we can do this is about 9900 in X and 14000 in Y for a total of about 139 million. The odds are overwhelming that about one out of every $2048^2 = 4\,194\,304$ pixels will happen to have two transparent pixels directly on top of one another, and will therefore be transparent. If we view a bright backlight through the two sheets, we can easily detect this pixel. Each such pixel that we measure reduces the number of candidate alignments by about a factor of 2048; any set of more than about four such pixels is adequate to identify the configuration uniquely, again with overwhelming probability.

All in all, for each configuration, there are about 33 such pixels, several times more than enough.

These discrete configurations are separated by 21 microns, an inch divided by 1200.

Let's consider the more general case, where the displacement happens in more degrees of freedom and is continuous rather than discrete. Allow the sheets to rotate in two dimensions and have a perspective difference between them. In this case, almost all pixels in one sheet will overlay parts of four pixels in the other sheet, which increases the number of pixels with some light leaking through by about a factor of 4, up to about 133. It also means that we can measure more than just whether a given pixel is transparent or not; we can distinguish degrees of transparency. How many degrees we can distinguish depends on the sources of noise, but we can probably reliably measure something like 64 gray levels for the pixel. This means we can measure the displacement of any given overlapping pixel of something like a 64th of a pixel, which would increase our measurement precision to under 400 nanometers if geometric optics were the truth. But geometric optics breaks down before that point, so we probably can't do better than about a micron, at least with visible light.

Note that we're up to about 2400 bits of signal here (133 * (6 + 12)), although it's heavily redundant. In the worst case, we're trying to estimate nine degrees of freedom: the distance from the camera to the scene, two degrees of freedom of camera angle, and six degrees of freedom of relative position and orientation of the two transparency

sheets. We're hoping to calculate the relative position and orientation accurate to within about one part in 300 000, which is 18 bits; the other three degrees of freedom might need to be estimated to a similar relative precision even if we don't care. The upshot is that we need to estimate 162 bits of information from 2400 bits of data, which seems eminently feasible.

How much camera resolution do we need for the pattern of pinholes we see to be unique to that configuration? We might need enough resolution to be able to usually separate the light from different pinholes. This might require something like 256 × 256 pixels on the sensor plane, so that most of the 133 pinholes are by themselves in a row and by themselves in a column. This is a few hundred kilobits of information.

How can we make it practical to estimate this information, even if it is in principle contained in the camera image? In principle, you could simply measure the distance from the 2^{162} or so interestingly different configurations and pick the one with the lowest error. Hopefully this is not necessary in practice. Here are some techniques that will probably work:

• First identify which pixels are bright, estimating the pixel-rounded displacements from that, then compare their brightnesses to get subpixel data. For a given camera angle, scene distance, and rotation, this cuts the number of configurations down to only 139 million.
• Do the coregistration computation in Fourier-transformed spatial frequency space rather than in the spatial domain. What we're doing in physical space here amounts to multiplying the two transparencies pointwise in the spatial domain, which is equivalent to convolving their frequency spectra. If we guess roughly the right rotation, maybe we can roughly deconvolve the frequency-domain signal of the product with the frequency-domain signal of one of the transparencies, giving us an estimate of the frequency-domain signal of the other — hopefully including its phase shift.
• Instead of one single-scale pinhole field, we could divide the sheets into areas of "pinholes" of different sizes, or merely mix different sizes of pinholes together, from the 21-μm single-pixel pinholes up to 21-mm finger holes. Then we can use the much smaller number of significantly different configurations of the bigger holes to tell us what neighborhood to search in for configurations of the smaller holes, in this example through perhaps ten power-of-two hole sizes. There are only about 140 interestingly different translational configurations of 21-mm holes, about 12000 if you include 2D rotation, and half a million if you include 3D rotations. These numbers make exhaustive search feasible. (By necessity the larger holes will need to be distributed somewhat more densely, unless you try to recognize the patterns of smaller holes visible through them instead of just comparing them with each other.) Larger holes may also make the Fourier approach more feasible.
• If you're tracking motion, you can use position estimates from previous frames to find which neighborhoods of configurations to search in in a new frame. You can extend this to many simultaneous hypotheses using particle filters and the like.
• Rotations and perspective distortion are less important in small neighborhoods; if you examine a small neighborhood, you don't need

to consider nearly as many rotations. Like large holes, this might benefit from higher density of holes. Consider a circular neighborhood of radius 1150 pixels (about 25 mm), which will contain on average about four million pixels and four of these coincidental pinholes; you can rotate it by up to about 440 microradians before the pattern of pinholes changes. If you were to increase the density of pinholes in the original from one per 2048 to one per 64, then you would have four coincidental pinholes every 2048 pixels, contained in a circle of radius about 72 pixels, which wouldn't change constellations until it had been rotated by about 2.2 milliradians.

• If you compute the Delaunay triangulation of the bright points, you should be able to eliminate the necessity to try many different rotations. Many such tests on clouds of detected features are known in the computer vision literature. You can probably even hash some aspects of the feature graph.

The techniques that involve increasing the pinhole density will decrease the information available per pinhole and probably require more pixels on the image sensor, but they provide more information overall (at least until the density of holes goes above, I think, 1/e).

Sparkly surfaces

A related approach is to use the reflections from a sparkling surface illuminated from a single direction for the feedback. For example, a piece of sandstone in the sun. If the light source and camera are fixed, the pattern of sparkles gives fairly precise information about the two angles of the surface to the axis bisecting the angle between the direction to the light and the direction to the camera; the rotation of the pattern on the focal plane gives fairly precise information about the rotation of the surface *around* that axis; the position of the pattern gives fairly precise information about the translation of the surface perpendicular to that axis; and the scale of the pattern gives very crude information about the translation of the surface *along* that axis.

Of course, you have to start by calibrating the system with a massive database of sparkle patterns from that surface from many different angles.

The system as described can be improved in several ways:

• by using more than one light, ideally in separate frames of video, in order to provide redundant information and in particular to triangulate to get more precision in the imprecise dimension;
• by making the background of the sparkles as dark as possible;
• by making the bright facets themselves slightly concave, both so that they generate a brighter sparkle and so that it is visible over a smaller part of the surface's rotation;
• by making *some* facets sufficiently convex that they can be used to get an approximate angle fairly quickly, thus speeding the search for the precise position;
• by making the sparkles very sparse, so that there are always a few visible, but only a few;
• by defocusing the camera to optically convolve the sparkle pattern with a sharp-edged bokeh, permitting the use of many pixels along the border of the bokeh from a given sparkle to find its position on the focal plane to much better than a single pixel of precision, a feat which is not possible if the sparkle is perfectly focused and thus just

saturating the fuck out of a single focal-plane pixel;
• by (as described above) placing a moiré-generating screen between the camera and the sparkling surface, close to the sparkling surface, in order to distinguish displacements of a screen thread or so, which may be much less than a pixel;
• by using a combination of large facets for better angular resolution with small facets for better spatial resolution.

Riding the Sarmiento train to Once, when it was stuck in a station for a while with the doors open, I observed that moving my head by two fingerswidths (≈20 mm) caused a certain sparkle in the floor to appear; moving it two fingerswidths further caused it to disappear. The sparkle on the floor was about 3 m away from my head, suggesting an angular resolution of some 6 milliradians from simply thresholding that single sparkle; presumably you could get down to 1 mrad by comparing the relative brightnesses of several.

The sharp boundary of the sun's disc (which should be about 9.3 milliradians across, with an edge of 0.5 milliradians or less) was not evident, suggesting that the sparkle (or something, maybe some clouds) was introducing several milliradians of divergence.

Achieving a divergence of 0.5 milliradians (1.7 minutes of arc) at a wavelength of 400 nm from a blue LED requires a beam waist of at least about 0.4 mm, so if you want that much angular resolution, your facets need to be at least that big. (And your light source needs to, if not subtend that little of the field of view of the sparkly surface, at least have significant energy in spatial frequencies that high — for example, the sun's sharp boundary. A fuzzy Gaussian light source is kind of the worst case for a light source of a given size, although of course uniform ambient illumination is the worst case.)

Topics

Más pensamientos acerca de diseñar un calefón solar

Kragen Javier Sitaker, 2012-10-15 (5 minutes)

Así que la radiación de cuerpos negros no nos conviene tanto. Pero tiene una caída bastante fuerte a su frecuencia máxima; hay una frecuencia máxima, más arriba de lo cual efectivamente no hay nada de radiación. *Podría* ser posible formular una ventana que efectivamente no transmite nada más allá de, no sé, unos 8000nm, para que la pérdida de calor al cielo subirá por mucho más que unos 7% durante esa subida de 43 grados hasta 49 grados. Pero esto parece bastante difícil para mí en este momento, porque hay pocos plásticos distintos en uso común.

(En algún sentido, esto es más o menos lo mismo que un superficie selectivo común, para lo cual, para subir la eficiencia y la temperatura, queremos mucho menos emisividad en las bandas LWIR, donde emite el panel por su temperatura, que en las bandas visible, donde el sol transmite la mayoría de su energía. Pero es mucho más exigente, porque estamos buscando una subida de emisividad de alrededor de una orden de magnitud por un cambio de frecuencia de unos 2%, entre 316 y 322 K, mientras los superficies selectivos normales cambian su emisividad por un factor de solo 4 con un cambio de frecuencia de una orden de magnitud.)

Otro tipo de ventana que permite pasar el LWIR se hace de polietileno, a veces fortalecido con un mosquitero. Hay un patente acerca de esto: http://www.google.com/patents/US5493126. Una gran desventaja para nuestro propósito es que el ultravioleta solar hace mierda a polietileno sin protección en pocas meses. (Creo que es por eso que los caños de polietileno son negros, para protegerlos del sol.)

Pero bueno, todo eso me hace pensar que probablemente no es práctico hacer un colector solar para agua caliente que es inherentemente seguro contra subir a temperaturas demasiado altas. Lo típico es usar una válvula automática que mezcla agua fría con el agua caliente mientras sale del termotanque, pero también me parece que se puede resolver el problema con controlar bien la temperatura del agua que entra al termotanque. Es que hay que tener un sistema de control lo suficientemente confiable que el riesgo de muerte o lesiones serias baja a un nivel aceptable.

Mi papá también sugerió el uso de válvulas de viejas lavarropas para poder controlar el agua, lo cual me parece una idea bastante inspirada. Además del tema de un posible recicle, capaz que las válvulas que usan para reparar lavarropas serán mucho más baratas (porque más comunes) y fáciles de conseguir que otras válvulas de capacidad parecida.

También sugerió usar un termostato de una heladera para controlar las válvulas, ya que suelen ser adaptables a temperaturas así. Para mí eso será mucho más lío que simplemente usar un microcontrolador.

Sería re interesante de un punto de vista poética o artística poder fabricar la cosa entera de materiales reciclados, pero me parece que puede ser un poco difícil reciclar microcontroladores, porque se suelen deshabilitar el reprogramamiento.

Así que ahora estoy pensando en los próximos partes:

• Panel colector, de

• caño:

• caño de polietileno en espiral, atado a un par de palos o atado con plástico; o
• chapa pintado en negro, posiblemente con oxido cúprico, con caños de cobre bronceados por atrás;
• posiblemente, caja aislada para permitir que el agua pueda subir a 43 grados hasta cuando el aire está a 0 grados. Con 800W/m² de sol, esto implica una resistencia térmica de R = 0.054 K m²/W, lo cual es muy poca aislación: capaz que un simple espacio de aire con vidrio o acrílico arriba, o simplemente envolver un superficie irregular con ese polietileno suave que usan para atar cartones en pallets, atrapando un poco de aire;
• posiblemente, un reflector en vez de una caja aislada, para que la temperatura suba lo suficiente sin necesitar aislación térmica.
• válvulas de lavarropas;
• dos o tres termistores apto para uso en temperaturas entre 0 y 50, con repetibilidad de medida de un grado o menos;
• cables eléctricos para conectar las válvulas y termistores con un microcontrolador;
• un microcontrolador;
• un fuente para el microcontrolador;
• 4 barriles de 200 litros, con caños suficientes para conectarlos;
• telgopor o lana de vidrio suficiente para aislarlos hasta unos 20 cm.
• soportes de algún tipo? Soldados de acero?

Para mí, primero tendremos que armar prototipos para averiguar:

• si determinado diseño de panel puede lograr la temperatura que queremos con el sol, para cada diseño que vale la pena probar;
• como calibrar los termistores;
• qué tan grueso caño (y válvula) necesitamos para el flujo apto, para llenar el tanque dentro de poco tiempo;
• qué tanta electricidad necesitamos para operar las válvulas y el micro;
• si hay lugar en el techo de Vi para 800kg de agua;
• y qué más?

Topics

• Physics (p. 3632) (119 notes)
• Energy (p. 3438) (63 notes)
• Thermodynamics (p. 3747) (49 notes)
• Household management and home economics (p. 3504) (44 notes)
• Solar (p. 3717) (30 notes)
• Español (6 notes)

On hanging out with cranks

Kragen Javier Sitaker, 2008-04 (4 minutes)

comment to http://science-professor.blogspot.com/2008/04/someone-should-study-this.html

This is a fantastic comment thread! I especially appreciate the reference to the MJT book.

I've hung out with a few cranks in my time. One person I know, who's extremely well-respected in his field (which is not a science), is trying to publish a book on an improved version of the Titius–Bode law. I suggested that when he's sending a draft to researchers who have expressed interest, he should leave out the chapter on how the planets' orbits relate to musical scales, at least.

And I've had some lovely conversations with Ed Fredkin, a brilliant computer scientist who says, "I'm very interested in physics, but physicists are very uninterested in me being interested in physics." Maybe one of these days he will turn out to have been right, that the universe really is some kind of simple automaton. His work has gone on to inspire another famous crank, Stephen Wolfram, who (again) has done some solid academic work — before giving up altogether on academic norms like giving Fredkin credit and not suing your collaborators for publishing papers on their work.

And a few weeks ago, I had the privilege to sit next to Carl Hewitt in a meeting. Some ideas he was exploring in the 1970s have been the basis for a large fraction of the work in programming languages in the 80s, 90s, and today (Erlang, the latest hot language, is largely the latest realization of his Actors model), so he's the real deal. He gave me a copy of a paper he's working on about nonstratified inference in paraconsistent logics; I haven't been able to make heads or tails of it yet, partly because my grounding in symbolic logic is pretty weak. So what's his crank-hood? He was banned from editing Wikipedia because he kept editing physics articles to explain the importance of the Actors Model in physics. (Maybe he'll turn out to be right, but I think for now there are problems with Bell's Inequality, which is also a problem with Wolfram's ideas.)

Worse are the cases where cranks like Martha Rogers have actually gotten their crazy ideas inside the academy, destroying it from within.

(I haven't met Wolfram or Rogers.)

I have a mailing list to publish my own crazy ideas, but I try not to get too attached to them. I hope that some of them might be significant enough to turn into academic publications, but I'm constantly terrified that I'll turn into a crank myself. The major thing distinguishing me from the people I've listed above, then, would be that they have accomplished some significant things before retiring into crankhood, while I'm just some guy.

It would probably be helpful for people like me to have a guide to the warning signs of crankhood, with examples. Obviously, if this just allowed cranks to take on protective coloring and infiltrate academic mailboxes with greater ease, it would not be a public service; but if it could enable some of us non-academics to recognize

when we're being insufficiently critical of our own ideas, and perhaps in the occasional case that we have some idea that's actually worthwhile, to figure out how to distinguish it from our less worthwhile ideas and then present it in a way that its merit could be detectable — that might have the effect of actually diminishing the amount of wacko mail that comes in.

Oh, and there was this time I was at Google just after lunching with some friends there, sitting down outside with my laptop on the edge of the grounds before leaving, and the security guards were politely turning away a guy who just *had* to present his just-patented invention to *somebody*, *anybody* at Google — but he didn't actually know anybody there, and wasn't willing to disclose anything about his invention.

Topics

Reflections on rebraining calculators with this RPN calculator code I just wrote

Kragen Javier Sitaker, 2017-04-11 (4 minutes)

Over the last three nights I've written a substantial part of an RPN calculator in C in order to rebrain a four-function pocket calculator with a somewhat more reasonable calculating facility. Last night I spent about 1½ hours on it; the night before, about an hour; the night before that, about 3½ hours; for a total of six hours.

Performance and battery life projections

The basic user-level functionality mostly works on Linux with a sort of testing stub implementation of keyboard and screen, although there are a couple of egregious bugs. So it should be adequate for estimating performance.

Using it more or less continuously for 90 seconds, on Linux, compiled with optimization, executed 340,838 amd64 instructions. Just starting and stopping it executed 162,347 instructions, for a difference of 178,491, or almost 2000 instructions per second. Running it with four numeric-entry keystrokes executed 169,718 instructions, or 7371 instructions beyond setup and teardown, about 1800 instructions per keystroke; adding 3 + 4 (four keystrokes) took 168,824 instructions, or 6477 instructions beyond setup and teardown, about 1600 instructions per keystroke. If we round this to 2000 amd64 instructions per keystroke or per second, which will probably be more like 4000 AVR instructions, and if the AVR uses 8 nJ per instruction, this is about 32 μJ per keystroke (or per second).

My intent here is to replace the calculator's existing chip with an AVR or something similar, without replacing its keyboard, LCD, and maybe even battery.

Some sample batteries:

- the 300 mAh 1.2 V NiCd AA cell that powers my garden light: 1.3 kJ or 11000 hours
- a CR2032 lithium coin cell: 240mAh at nominally 3 V but down to 2 V; supposedly 192 mWh/g and 3.0 g, so 576 mWh or 691 J or 6000 hours. The datasheet makes it look like it's reasonable to suck up to about 10 or 20 mA out of the battery from time to time without unduly hurting its capacity.
- a regular non-alkaline 1.5 V AA cell: the Energizer E91 datasheet says that it's 3000 mAh down to 0.8 V if discharged at 25 mA. If it were 1.5 V until the end, that would be 16 kilojoules, which is unreasonably high since it would actually be higher energy density than the lithium cell; we can probably estimate half that or 8 kJ or 69000 hours.

I conclude that the AVR's active power draw will not consume a significant amount of battery unless the code gets a lot less efficient or I program it to do much more complicated things.

However, the AVR can run at 20 MIPS, which requires over 100

milliwatts; at this speed, it could drain some of the batteries mentioned above in an hour or two.

Possible features

So far all I've done is a basic four-function calculator, in part because the keyboard on the thing I want to rebrain is not very large. But it already offers these features above and beyond a standard four-function calculator:

- Deep expression nesting;
- Visual feedback on pending values (by displaying the whole stack, or as much of it as will fit);
- RPN syntax;
- Character-by-character number error correction.

It should be feasible to also include features like:

- Stored programs, including iterative ones;
- Named variables;
- Measurement units (i.e. dimensional analysis);
- Undo;
- Persistence;
- Vector values with broadcasting.

Topics

Range literals

Kragen Javier Sitaker, 2014-04-24 (6 minutes)

Range literals

Python syntax and semantics, considered as executable pseudocode, has a curious wart relating to ranges of numbers. Because it's so common to store data in arrays in Python (called "lists"), this wart comes up often.

This is: if we want to iterate over the numbers 1 to 15 (excluding 15), we have to write:

```
for i in range(1, 15):
    ...
```

But if we want to do something `f` with the corresponding elements of an array `arr`, we must write:

```
f(arr[1:15])
```

It's possible to store the object encapsulating those indices from 1 and 15 in another object and pass it to `arr` later. You would think you would do it like this, like in Octave:

```
idx = 1:15
f(arr[idx])
```

but actually that's invalid syntax and you have to do it like this:

```
idx = slice(1, 15)
f(arr[idx])
```

This syntax and function names are far from self-explanatory. Far better, in terms of readability, would be to write this:

```
for i in 1:15:
    ...
```

Or even this:

```
for i in (1:15):
    ...
```

And, in terms of consistency, it would be far better for the expression syntax within [] to be the same as the expression syntax outside of them. This would speed up the learning of Python.

This is one of the few obstacles to universally using Python in place of more traditional pseudocodes in the explanation of algorithms. As things are now, it is much clearer to write the traditional:

```
for i = 1 to 14
    ...
end for
```

than the Python

```
for i in range(1, 15):
    ...
```

This proposed universality of the slice syntax would simplify this, although you'd still have to explain that `1:15` doesn't include 15.

You could maybe make the argument that the extra redundancy improves the Python interpreter's error-reporting ability. That is, there are some cases where someone might accidentally write a range literal, or something like one, which would currently be a syntax error, which would give you a less clear syntax error or even broken code. I can think of a few possibler cases of this — when you want to iterate over an infinite range, for example:

```
for i in 2::
    ...
```

Or worse:

```
for i in ::
    ...
```

I think these can be basically solved by always requiring parens around the range literal, as is currently done with generator expressions, with the added looseness that the parens can also be square brackets:

```
for i in (2:):
    ...
for i in (:):
    ...
```

Negative indexing

Python's array indexing has an additional semantic wart, shared with Perl and I think originally from APL, which is that the use of negative indices counts from the end of the array. This has occasionally worsened bugs in my Python code, but mostly it's just unnecessarily mysterious for newcomers to Python.

I think a better approach would be to have a special object, called, say, `end` or `last`, which can have integers subtracted from it to produce indices that are interpreted to mean "N from the end". You wouldn't be able to iterate directly over ranges containing it, but that's not really a problem.

So, for example, you could say:

```
while line.endswith("\n") or line.endswith("\r"):
    line = line[:end-1]
```

which I think is clearer than the current real Python code for doing this. But you couldn't say:

```
for i in (:end-1):
    print line[i]
```

This would also eliminate the need for omitted indices in slices. Rather than `arr[:]`, you could write `arr[0:end]`, at a slight cost in verbosity but a great gain in clarity, at least to my eyes.

Greek letters

JavaScript allows you to use Greek letters in your variable names, which lets you write code like this (from http://canonical.org/~kragen/sw/dev3/spirals.html):

```
var Δθ = 1/r/2
  , Δr = slope * Δθ
;
if (Δr > 0.5) {
  Δr = 0.5;
  Δθ = Δr / slope;
}
if (neg) Δθ = -Δθ;

r += Δr;
θ += Δθ;
```

This code would be substantially wordier and less clear if it had to be written without Greek letters, and the same is true for a substantial amount of code that is more or less mathematical in nature. (You could argue that Greek letters are going to be intimidating for people who aren't familiar with math, but those people are going to have to learn the math before they understand the code anyway.)

I would argue that being able to use Greek letters in pseudocode often improves its clarity substantially.

An example

Here's a function from the scrypt paper, rendered in the traditional ALGOLish pseudocode from the paper, though without the LaTeX typography:

```
Algorithm ROMix_H(B, N)
Parameters:
       H              A hash function.
       k              Length of output produced by H, in bits.
       Integerify     A bijective function from {0,1}^k to {0,...2^k-1}.
Input:
       B              Input of length k bits.
       N              Integer work metric, < 2^{k/8}
Output:
       B'             Output of length k bits.
Steps:
 1: X ← B
 2: for i = 0 to N - 1 do
 3:   V_i ← X
 4:   X  ← H(X)
 5: end for
 6: for i = 0 to N - 1 do
 7:   j ← Integerify(X) mod N
 8:   X ← H(X ⊕ V_j)
```

```
 9: end for
10: B' ← X
```

Here's the Python version, with the notational improvements suggested above:

```
def ROMix(H):
    """H: a hash function producing k bits.
    Returns a function(B, N):
    B: Input suitable for H.
    N: Integer work metric, < 2**(k/8)
    returns an output of length k bits.
    Uses a bijective function Integerify from {0,1}^k to {0,...2**k-1}.
    """

    def f(B, N):
        V = [B]
        while len(V) < N:
            V.append(H(V[end-1]))

        X = V[end-1]
        for i in (0:N):
            j = Integerify(X) % N
            X = H(X ^ V[j])

        return X

    return f
```

I think this is substantially clearer than the original pseudocode version, especially if you already speak Python or C. It would be better if the inner function `f` didn't have to be named, or if the return statement could be placed before its definition (as in JS).

Topics

• Programming (p. 3658) (286 notes)
• Syntax (p. 3738) (28 notes)
• Python (p. 3671) (27 notes)

Passivhaus seasonal thermal store

Kragen Javier Sitaker, 2017-03-02 (updated 2017-03-07) (2 minutes)

If a house is sufficiently insulated, it need not actively reject heat outdoors, for example with air conditioners. It is sufficient to store the heat generated within by the inhabitants and their devices (lights, appliances, etc.) in a seasonal thermal store until the environment is cool enough for the heat to flow to it passively, given the opportunity.

What's the worst-case scenario? One person dissipates around 2000 kcal/day, about 100W. Household appliances are usually a few times this; let's say 400W in total.

Let's suppose we store cold in the form of ice during the winter, and that it will have to last six months.

400 J/s · 6 months · 30 days/month · 86400 s/day = 6.2208 GJ

Melting ice consumes 75 kcal/kg.

6.2 GJ/person · (1/75) kg/kcal · 4.2 cal/J = 0.35 Gg/person = 350 Mg/person

350 tons of water: 350 m^3, a little more due to the expansion of ice. At a height of 5 m, that would be 70 m^2: basically a bedroom or two of space. Or, if it were below an entire floor of 16m × 16m = 256 m^2, it'd be 1½m of height of ice. Per person. If four people live in the house, it'll be 6 meters of depth of ice below the house — 1 meter per month.

If we're talking about storing cold in a material without a change of phase, we'd have just 20° or 30° of ΔT, which is 20–30 kcal/kg, if it were water, or 7–20 kcal/kg if it were something else — in total 4× to 10× the ice, by weight — 1400 to 3500 tons per person. Nevertheless, it could be a similar volume of rock or soil, since they are some 2× to 4× as dense as ice.

In practice, it's probably possible in most places to radiatively couple to a comfortable temperature almost every day, so a six-month thermal store is extreme overkill. However, these calculations show that even a six-month purely passive thermal store is feasible.

Topics

• Physics (p. 3632) (119 notes)
• Thermodynamics (p. 3747) (49 notes)
• Household management and home economics (p. 3504) (44 notes)
• Cooling (p. 3393) (15 notes)
• Phase change materials (p. 3627) (8 notes)

Nova RDOS

Kragen Javier Sitaker, 2017-06-15 (22 minutes)

Bob Supnik got a revocable hobbyist license for RDOS from Dirty Genitals identical to the DEC OS/8 license

RDOS for the Nova seems to have been multitasking; it has error messages like OUT OF TCB'S and ERROR IN USER TASK QUEUE TABLE
also I am inferring a hierarchical filesystem from DIRECTORY DEPTH EXCEEDED

I'm `dd conv=swab`ing the image from http://simh.trailing-edge.com/kits/rdosswre.tar.Z
I guess this OS is from 1984 though. I wonder if it's really for a Nova?
it looks a lot more like TOPS-10 or something than it does like Unix

NUL-terminated strings, ^M for newlines, things that appear to be device names like MT0: MT1: MT2: CT0: CT1: CT2:, allusions to using matching pairs of <> in filenames (presumably to delimit the
name of the directory)
two-letter filename extensions, ALL CAPS
command flags leading with /
different file types, user-visible contiguous file allocation
misspelled error messages
SYSGEN
octal addresses
I am inferring a 16-bit word-addressed memory?
up to a mebiword of memory
horrifying interactivity: DUAL PROCESSORS (IPB)? ("0"=NO "1"=YES)
ENTER RTC FREQ (1=10HZ 2=50HZ 3=60HZ 4=100HZ 5=1000HZ)

WANT STANDARD(11 INCH) ?^M^@ENTER FORM SIZE IN LINES(1-143) ==>^@ENTER LINE NUMBER OR CR ==>^@

CHARACTER PRECEEDING A "(" MUST BE AN OPERATOR^M^@LEGAL OPERATORS ARE +, -, !, #, &, @, <, >^M^@NESTED OR MISSING BRACKET(S)^M^@")" MUST BE PRECEEDED BY A NUMBER OR LETTER^M^@
oh hey, here are "shell script commands" from DG RDOS: DELETE BADSP.RB SYS000.RB

I think this is a formatted output template: **##### TOTAL ERRORS, ##### PASS 1 ERRORS
COPYRIGHT (C) DGC 1977,1978,1979,1980,1981,1982,1984,1985ALL RIGHTS RESERVED

Here's a domacro command line:

    DUMP/V BACKUP:7 URDOS<A B C I O>.LB ALMSPD.<SR RB> BACKUP

This suggests the <> aren't directory names but limited wildcards.

Oh hey, here's some source code; I think their macro assembler defined
the instructions as macros!

```
; COPYRIGHT (C) DATA GENERAL CORPORATION 1977, 1978, 1979, 1980, 1981, 1982
; 1983, 1984
; ALL RIGHTS RESERVED.
; LICENSED MATERIAL-PROPERTY OF DATA GENERAL CORPORATION.
; THIS SOFTWARE IS MADE AVAILABLE SOLELY PURSUANT TO THE TERMS OF A
; DGC LICENSE AGREEMENT WHICH GOVERNS ITS USE.
;
;INSTRUCTION DEFINITON FILE

  .TITLE  NBID
  .XPNG           ;DELETE ALL SYMBOLS

;DEFINE STANDARD IO DEVICES

.DUSR MDV=      01      ;MULTIPLY-DIVIDE
.DUSR MAP=    02      ;840 MEMORY MANAGEMENT & PROTECTION UNIT
.DUSR ERCC=   02      ;KLUDGE ENTRY FOR S-20 (YRDOS) -MEM PARITY
.DUSR MAP0= 02      ;MEMORY ALLOCATION AND PROTECTION
.DUSR BMAP=  03      ;ECLIPSE MAP
.DUSR MAP1=       03      ;MEMORY ALLOCATION AND PROTECTION
.DUSR MAP2=  04      ;  "
.DUSR PAR=
       04      ;NOVA 3 PARITY MEMORY UNIT
.DUSR MCAT= 06      ;MULTI-PROCESSOR COMMUNICATIONS ADAPTER TRANSMITTER
.DUSR MCAR=        07      ;MULTI-PROCESSOR COMMUNICATIONS ADAPTER RECEIVER
.DUSR TTI=    10
      ;TELETYPE READER/KEYBOARD
.DUSR TTO=   11      ;TELETYPE PUNCH/PRINTER
.DUSR PTR=     12      ;PAPER TAPE READER
.DUSR PTP=  13      ;PAPER TAPE PUNCH
.DUSR RTC=   14      ;REAL TIME CLOCK
.DUSR PLT=
   15      ;INCREMENTAL PLOTTER
.DUSR CDR=        16      ;CARD READER
.DUSR LPT=        17      ;LINE PRINTER
.DUSR DJP=       20      ;FIRST JAWA CONTROLLER
.DUSR DSK=      20      ;FIRST FIXED HEAD DISK CONTROLLER
.DUSR ADCV=  21      ;A/D CONVERTER
.DUSR MTA=      22      ;FIRST MAG TAPE CONTROLLER
.DUSR DACV= 23      ;D/A CONVERTER
.DUSR DCM=      24      ;DATA COMMUNICATIONS MULTIPLEXOR
.DUSR DAP=    24
      ;PRIMARY CENTAUR/ARGUS CONTROLLER
.DUSR DMP=   25      ;FIRST PACMAN
.DUSR DSP=       26      ;PAGING DISK CONTROLLER
.DUSR DEP=     26      ;THIN-AIR ECHO/CACTUS/QUAD
.DUSR DZP=  27      ;ZEBRA DISK CONTROLLER
.DUSR DHP=      27      ;R2D2 PHOENIX CONTROLLER
.DUSR QTY=    30      ;QUAD MULTIPLEXOR
.DUSR IBM1=  31      ;IBM 360/370 INTERFACE
.DUSR IBM2=     32
.DUSR DKP=  33      ;FIRST MOVING HEAD DISK CONTROLLER
.DUSR CAS=  34      ;FIRST CASSETTE CONTROLLER
.DUSR ALM=  34      ;ASYNCHRONOUS LINE MULTIPLEXOR (ALM-8 AND ALM-16)
```

```
.DUSR ASLM=  34      ;ASLM (4336)
.DUSR MX1=        34      ;1024 LINE ASYNCHRONOUS MULTIPLEXOR
.DUSR MX2= 35
.DUSR IPB=  36      ;INTER-PROCESSOR BUS
.DUSR IVT=        37      ;NON-PROGRAMMABLE INTERVAL TIMER
.DUSR DPI=    40      ;DUAL PROCESSOR INPUT
.DUSR DPO=       41      ;DUAL PROCESSOR OUTPUT
.DUSR DIO=      42      ;DIGITAL I/O INTERFACE
.DUSR MXM=      44      ;1024 LINE MUX MODEM CONTROLLER
.DUSR ALM1=    44      ;SECOND ASYNC. LINE MULTIPLEXOR
.DUSR MCAT1=  46      ;SECOND MULTI-PROC COMMO ADAPTER XMITTER
.DUSR MCAR1=  47      ;SECOND MULTI-PROC COMMO ADAPTER RECEIVER
.DUSR TTI1=  50      ;SECOND TTY
.DUSR TTO1=        51
.DUSR PTR1= 52      ;SECOND PAPER TAPE READER
.DUSR PTP1=  53      ;SECOND PAPER TAPE PUNCH
.DUSR RTC1=   54      ;SECOND REAL TIME CLOCK
.DUSR PLT1=    55      ;SECOND PLOTTER
.DUSR CDR1=    56      ;SECOND CARD READER
.DUSR LPT1=        57      ;SECOND LINE PRINTER
.DUSR DSK1=       60      ;SECOND FIXED HEAD DISK CONTROLLER
.DUSR DJP1= 60      ;SECOND JAWA CONTROLLER
.DUSR MTA1=    62      ;SECOND MAG TAPE CONTROLLER
.DUSR DAP1=        64      ;SECONDARY CENTAUR/ARGUS CONTROLLER
.DUSR DMP1=        65      ;SECOND PACMAN CONTROLLER
.DUSR DSP1=  66      ;SECOND PAGING DISK CONTROLLER
.DUSR DEP1=     66      ;SECOND THIN-AIR CACTUS/ECHO/QUAD CONTROLLER
.DUSR DZP1=       67      ;SECOND ZEBRA DISK CONTROLLER
.DUSR DHP1=      67      ;SECOND R2D2 PHOENIX CONTROLLER (THIN-AIR)
.DUSR QTY1= 70      ;SECOND QUAD MULTIPLEXOR
.DUSR DKP1=   73      ;SECOND MOVING HEAD DISK CONTROLLER
.DUSR ASLM1=
       74      ;SECONDARY ASLM (4336)
.DUSR CAS1=     74      ;SECOND CASSETTE CONTROLLER
.DUSR FPU1=        74      ;SINGLE-PRECISION FLOATING POINT
.DUSR FPU2=   75      ;DOUBLE-PRECISION FLOATING POINT
.DUSR FPU=    76      ;FLOATING-POINT CONTROLLER
.DUSR CPU=  77      ;CENTRAL PROCESSING UNIT
;DEFINE THE STACK INSTRUCTIONS

;MULTIPLY/DIVIDE
.DUSR DIV=  073101
.DUSR MUL=      073301

;DEFINE MEMORY REFERENCE INSTRUCTIONS THAT DON'T REQUIRE AC'S
.DMR JMP=      000000
.DMR JSR=       004000
.DMR ISZ=       010000
.DMR DSZ=       014000

;DEFINE MEMORY REFERENCE INSTRUCTIONS THAT REQUIRE AC'S
.DMRA LDA=   020000
.DMRA STA=      040000

;DEFINE THE ALC INSTRUCTIONS
```

```
.DALC COM=      100000
.DALC NEG=      100400
.DALC MOV=      101000
.DALC INC=      101400
.DALC ADC=      102000
.DALC SUB=      102400
.DALC ADD=      103000
.DALC AND=      103400

;DEFINE THE ALC SKIPS
.DUSR SKP=     1
.DUSR SZC=   2
.DUSR SNC=   3
.DUSR SZR=   4
.DUSR SNR=   5
.DUSR SEZ=   6
.DUSR SBN=   7

;DEFINE THE IO INSTRUCTIONS
.DIO  NIO=  060000
.DIOA DIA=      060400
.DIOA DOA=      061000
.DIOA DIB=      061400
.DIOA DOB=      062000
.DIOA DIC=      062400
.DIOA DOC=      063000

;DEFINE THE IO SKIP INSTRUCTIONS
.DIO SKPBN= 063400
.DIO SKPBZ=     063500
.DIO SKPDN=     063600
.DIO SKPDZ=     063700

;DEFINE SPECIAL INSTRUCTIONS
.DUSR INTEN=  NIOS CPU               ;INTERRUPT ENABLE
.DUSR INTDS= NIOC CPU               ;INTERRUPT DISABLE
.DIAC READS=        DIA 0,CPU              ;READ THE SWITCHES
.DIAC INTA= DIB 0,CPU              ;INTERRUPT ACKNOWLEDGE
.DIAC MSKO=     DOB 0,CPU              ;MASK OUT
.DUSR IORST= DICC 0,CPU             ;IO RESET
.DUSR HALT=  DOC 0,CPU              ;HALT
 .EOT
```

oh, and here's their system call interface

separate EXECUTE FOREGROUND and EXECUTE BACKGROUND calls, and separate READ BLOCK, READ SEQUENTIAL CHARACTERS, READ SEQUENTIAL LINE, READ RANDOM, and GET TTY CHAR calls

I don't know how you start designing an operating system in 1977 and screw up the design this badly

```
;
```

;
; DEFINE AS PERMANENT SYMBOLS ALL MONITOR RELATED SYMBOLS

; DEFINE THE NOVA SYSTEM CALL

```
.DUSR   .SYSTM= JSR     @17

; DEFINE THE USER STACK POINTER LOCATION

.DUSR
USP=    16

; DEFINE THE MONITOR CALLS

; COMMANDS WHICH DO NOT REQUIRE DEVICE ACTION OR CHANNEL NUMBER

.DUSR   .CREAT= 0B7             ; CREATE FILE
.DUSR   .DELET= 1B7             ; DELETE FILE
.DUSR   .RENAM= 2B7             ; RENAME A FILE
.DUSR   .MEM=   3B7             ; RETURN MEMORY LIMITS
.DUSR   .BREAK= 4B7             ; BREAK
.DUSR   .RLSE=  5B7
; RELEASE A DEVICE
.DUSR   .DIR=   6B7             ; CHANGE BASE DIRECTORY
.DUSR   .EXEC=  7B7             ; EXECUTE A PROGRAM OVERLAY
.DUSR   .INIT=  10B7            ; INIT DISK DEVICE

.DUSR   .RTN=   11B7            ; SYSTEM RETURN
.DUSR   .RESET= 12B7            ; I/O RESET
.DUSR   .ERTN=  15B7            ; ERROR RETURN FROM PROGRAM
.DUSR   .CRAND= 16B7            ; CREATE RANDOM
.DUSR   .GCHAR= 17B7            ; GET TTY CHAR
.DUSR   .PCHAR= 20B7            ; TTY PUT CHAR
.DUSR   .DELAY= 21B7            ; WAIT N CYCLES
.DUSR   .MEMI=  22B7
; ALLOCATE MEMORY INCREMENT
.DUSR   .CCON=  41B7            ; CREATE CONTIGUOUS
.DUSR   .EXFG=  43B7            ; EXECUTE FOREGROUND
.DIO    .IOCS=  44B7            ; IOCS SYSTEM CALL
.DUSR   .IOCO=  45B7            ; IOCS OPEN
.DUSR   .EXBG=  55B7            ; EXEC IN BG
.DUSR   .IOCP=  57B7            ; IOCS PRE-OPEN
;       .DUSR   .XXXX=  60B7            ; RESERVED
.DUSR   .MOUNT= 64B7            ; MOUNT FLOPPY
;     COMMANDS WHICH REQUIRE CHANNEL NUMBER

.DIO    .ROPEN= 23B7    ; OPEN FOR READING
.DIO    .MTOPD= 52B7    ; OPEN MAG TAPE FOR DIRECT I/O
.DIO    .OVOPN= 24B7    ; OPEN OVERLAYS
.DIO    .CHATR= 26B7    ; CHANGE THE FILE ATTRIBUTES
```

```
.DIO    .GTATR= 27B7    ; GET THE FILE/DEVICE ATTRIBUTES
.DIO    .RDB=   13B7    ; READ BLOCK
.DIO    .WRB=   14B7    ; WRITE BLOCK
.DIO    .APPEND=25B7    ; OPEN FILE FOR APPENDING
.DIO    .OPEN=  30B7    ; OPEN FILE
.DIO    .CLOSE= 31B7    ; CLOSE FILE
.DIO    .RDS=   32B7    ; READ SEQUENTIAL CHARACTERS
.DIO    .RDL=   33B7    ; READ SEQUENTIAL LINE
.DIO    .RDR=   34B7    ; READ RANDOM
.DIO    .WRS=   35B7    ; WRITE SEQUENTIAL CHARACTERS
.DIO    .WRL=   36B7    ; WRITE SEQUENTIAL LINE
.DIO
.WRR=   37B7    ; WRITE RANDOM
.DIO    .OVLOD= 40B7    ; LOAD OVERLAY
.DIO    .SCALL= 42B7    ; GENERAL CALL
.DIO    .MTDIO= 46B7    ; MAG TAPE DIRECT I/O
.DIO    .SPOS=  47B7    ; SET FILE POSITION
.DIO    .GPOS=  50B7    ; GET FILE'S CURRENT POSITION
.DIO    .EOPEN= 51B7    ; OPEN FOR EXCLUSIVE USE
.DIO    .TOPEN= 52B7    ; TRANSPARENT OPEN
.DIO    .CHLAT= 53B7    ; CHANGE LINK ACCESS ATTRIBUTES
.DIO    .CHSTS= 54B7    ; GET CHANNEL STATUS
.DIO    .UPDAT= 56B7    ; UPDATE FILE SIZE INFORMATION
.DIO    .EWRB=  61B7    ; EXTENDED MEM WRITE
.DIO    .ERDB=  62B7    ; EXTENDED MEM READ
.DIO    .POPEN= 63B7    ; PHYSICAL I/O OPEN

; THE FOLLOWING CALLS ARE SCALLS

.DUSR   .GHRZ=  .SCALL 0        ; GET CLOCK FREQ
.DUSR   .DUCLK= .SCALL 1        ; DEF USER CLOCK
.DUSR   .RUCLK= .SCALL 2        ; REMAOVE USER CLOCK
.DUSR   .GTOD=  .SCALL 3        ; GET TOD
.DUSR   .STOD=  .SCALL 4        ; SET TOD
.DUSR   .SDAY=  .SCALL 5        ; SET DAY

.DUSR   .GDAY=  .SCALL 6        ; GET DAY
.DUSR   .IDEF=  .SCALL 7        ; DEFINE DEVICE INT
.DUSR   .IRMV=  .SCALL 10       ; REMOVE DEV INT
.DUSR   .SPKL=  .SCALL 11       ; SPOOL KILL

.DUSR   .SPDA=  .SCALL 12       ; SPOOL DISABLE
.DUSR   .SPEA=  .SCALL 13       ; SPOOL ENABLE
.DUSR   .RSTAT= .SCALL 14       ; STATUS OF RESOLUTION ENTRY
.DUSR   .CPART= .SCALL 15       ; CREATE PARTITION
.DUSR   .CDIR=  .SCALL 16       ; CREATE SUBDIRECTORY
.DUSR   .LINK=  .SCALL 17       ; LINK ENTRY
.DUSR   .EQIV=  .SCALL 20       ; CHANGE DIRECTORY SPECIFIER
.DUSR   .GDIRS= .SCALL 21       ; GET DIRECTORY SPECIFIER
.DUSR   .SYSI=  .SCALL 22       ; SOS COMPATIBLE CALL
.DUSR   .WCHAR= .SCALL 23       ; WAIT FOR TTY CHAR
.DUSR   .ICMN=  .SCALL 24       ; INIT COMMON
.DUSR   .WRCMN= .SCALL 25       ; WRITE TO COMMON
.DUSR   .RDCMN= .SCALL 26       ; READ COMMON
.DUSR   .ODIS=  .SCALL 27       ; DISABLE INT (CONTL A,C,F)
```

```
.DUSR   .OEBL=  .SCALL 30       ; ENABLE INT
.DUSR   .DEBL=  .SCALL 31       ; ENABLE MAPPED DEV ACCESS
.DUSR   .DDIS=  .SCALL 32       ; DISABLE MAPPED DEV ACCESS
.DUSR   .RDOPR= .SCALL 33       ; READ OPERATOR
.DUSR   .WROPR= .SCALL 34       ; WRITE OPERATOR
.DUSR   .STMAP= .SCALL 35       ; DCH MAP REQ FOR USER
.DUSR   .GCIN=  .SCALL 36       ; GET CONSOLE INPUT DEV
.DUSR   .GCOUT= .SCALL 37       ; GET CONSOLE OUTPUT DEV
.DUSR   .STAT=  .SCALL 40       ; GET STATUS OF FILE
.DUSR   .ECLR=  .SCALL 41       ; RELEASE A FILE
.DUSR   .TCRET= .SCALL 42       ; TRANSPARENT .CREATE
.DUSR   .TCRND= .SCALL 43       ; TRANSPARENT .CRAND
.DUSR   .TCCON= .SCALL 44       ; TRANSPARENT .CCON
.DUSR   .FGND=  .SCALL 45       ; IS THERE A FOREGROUND
.DUSR   .GMEM=  .SCALL 46       ; GET MEM PARTITIONS
.DUSR   .SMEM=  .SCALL 47
; SET MEM PARTITIONS
.DUSR   .BOOT=  .SCALL 50       ; INVOKE BOOT
.DUSR   .MDIR=  .SCALL 51       ; GET MASTER DIR. SPECIFIER
.DUSR   .GCHN=  .SCALL 52       ; GET A FREE CHANNEL
.DUSR   .ULNK=  .SCALL 53       ; DELETE A LINK ENTRY
.DUSR   .WRPR=  .SCALL 54       ; WRITE PROTECT MEMORY
.DUSR   .WREBL= .SCALL 55       ; WRITE ENABLE MEMORY
.DUSR   .GSYS=  .SCALL 56       ; GET CURRENT OPERATING SYSTEM NAME
.DUSR   .OVRP=  .SCALL 57       ; REPLACE AN OVERLAY
.DUSR   .ABTC=  .SCALL 60       ; ABORT A TCB CALL
.DUSR   .GMCA=  .SCALL 61       ; WHAT MCA AM I
.DUSR   .SECI=  .SCALL 62       ; RESCHEDULE EVERY SEC
.DUSR   .HSTRU= .SCALL 63       ; RUN HISTOGRAM
.DUSR   .HSTST= .SCALL 64       ; STOP HISTOGRAM
.DUSR   .RDSW=  .SCALL 65       ; READ SWITCHES
.DUSR   .VMEM=  .SCALL 66       ; GET VIRTUAL MEMORY
.DUSR   .MAPDF= .SCALL 67       ; VIRTUAL DATA MAP DEF
.DUSR   .TUOFF= .SCALL 70       ; TURN TUNING OFF
.DUSR   .TUON=  .SCALL 71       ; TURN TUNING ON
.DUSR   .INTAD= .SCALL 72       ; DEFINE INT TASK
.DUSR   .IOCI=  .SCALL 73       ; IOCS MAGTAPE INIT
.DUSR   .CONN=  .SCALL 74       ; CREATE CONTIGUOUS NO INIT

.EOT
```

Oh fuck, here's how you open a fucking file. You fill in a User File Table entry?

```
;
; COPYRIGHT (C) DATA GENERAL CORPORATION 1977,1978,1979,1980,1982,1983,
; 1984,1985.
; ALL RIGHTS RESERVED.
; LICENSED MATERIAL-PROPERTY OF DATA GENERAL CORPORATION.
; THIS SOFTWARE IS MADE AVAILABLE SOLELY PURSUANT TO THE TERMS OF A
; DGC LICENSE AGREEMENT WHICH GOVERNS ITS USE.

;==================================
```

```
; RDOS REVISION 07.50 USER PARAMETERS
;==================================

        .TITL   PARU

;
; USER FILE TABLE (UFT) TEMPLATE
;

; USER FILE DEFINITION (UFD) OF UFT

.DUSR UFTFN=0           ;FILE NAME
.DUSR UFTEX=5           ;EXTENSION
.DUSR UFTAT=6           ;FILE ATTRIBUTES
.DUSR UFTLK=7           ;LINK ACCESS ATTRIBUTES
.DUSR UFLAD=7           ;LINK ALTERNATE DIRECTORY
.DUSR UFTBK=10          ;NUMBER OF LAST BLOCK IN FILE
.DUSR UFTBC=11          ;NUMBER OF BYTES IN LAST BLOCK
.DUSR UFTAD=12          ;DEVICE ADDRESS OF FIRST BLOCK (0 UNASSIGNED)
.DUSR UFTAC=13          ;YEAR-DAY LAST ACCESSED
.DUSR UFTYD=14          ;YEAR-DAY CREATED
.DUSR UFLAN=14          ;LINK ALIAS NAME
.DUSR UFTHM=15          ;HOUR-MINUTE CREATED
.DUSR UFTP1=16          ;UFD TEMPORARY
.DUSR UFTP2=17          ;WORDS/BLOCK .STAT.RSTA.CHST
.DUSR UFTUC=20          ;USER COUNT
.DUSR UFTDL=21          ;DCT LINK (RH) HIGH-ORDER DEVICE ADDRESS (LH)

; DEVICE CONTROL BLOCK (DCB) OF UFT

.DUSR UFTDC=22          ;DCT ADDRESS
.DUSR UFTUN=23          ;UNIT NUMBER
.DUSR UFCA1=24          ;CURRENT BLOCK ADDRESS (HIGH ORDER)
.DUSR UFTCA=25          ;CURRENT BLOCK ADDRESS (LOW ORDER)
.DUSR UFTCB=26          ;CURRENT BLOCK NUMBER
.DUSR UFTST=27          ;FILE STATUS
.DUSR UFEA1=30          ;ENTRY'S BLOCK ADDRESS (HIGH ORDER)
.DUSR UFTEA=31          ;ENTRY'S BLOCK ADDRESS (LOW ORDER)
.DUSR UFNA1=32          ;NEXT BLOCK ADDRESS (HIGH ORDER)
.DUSR UFTNA=33          ;NEXT BLOCK ADDRESS (LOW ORDER)
.DUSR UFLA1=34          ;LAST BLOCK ADDRESS (HIGH ORDER)
.DUSR UFTLA=35          ;LAST BLOCK ADDRESS (LOW ORDER)
.DUSR UFTDR=36          ;SYS.DR DCB ADDRESS
.DUSR UFFA1=37          ;FIRST ADDRESS (HIGH ORDER)
.DUSR UFTFA=40          ;FIRST ADDRESS (LOW ORDER)

; DCB EXTENSION

.DUSR UFTBN=41          ;CURRENT FILE BLOCK NUMBER
.DUSR UFTBP=42          ;CURRENT FILE BLOCK BYTE POINTER
.DUSR UFTCH=43          ;DEVICE CHARACTERISTICS
.DUSR UFTCN=44          ;ACTIVE REQ COUNT
                        ;B0 INDICATES Q, 0=DSQ1,1=DSQ2

.DUSR UFTEL=UFTCN-UFTFN+1       ;UFT ENTRY LENGTH
```

```
.DUSR UFDEL=UFTDL-UFTFN+1       ;UFD ENTRY LENGTH

.DUSR UDBAT=UFTAT-UFTDC ;NEGATIVE DISP. TO ATTRIBUTES
.DUSR UDDL=UFTDL-UFTDC  ;NEGATIVE DISP. TO FIRST ADDRESS (HIGH ORDER)
.DUSR UDBAD=UFTAD-UFTDC ;NEGATIVE DISP. TO FIRST ADDRESS (LOW ORDER)
.DUSR UDBBK=UFTBK-UFTDC ;NEGATIVE DISP. TO LAST BLOCK
.DUSR UDBBN=UFTBN-UFTDC ;POSITIVE DISP. TO CURRENT BLOCK

; FILE ATTRIBUTES  (IN UFTAT)

.DUSR ATRP =1B0         ;READ PROTECTED
.DUSR ATCHA=1B1         ;CHANGE ATTRIBUTE PROTECTED
.DUSR ATSAV=1B2         ;SAVED FILE
.DUSR ATNRS=1B7         ;CANNOT BE A RESOLUTION ENTRY
.DUSR ATUS1=1B9         ;USER ATTRIBUTE # 1
.DUSR ATUS2=1B10        ;USER ATTRIBUTE # 2
.DUSR ATPER=1B14        ;PERMANENT FILE
.DUSR ATWP =1B15        ;WRITE PROTECTED

; FILE CHARACTERISTICS  (IN UFTAT)

.DUSR ATLNK=1B3         ;LINK ENTRY
.DUSR ATPAR=1B4         ;PARTITION ENTRY
.DUSR ATDIR=1B5         ;DIRECTORY ENTRY
.DUSR ATRES=1B6         ;LINK RESOLUTION (TEMPORARY)
.DUSR ATCON=1B12        ;CONTIGUOUS FILE
.DUSR ATRAN=1B13        ;RANDOM FILE

.DUSR ATMSK=377B7      ; Mask to get high order disk address from
                    ; left byte of offset UFTDL

;
; DCT PARAMETERS.
;

.DUSR DCTBS=0           ;1B0=1 => DEVICE USES DATA CHANNEL
.DUSR DCTMS=1           ;MASK OF LOWER PRIORITY DEVICES
.DUSR DCTIS=2           ;ADDRESS OF INTERRUPT SERVICE ROUTINE

; DEVICE CHARACTERISTICS  (IN UFTCH)

.DUSR   DC100=  1B15    ; CONSOLE INPUT DEVICE IS D100 OR D200
                        ; TERMINAL (SET BY INIT1)
.DUSR   DCSTB=  1B15    ; SUPPRESS TRAILING BLANKS $CDR ONLY
.DUSR   DCCPO=  1B15    ; DEVICE REQUIRING LEADER/TRAILER
.DUSR   DCSTO=  1B15    ; USER SPECIFIED TIME OUT CONSTANT (MCA)
.DUSR   DCCGN=  1B14    ; GRAPHICAL OUTPUT DEVICE WITHOUT TABBING
                        ; HARDWARE
.DUSR   DCIDI=  1B13    ; INPUT DEVICE REQUIRING OPERATOR INTERVENTION
.DUSR   DCLCD=  1B12    ; INPUT DEVICE IS 6053-TYPE TERMINAL
.DUSR   DCCNF=  1B12    ; OUTPUT DEVICE WITHOUT FORM FEED HARDWARE
.DUSR   DCTO=   1B11    ; TELETYPE OUTPUT DEVICE
.DUSR   DCKEY=  1B10    ; KEYBOARD DEVICE
.DUSR   DCNAF=  1B09    ; OUTPUT DEVICE REQUIRING NULLS AFTER FORM FEEDS
.DUSR   DCRAT=  1B08    ; RUBOUTS AFTER TABS REQUIRED
```

```
.DUSR   DCPCK=  1B07    ; DEVICE REQUIRING PARITY CHECK
.DUSR   DCLAC=  1B06    ; REQUIRES LINE FEEDS AFTER CARRIAGE RTN
.DUSR   DCSPO=  1B05    ; SPOOLABLE DEVICE
.DUSR   DCFWD=  1B04    ; FULL WORD DEVICE (ANYTHING GREATER THAN
.DUSR   MSK37=377
.DUSR   MSK17=177
.DUSR   DCLT8=  1B04    ; LESS THAN 8 BITS / CHARACTER (BYTE DEVICES).
.DUSR   DCFFO=  1B03    ; FORM FEEDS ON OPEN
.DUSR   DCLTU=  1B02    ; CHANGE LOWER CASE ASCII TO UPPER
.DUSR   DCC80=  1B01    ; READ 80 COLUMS
.DUSR   DCDIO=  1B00    ; SUSPEND PROTOCOL ON TRANSMIT (MCA)
.DUSR   DCBDK=  1B00    ; DISK CHARACTERISTIC (SET NON-PARAMETRICALLY)
                        ; SET MEANS ITS 3330
.DUSR   DCSPC=  1B00    ; SPOOL CONTROL
                        ; SET = SPOOLING ENABLED
                        ; RESET = SPOOLING DISABLED
.DUSR   DCBIM=  1B12+1B15  ;Absolute binary I/O key.

;CHARACTERISTICS WORD FOR MY DCT'S

.DUSR   TTOCH=DCCGN+DCCNF+DCTO+DCPCK+DCLAC+DCC80+DCSPO+DCSPC
.DUSR   TTICH=DCKEY+DCLTU

;
; DEVICE CHARACTERISTICS FOR QTY, ULM, AND ALM (PARU.SR)
;
 .DUSR  DCNI=   1B15    ;(MASKING ENABLES) CONSOLE INTERRUPTS
;.DUSR  DCCGN=  1B14    ;(MASKING DISABLES) TAB EXPANSION
 .DUSR  DCLOC=  1B13    ;LOCAL LINE (MASKING MAKES MODEM LINE)

;.DUSR  DCTO=   1B11    ;_ FOR RUBOUT (MASKING GIVES BACKSPACE)
                        ;IGNORE LINEFEED (MASKING CONVERTS
                        ; LF/NL TO CR)
;.DUSR  DCKEY=  1B10    ;(MASKING DISABLES) INPUT ECHOING.
                        ;MASKING ALSO DISABLES LINE EDIT
                        ;(^Z,ESC,DEL,\),
                        ;UNLESS "DCEDT" ALSO MASKED.

;.DUSR  DCNAF=  1B9     ;(MASKING DISABLES) 20 NULLS AFTER FORM FEED
 .DUSR  DCXON=  1B8     ;(MASKING ENABLES) XON/XOFF FOR $TTR
;               1B7     ;SAVE FOR FUTURE USE

;.DUSR  DCLAC=  1B6     ;(MASKING DISABLES) LINE FEED AFTER
                        ;   CARRIAGE RETURN
;.DUSR  DCSPO=  1B5     ;(MUST BE OFF) SPOOLING
 .DUSR  DCCRE=  1B4     ;CARRIAGE RETURN ECHO (MASKING DISABLES)
 .DUSR  DCEDT=  1B0     ;LINE EDIT (ESC,^Z,DEL,\) DISABLED IF
                        ; MASK THIS BIT OR "DCKEY", BUT NOT BOTH.
;
; .WRL TO QTY:64
;
;       AC0= CODE+LINE #
;       AC1= DATA

.DUSR   W64DC=  0B7     ;NEW DEVICE CHARACTERISTIC MASK
```

```
                                ;FOR OPEN CHANNEL, AC1 AS ABOVE.
.DUSR   W64LS=  1B7     ;CHANGE LINE SPEED FOR DG/CS,
                                ; AC1 RIGHT-JUSTIFIED CLOCK SELECT.
.DUSR   W64MS=  2B7     ;CHANGE DG/CS MODEM STATE, AC1=
.DUSR     W64DTR= 1B15          ;  RAISE DATA TERMINAL READY
                                ;       ELSE LOWER
.DUSR     W64RTS= 1B14          ;  RAISE REQUEST TO SEND
                                ;       ELSE LOWER
.DUSR   W64CH=  3B7     ;CHANGE CHARACTERISTICS FOR LINE
                                ;AC1 SAME AS DG/CS HARDWARE SPEC.

;       JAWA Controller Board
;       ---------------------------
;       The following values will be placed in PARU.SR and specify
;       the various fields used by the JAWA controller to determine its
;       current state and type of media being used.

;       Mode of operation (used externally)
;       -----------------

.DUSR   CMODE = 0B7             ; Don't use bits 8-15 for mode
.DUSR   SMODE = 1B7             ; Set mode using bits 8-15

;       Tracks per inch (used externally)
;       ----------------
.DUSR   TPI48   = 1B8           ; 2 step pulses to next track (48)
.DUSR   TPI96   = 0B8           ; 1 step pulse to next track (96)

;       Number of Heads (used externally)
;       ---------------
.DUSR   NHED1   = 0B9           ; 1 head (single sided media)
.DUSR   NHED2   = 1B9           ; 2 heads (double sided media)

;       Sectors per Track (used externally)
;       -----------------
.DUSR   SPT01   = 01B6          ;  1 sector per track
.DUSR   SPT02   = 02B6          ;  2 sectors per track
.DUSR   SPT03   = 03B6          ;  3 sectors per track
.DUSR   SPT04   = 04B6          ;  4 sectors per track
```

Hmm, this error code is interesting:

```
    .DUSR   CTMLI=  321     ; TOO MANY LEVELS OF INDIRECT FILES
```

I guess "indirect files" are symlinks?

Here's some of their subroutine linkage convention:

```

        .TITLE  PARS

;
; LINKAGE & STACK STUFF
```

```
;

.DO ?ANSW

        .MACRO  RSAVE           ; CALL TO SAVE REGISTERS
          STA     3,@CSP
          JSR     @.SAV
%

.DUSR   RTRN=   JSR     @4      ; CALL TO RESTORE REGISTERS
.DUSR   RTLOC=  0               ; RETURN LOCATION (THIS FRAME)
.DUSR   AC0=    1               ; AC0
.DUSR   AC1=    2               ; AC1
.DUSR   AC2=    3               ; AC2
.DUSR   TMP=    4               ; FIRST TEMPORARY
.DUSR   MXTMP=  TMP+7           ; LAST TEMPORARY

.DUSR   VRTN=   MXTMP+1         ; VIRTUAL RETURN (THIS FRAME)
.DUSR   SP=     -1              ; CURRENT STACK POINTER
.DUSR   SLGT=   VRTN-SP+1       ; STACK FRAME LENGTH
.DUSR   OSP=    -SLGT+SP        ; LAST FRAME POINTER
.DUSR   NSP=    SLGT+SP         ; NEXT FRAME POINTER
.DUSR   OTMP=   TMP-SLGT        ; OLD FIRST TMP POINTER
.DUSR   OAC0=   AC0-SLGT        ; OLD AC0
.DUSR   OAC1=   AC1-SLGT        ; OLD AC1

.DUSR   OAC2=   AC2-SLGT        ; OLD AC2
.DUSR   ORTN=   RTLOC-SLGT      ; RETURN LOCATION (PREVIOUS FRAME)
.DUSR   OVRTN=  VRTN-SLGT       ; VIRTUAL RETURN (PREVIOUS FRAME)
.DUSR   NFRAM=  11              ; NUMBER OF SYSTEM STACK FRAMES
.DUSR   NDSF=   16              ; NUMBER FRAMES ON DISK STACK

  .DO ?MSW

.DUSR   SVC=    103510          ; SYSTEM CALL ON NOVA 3
.DUSR   SCL=    127510          ; SYSTEM CALL ON NOVA 3

  .ENDC

.ENDC

.DO ?ABSW

        .MACRO  RSAVE           ; CALL TO SAVE STATE
          SAVE    ^1+1          ; PLUS 1 FOR OVLY RTN
%

.DUSR   RTRN=    RTN            ; CALL TO RESTORE STATE

.DUSR   OAC0=   -4              ; CALLER'S AC0
.DUSR   OAC1=   -3              ; CALLER'S AC1
.DUSR   OAC2=   -2              ; CALLER'S AC2
.DUSR   OSP=    -1              ; CALLER'S CSP
.DUSR   ORTN=   0               ; RETURN LOCATION
.DUSR   OVRTN=  1               ; CALLER'S VIRTUAL RETURN
```

```
.DUSR   TMP=    2               ; CALLEE'S FIRST TEMPORARY
.DUSR   SLGTH=  300             ; SYSTEM STACK LENGTH
.DUSR   ISLGT=  100             ; INTERRUPT STACK LENGTH

.ENDC
```

Here we have some documentation about their shell scripts, which are
called "domacros" and probably are the files whose names end in ".MC":

```
     DO[/H]  domacro  arg1/s1  arg2/s2  . . . .

Global /H    Help; display this message

  domacro    Name of a DO macro file
             This file may contain any commands which are valid in a
               standard CLI macro file
             It may also contain variables like %1%, %2%, etc.

  argn/sn    Local arguments and switches
             %1% in the domacro will evaluate to arg1/s1
             Up to 256 arguments are allowed
```

Backup script domacro:

```
    EQUIV/P BACKUP MT0
    INIT/F BACKUP
    XFER TBOOT.SV BACKUP:0
    DUMP/V BACKUP:1 CLI.SV CLI.OL CLI.ER EBOOT.SV ABOOT.SV BOOTSYS.SV RCLI.SV
    XFER BOOTSYS.SV BACKUP:2
    DUMP/A/V BACKUP:3 BOOTSYS.OL
    XFER DKINIT.SV BACKUP:4
    XFER EBOOT.SV BACKUP:5
    DUMP/V BACKUP:6 SYS.LB SYS5.LB IDEB.RB RDOS.SR CBOOT.SV TBOOT.SV ^
    MCABOOT.SV DKINIT.SV DSKED.SV OVLDR.SV SEDIT.SV OEDIT.SV SYSGEN.SV ^
    NSPEED.SV SPEED.ER MEDIT.RB EDIT.SV EDIT.RB MAC.SV MACXR.SV ASM.SV XREF.SV ^
    RLDR.SV RLDR.OL BATCH.SV LFE.SV VFU.SV ENPAT.SV PATCH.SV FLOAD.SV FDUMP.SV ^
    NBID.SR OSID.SR NSID.SR NEID.SR N4ID.SR NFPID.SR BURST.SV PARU.SR PARS.SR ^
    NCID.SR NSKID.SR FPID.SR LITMACS.SR RFPI.RB MATH.LB N3SAC3.RB ^
    BATCH.OL BATCH.ER DBURST.SV OWNER.SV DDUMP.SV DLOAD.SV DDUMP.OL ^
    DLOAD.OL DDUMP.ER MICRODBOOT.SV ^
    DO.SV INITIALIZE.SV INSTALL.MC MBOOT.SV
    DUMP/V BACKUP:7 URDOS<A B C I O>.LB ALMSPD.<SR RB> BACKUP ^
    085000022.18 RDOS0750.FL 0694000<13,20>.00 0694000<15,19,22>.01 ^
    093400027.00
    XFER ABOOT.SV BACKUP:8
    RELEASE BACKUP
```

Topics

Sparkle wheel display

Kragen Javier Sitaker, 2017-05-10 (6 minutes)

Maybe this should be called a "disco ball display".

If you scatter some sparkle sparsely on an uneven black plate and illuminate it with a point source, each fleck of sparkle reflects a beam off in some quasirandom direction. If you rotate the wheel around its axis, the beams from different flecks will scan in a rotating pattern; if the light source is on the axis of the sparkle wheel, the pattern is a cone, but I don't think that's the case in general.

But I don't care that much, because what I care about is what this looks like from a single, somewhat arbitrary point of view, an eye. It looks like sparkling, which is to say, brief pulses of light coming from apparently random sources scattered around the disk. If you watch long enough without moving your eye, the sparkle pattern repeats. Most of the glitter flecks have scan patterns that miss the eye entirely, but those that do not only illuminate it once.

If you map out the pattern of sparkling for a full rotation, you can turn the light source on and off at chosen times to select which flecks appear. With this approach, you can generate a moving image with a single LED, a glitter disk mounted to spin repeatably, and some sort of apparatus for reliably positioning your eye.

Multiple flecks that are simultaneously visible are a problem. If you illuminate while they are visible, you add noise to the image; if you do not, you have even lower efficiency. So ideally the number of flecks would be small enough that the average number of flecks simultaneously visible is somewhere around 1.0.

In the limit of perfectly parallel light, perfectly flat and infinitesimal flecks, and a perfectly infinitesimal viewing pupil, this would not be a problem, because each fleck would be visible for an infinitesimal period of time, so almost all would be at unique times. You could pulse the light source with a Dirac delta function in order to fit nonzero light into this zero time, so you could still see it.

But of course light sources have divergence, glitter flecks have size, flecks are curved, and pupils have size. Picking some numbers, a laser pointer might have 1.2 milliradians of divergence and 2mm of light source diameter, and might be mounted 2 meters from the spinning plate; glitter flecks might be 200 microns in diameter and not have significant curvature; and your eye might be "mounted" 1 meter from the spinning plate and have a diameter of 4 mm. Given these numbers, it seems like the 4-milliradian pupil is almost certainly going to be the limiting factor (and so you might as well use 4-millimeter or 8-millimeter flecks), but perhaps you could fix it by using a small (less than 2 mm) peephole.

You should be able to get up to 4π milliradians of angular scanning out of the apparatus, depending on the relative angles of the light, your eye, the axis, and the flecks. Unfortunately that still only gives you about 3142 pixels at 4 milliradians, which is a shitty display. Illuminating only part of the plate doesn't help; spinning the plate faster doesn't help; the issue is that you need each pixel to get its own separate timeslot in the rotation, and the timeslots are $.004/(4\pi)$ of a full rotation. Making your pupil smaller will help, and if you can

make different rotations different, that will help too.

If you add more light sources, that will help a lot, because the different light sources can activate different flecks simultaneously with no interference. Alternatively you could move the eye, the axis of rotation of the plate, or the light sources, in a controlled fashion, so that different rotations are different in a repeating pattern. So, for example, if you use an 8x8 array of light sources, a 2mm peephole, and rock the axis of rotation of the plate around at 4× slower than the rotation of the plate, you might be able to get 3142·8·8·2·4 = 1.6 megapixels. Now we're talking!

If you rotate the plate at 5400 rpm (90 Hz) and revolve its axis of rotation at one-fourth that, you get 22½ "frames" per second.

You might think that the contrast ratio between the glitter and the black background will be a problem, but I think that you can enhance that contrast ratio arbitrarily by getting further away — until the beam from the glitter fleck is as wide as your pupil, at which point you stop winning.

Then it's just a matter of mapping out which light source illuminates which spots on the disk at which position in the rotational cycle.

As displays go, this is grievously inefficient. If your disc is 50mm across and only the light falling on a 200-micron-square portion of it is being used, and that only half the time, then 62499/62500 of the light is lost. But with a sufficiently bright light source, that should be okay.

Using this approach backwards, you can recognize the fleck pattern of a particular position with a camera in order to detect the rotational position of the disc: a rotary encoder, potentially with three degrees of freedom.

Topics

Notes on Raph Levien's "Io" Programming Language

Kragen Javier Sitaker, 2007 to 2009 (10 minutes)

(This is distinct and unrelated to Steve Dekorte's "Io" programming language.)

The original paper, which I don't have a copy of, is:

Raphael Levien, 1989, "Io: a new programming notation", SIGPLAN Notices 24(12) December 1989

There is a little material about Io online, including quotes from the paper. From http://hopl.murdoch.edu.au/showlanguage.prx?exp=4671&language%oo=IO:

Coroutines

Coroutines are an important concept of computing science, but few programming notations properly support them. It is surprising how easy they are to implement in Io.

The idea of coroutines is to have two (or more) routines. When one of the routines gets to a point where it can no longer proceed (such as, when it needs more input), it is suspended, and another routine continues until it, in turn, can no longer continue (such as, when it has a value to output). Then, it is suspended and another routine is resumed.

This is used, for example, in creating a stream. A stream carries a sequence of numbers, without consuming storage. Therefore, it can be infinite. Even in the case of a finite stream, though, it has an advantage over a linked list, because computation can begin immediately after the first number is known.

The Io implementation of streams is analogous to linked lists. A stream takes two arguments. If there is no more data in the stream, it performs its first argument. Otherwise, it performs the second argument, with a data value and the continuation of the stream.

Here we define the operator `count-stream`, and bind an infinite counting stream to the variable `s`.

```
count-stream0: ~ x out;
out x ~ null out;
+x1~x;
count-stream0 x out.
count-stream: -..) ret;
ret .-9 null full;
count-stream0 0 full.
count-stream ~ s
```

S has exactly the same structure as a linked list. In fact, `writelist s` will write `0 1 2 3 4 5...` on the screen.

There seem to be some OCR errors here. I think `+x1~x` is supposed to be `+ x 1 ~ x`, and I suspect (from Raphael Finkel's book, see below) that `~` is actually supposed to be `->`. So the definition of count-stream0 should be as follows:

```
count-stream0: -> x out;
        out x -> null out;
        + x 1 -> x;
        count-stream0 x out.
```

In Scheme:

```
(define count-stream0
  (lambda (x out)
    (out x (lambda (null out)
             (%+ x 1 (lambda (x) (count-stream0 x out)))))))
```

with the following definition of %+:

```
(define (%+ a b cont) (cont (+ a b)))
```

I'm more mystified about the `count-stream` definition. From the text, perhaps the definition is as follows:

```
count-stream: -> ret;
        ret -> null full;
        count-stream0 0 full.
```

Because then s gets `-> null full; count-stream0 0 full`, which takes two arguments (as the text explains) and hands the second one off to `count-stream0`, which performs it with a data value and the continuation of the stream.

Raphael Finkel's 1995/1996 book "Advanced Programming Language Design", chapter 2, section 3, contains some more examples.

```
write 5; write 6; terminate
```

which means, in Scheme:

```
(write 5 (lambda () (write 6 (lambda () (terminate)))))
```

Then

```
write-twice: -> number; write number; write number; terminate.
```

which means

```
(define write-twice
  (lambda (number)
    (write number
           (lambda () (write number (lambda () (terminate)))))))
```

Then

```
write-twice: -> number return;
        write number; write number; return.
write-twice 7; write 9; terminate
```

Which means

```
(define write-twice
  (lambda (number return)
    (write number (lambda () (write number
                                    (lambda () (return)))))))
(write-twice 7 (lambda () (write 9 (lambda () (terminate)))))
```

Then

```
+ 2 3 -> number; write number; terminate
```

which means

```
(%+ 2 3 (lambda (number) (write number (lambda () (terminate)))))
```

Then

```
count: -> start end return;
        write start;
        = start end (return);
        + start 1 -> new-start;
        count new-start end return.
count 1 10; terminate
```

which means

```
(define count
  (lambda (start end return)
    (write start
           (lambda ()
             (%= start end return
                 (lambda ()
                   (%+ start 1
                       (lambda (new-start)
                         (count new-start end return)))))))))
```

with the new definition of %=:

```
(define (%= a b consequent alternate)
  (if (= a b) (consequent) (alternate)))
```

This is the CPS expansion of this:

```
(define (count start end)
  (write start)
  (if (not (= start end)) (count (+ start 1) end)))
```

I don't know why there are parentheses in "= start end (return)" in the Io example. Perhaps it's an error introduced by Finkel.

One final example, showing the use of parentheses:

```
make-pair: -> x y return;
        user (-> client; client x y); return.
```

which means

```
(define make-pair
  (lambda (x y return)
    (user (lambda (client) (client x y)) (lambda () (return)))))
```

Here's the definition of writelist mentioned above:

```
writelist: -> list return;
        list return -> first rest;
        write first;
```

```
        writelist rest;
        return.
emptylist: -> null notnull; null.
cons: -> number list econtinuation;
        econtinuation -> null notnull;
        notnull number list.
```

Usefulness

I wouldn't want to program in Io in the raw way described above; it's pretty verbose and confusing. But it's *much* clearer than Scheme for expressing code in explicit CPS, for three simple reasons.

First, a series of nested lambdas is a flat structure rather than a nested structure as in Scheme.

Second, the syntactic overhead of the lambda is a single punctuation character, or possibly three, rather than ten characters including some letters: `(lambda())`.

Third, as a result, in the usual case, the distance between the names of arguments and the place they come from (that is, the procedure that will eventually invoke the lambda that the arguments belong to) is much less, and they appear as a unit rather than as things far apart. `+ x 1 -> x;` is quite clear. (Unfortunately, this closeness of association is misleading sometimes; consider `out x -> null out;` in the definition of `count-stream0`, where the `-> null out; ...` part of the routine is suspended for some arbitrary period of time while the rest of the program runs, and may in fact never resume.)

More Syntactic Sugar

If you actually wanted to write programs in the language, you could benefit from changing it to have a little bit more syntactic sugar.

Nested expressions

For example, you could define

```
count [+ start 1] end return
```

as an abbreviation for

```
+ start 1 -> new-start;
count new-start end return
```

and for procedures that have only a single exit point, you could imagine writing

```
{-> number; write number; write number}
```

as an abbreviation for

```
-> number return; write number; write number return
```

In cases where a "statement" contains more than a single set of square brackets, the order of evaluation could be undefined, so that e.g.

```
string-scan src [+ srcidx 1] [- len 1] c
```

could rewrite either to

```
+ srcidx 1 -> v1;
- len 1 -> v2;
string-scan src v1 v2 c
```

or to

```
- len 1 -> v1;
+ srcidx 1 -> v2;
string-scan src v2 v1 c
```

Or the order of evaluation could be defined; who cares? However, it's important for our sanity that this:

```
string-scan src [+ srcidx 1]; foobar [- len 1]
```

rewrite to this:

```
+ srcidx 1 -> v1;
string-scan src v1;
- len 1 -> v2;
foobar v2
```

and not this:

```
+ srcidx 1 -> v1;
- len 1 -> v2;
string-scan src v1;
foobar v2
```

Note that the above transformation is just the CPS transformation in Scheme for normal nested application expressions. It's just a thousand times more readable than usual because of the Io lambda notation.

One-argument lambda sugar

It might also be helpful to be able to write one-argument lambdas more concisely, with an automatic name for "the last result". In Python's REPL and in Arc, this variable is called "_". With this, for example, you could write each of the following:

```
count-stream: ; _ -> null full; count-stream0 0 full.
```

```
+ 2 3; write _; terminate
```

```
make-pair: -> x y ret; user (; _ x y) ret.
```

Mostly this is duplicative with the []-nesting idea, though. I'm not sure which is better in the cases where both are applicable. Consider this example:

```
def render(text):
    body = str(markdown.Markdown(text))
```

```
    soup = BeautifulSoup.BeautifulSoup(body)

    headers = soup('h1')
```

In Io, that looks like this:

```
render: -> text;
    markdown.Markdown text -> foo;
    str foo -> body;
    BeautifulSoup.BeautifulSoup body -> soup;

    soup "h1" -> headers; ...
```

With implicit single arguments:

```
render: ;
    markdown.Markdown _;
    str _;
    BeautifulSoup.BeautifulSoup _;

    _ "h1" -> headers; ...
```

With nesting:

```
render: -> text;
    [BeautifulSoup.BeautifulSoup [str [markdown.Markdown text]]] "h1"
        -> headers; ...
```

The nested expressions are more compact, but in this case, I think the implicit arguments are clearer.

Conditionals

It would be nice if there were a way to conveniently rejoin streams of control after a conditional. For example, it would be nice to be able to write

```
if (= x y) (write "x y equal") (write "x y not equal");
if (= x z) (write "x z equal") (write "x z not equal");
if (= y z) (write "y z equal") (write "y z not equal");
whatever
```

If the language had automatic currying, you could define this `if` quite easily:

```
if: -> cond result alt cont; cond (; result cont) (; alt cont).
```

You can use the above `if` definition without automatic currying if you write out the arguments explicitly:

```
if (-> a b; = x y a b) (-> c; write "x y equal" c)
                       (-> c; write "x y not equal" c)
```

You could, however, imagine syntactic sugar for this as well. For example, this expression could expand into the above call to "if":

```
= x y ? write "x y equal" : write "x y not equal"
```

As with the nested expressions, note that this is just the CPS transformation for `if`.

Topics

Vibratory powder delivery

Kragen Javier Sitaker, 2017-02-25 (2 minutes)

One of the problems I'm having with powder-bed processes in the ceramic studio is that fine powders, whether quartz, feldspar, or glass, are clumpy. I'm thinking I can vibrate a tool to break them up. I've ascertained that some commercial ultrasonic vibrating sieves use about 200 watts of power per kilogram of vibrating mass; I think I can get by with 10 grams of apparatus and powder (really more like 100mg of powder) and thus 2 watts, and I'm wondering if piezoelectric speakers are a plausible way to deliver that vibration.

Digi-Key's most popular "buzzer element/piezo bender" is the CUI CEB-20D64, a US$1.38 6.5kHz piezo buzzer element that takes 30 volts peak-to-peak at an impedance of 350 ohms, which I guess works out to 2.6 watts.

So the answer is yes, cheap piezoelectric speakers have enough power to do the job. How about reaching ultrasonic frequencies?

The Murata MA40S4S is a US$6.80 40kHz ultrasonic transmitter; it takes a 20-volt peak-to-peak square wave input and delivers 120 dB SPL output at 30cm with a 10Vrms sine wave. It claims a 2550 pF capacitance at 1kHz. Unfortunately, this is not enough information to estimate its power output, since the SPL (20 Pa) depends on its directionality, which is not specified.

The next most popular ultrasonic transmitter is the US$4.95 PUI UT-1240K-TT-R, 40kHz with a 70° -6dB beam angle, running on 30V peak-to-peak. It claims 2100pF and 115dB.

(I could totally calculate how much power gets stored in and paid back from that capacitance at a given frequency and voltage, but what I'm interested in is the fraction that doesn't get paid back because it's emitted as sound.)

In the "Alarms, Buzzers, and Sirens" Digi-Key category, I find the much more popular TDK PS1240P02BT, a 33¢ 3V 4kHz single-tone "piezoelectronic buzzer without oscillator circuit" that TDK markets for, among other things, "speech synthesis output"; they plot its frequency response curve out to 10kHz in the 70–80 dBA range at 10cm and 3V, but it takes up to 30V input.

These are probably in the same power range as the first buzzer, though, just less directional.

Topics

Bit difference array

Kragen Javier Sitaker, 2018-10-28 (10 minutes)

So I was thinking about the sparse difference-array representation METAFONT uses for rows of pixels: +1 at the left edge of an ink area, -1 at the right edge. Once all the ink is thus applied, potentially including overlapping strokes, the prefix sum is calculated, thus spreading the ink, and the pixels with positive, or nonzero, amounts of ink are then colored. If you instead color pixels with odd amounts of ink, you get even-odd filling rather than nonzero filling.

In METAFONT, IIRC, the difference array is sparse: it contains (X-coordinate, increment) pairs, which could be specialized to (X, ±1) if we allow duplicate Xes. This allows an arbitrarily high X-resolution, for a fixed number of edges anyway, at only logarithmic cost. For a number of edges E much larger than the number of pixel positions N, a dense array is more efficient, since it can coalesce multiple edges that happen to cross the scan line within the same pixel; it need only contain N items instead of the larger number E.

Dense arrays also have the advantage of permitting more regular processing which fits better into CPUs. In what follows, I'm only considering dense arrays.

Gray code on bit-packed scanlines

It occurred to me that, in the even-odd case, you could use single bits for the pixels, and the prefix-sum operation and its inverse are simply the operations of converting between binary and reflected-binary Gray code. Converting from binary to RBGC, the backward difference operation, is simple and efficient (`x ^ x >> 1`), but the inverse operation (RBGC to binary, prefix-sum of XOR) is somewhat less efficient on a normal CPU. Chapter 13 of *Hacker's Delight* is about Gray code, and it gives this algorithm for 32 bits:

```
B = G ^ G >> 1;
B ^= B >> 2;
B ^= B >> 4;
B ^= B >> 8;
B ^= B >> 16;
```

An additional shifted-XOR is needed for 64 bits. GCC for amd64 renders this as follows, requiring three instructions per line, 18 instructions in all, plus in this case a `retq` of overhead:

```
  400620:   48 89 f8              mov    %rdi,%rax
  400623:   48 d1 f8              sar    %rax
  400626:   48 31 f8              xor    %rdi,%rax
  400629:   48 89 c2              mov    %rax,%rdx
  40062c:   48 c1 fa 02           sar    $0x2,%rdx
  400630:   48 31 d0              xor    %rdx,%rax
  400633:   48 89 c2              mov    %rax,%rdx
  400636:   48 c1 fa 04           sar    $0x4,%rdx
  40063a:   48 31 d0              xor    %rdx,%rax
  40063d:   48 89 c2              mov    %rax,%rdx
```

```
400640:   48 c1 fa 08           sar    $0x8,%rdx
400644:   48 31 d0              xor    %rdx,%rax
400647:   48 89 c2              mov    %rax,%rdx
40064a:   48 c1 fa 10           sar    $0x10,%rdx
40064e:   48 31 d0              xor    %rdx,%rax
400651:   48 89 c2              mov    %rax,%rdx
400654:   48 c1 fa 20           sar    $0x20,%rdx
400658:   48 31 d0              xor    %rdx,%rax
40065b:   c3                    retq
```

This works out to 0.28 instructions per pixel, which is still less than 1, and furthermore a third of those are `mov`s that might disappear in the micro-op representation inside the CPU, but it's not the order-of-magnitude kind of speedup we might hope for.

(On ARM Thumb-2 (not shown) it's 28 instructions.)

If you're running this operation on an entire scanline to compute its XOR prefix sum, you need to bring in the output parity bit from the previous word as well.

Bitsliced scanlines

Another approach to this problem is to bitslice it. Suppose that we have an array of 1024 32-bit words W, each word W[i] of which has bit N set (i.e. `0 != (W[i] & (1 << N))`) iff there is a left-right boundary at (i, N). Then we can calculate the XOR prefix sum, i.e. convert each row of pixels from RBGC to binary, simply and efficiently in parallel across the scan lines:

```
for (int i = 1; i < 1024; i++) W[i] ^= W[i-1];
```

This gets compiled to the following for amd64:

```
 0:   48 8d 47 04             lea    0x4(%rdi),%rax
 4:   48 8d 8f 00 10 00 00    lea    0x1000(%rdi),%rcx
 b:   8b 50 fc                mov    -0x4(%rax),%edx
 e:   31 10                   xor    %edx,(%rax)
10:   48 83 c0 04             add    $0x4,%rax
14:   48 39 c8                cmp    %rcx,%rax
17:   75 f2                   jne    b <xor_prefix_sum_bitslice+0xb>
19:   f3 c3                   repz retq
```

This inner loop is 5 instructions (two of which contain memory references) processing 32 pixels. It takes about 2.2 µs on my 1.6 GHz Pentium N3700, or 67 ps per pixel, 9.3 pixels per clock cycle. But you can unroll it, say by a factor of 4:

```
W[1] ^= W[0];
W[2] ^= W[1];
W[3] ^= W[2];
for (int i = 4; i < 1024; i += 4) {
  W[i] ^= W[i-1];
  W[i+1] ^= W[i];
  W[i+2] ^= W[i+1];
  W[i+3] ^= W[i+2];
}
```

```
1b:   8b 47 04              mov    0x4(%rdi),%eax
1e:   33 07                 xor    (%rdi),%eax
20:   89 47 04              mov    %eax,0x4(%rdi)
23:   33 47 08              xor    0x8(%rdi),%eax
26:   89 47 08              mov    %eax,0x8(%rdi)
29:   31 47 0c              xor    %eax,0xc(%rdi)
2c:   48 8d 47 10           lea    0x10(%rdi),%rax
30:   48 8d 8f 00 10 00 00  lea    0x1000(%rdi),%rcx
37:   8b 10                 mov    (%rax),%edx          ; loop starts here
39:   33 50 fc              xor    -0x4(%rax),%edx
3c:   89 10                 mov    %edx,(%rax)
3e:   33 50 04              xor    0x4(%rax),%edx
41:   89 50 04              mov    %edx,0x4(%rax)
44:   33 50 08              xor    0x8(%rax),%edx
47:   89 50 08              mov    %edx,0x8(%rax)
4a:   31 50 0c              xor    %edx,0xc(%rax)
4d:   48 83 c0 10           add    $0x10,%rax
51:   48 39 c8              cmp    %rcx,%rax
54:   75 e1                 jne    37 <xor_prefix_sum_bitslice_unrolled+0x1c>
56:   f3 c3                 repz retq
```

This processes 128 pixels in 11 instructions, although 8 of those instructions contain memory references, so it's really more like 19 instructions. (Or more, if you count the indexing operation separately.) It takes about 1.0 µs on my 1.6 GHz Pentium N3700, or 31 ps per pixel, 20 pixels per clock cycle. Filling a megapixel thus would require 31 µs, assuming you don't incur cache misses.

These numbers give us respectively 0.156 (5/32), 0.086 (11/128), and 0.148 (19/128) instructions per pixel. That's more like it!

(That also suggests that the N3700 is managing about 1.7 instructions per clock cycle.)

You might hope that other architectures would be more efficient, but they seem to be about the same; here's an ARM Thumb-2 compilation of the same unrolled loop, which is one byte shorter and has 16 instructions instead of 11 in the inner loop; it avoids the redundant memory loads of the amd64 version, but has to use explicit loads and stores.

```
1c:   6842        ldr r2, [r0, #4]
1e:   6803        ldr r3, [r0, #0]
20:   405a        eors    r2, r3
22:   6042        str r2, [r0, #4]
24:   6883        ldr r3, [r0, #8]
26:   4053        eors    r3, r2
28:   6083        str r3, [r0, #8]
2a:   68c2        ldr r2, [r0, #12]
2c:   4053        eors    r3, r2
2e:   60c3        str r3, [r0, #12]
30:   4603        mov r3, r0
32:   f500 607f   add.w   r0, r0, #4080   ; 0xff0
36:   691a        ldr r2, [r3, #16]   ; loop starts here
38:   68d9        ldr r1, [r3, #12]
3a:   4051        eors    r1, r2
3c:   6119        str r1, [r3, #16]
3e:   695a        ldr r2, [r3, #20]
```

```
 40:   4051        eors    r1, r2
 42:   6159        str r1, [r3, #20]
 44:   699a        ldr r2, [r3, #24]
 46:   404a        eors    r2, r1
 48:   619a        str r2, [r3, #24]
 4a:   69d9        ldr r1, [r3, #28]
 4c:   404a        eors    r2, r1
 4e:   61da        str r2, [r3, #28]
 50:   3310        adds    r3, #16
 52:   4283        cmp r3, r0
 54:   d1ef        bne.n   36 <xor_prefix_sum_bitslice_unrolled+0x1a>
 56:   4770        bx  lr
```

On a 64-bit machine, we could halve all these numbers, at the expense of doubling the working set (to 8 KiB) and the data traffic to the cache, by using 64-bit words.

Unfortunately, then we need to transpose the bit matrix to get the pixels in the usual scanline order.

Oversampling

These approaches might be sufficiently efficient to permit you to do antialiasing by the brute-force approach of oversampling. For example, if you do brute-force oversampling in both X and Y, you have four subpixels per screen pixel, and thus five gray levels for each screen pixel (0, 1, 2, 3, 4).

Even-odd-filled polylines

If you have a closed polygon or polyline, in theory you can just draw the edges into your scanline buffer with XOR (`W[x] ^= 1 << y` or `W[y*w + x/N] ^= 1 << (x%N)` or whatever) and get it filled this way. But you have to be careful about vertices: in the scanline where a vertex falls, you must be careful to XOR it into that scanline only once, not once for the line that starts there and again for the line that ends there. That bug leads to an inverse-video bleed-across on that line.

Similarly, you need to be careful with local minima and maxima of the Y-coordinate along the border, particularly if there's a vertex present.

Gradient and texture fills

The above algorithms handle solid-color fills pretty well, but solid colors nearly don't exist in nature. Even objects whose color is perfectly homogeneous are usually lit by inhomogeneous lighting, either because they are curved, because of fuzzy shadows, or because of varying distances to nearby light sources, including indirect light diffused from other objects. So non-computer-screen objects almost always have gradients.

Also, non-computer-screen objects usually have texture. Wood has grain, cement or dirt has white-noise texture, hair has striation, cloth has weave, and so on. At large grain, we can handle this by drawing thousands of objects, but the grains of a brick or the hair of a crowd of people, for example, is impractical to handle this way.

Further thoughts on this in Cheap textures (p. 736).

Topics

Further notes on algebras for dark silicon

Kragen Javier Sitaker, 2016-09-17 (updated 2017-04-18) (23 minutes)

Typical computers support three algebras directly: $GF(2^n)$ (integer math), $GF(2)^n$ (bitwise operations), and floating-point math, which is its own weird thing. Combining operations from more than one of these algebras often gives surprisingly fast iteration-free code to solve problems that would seem to require iteration, like clearing the rightmost bit set in a word: `x & x - 1`. However, other operations that aren't inherently computationally expensive are still expensive with these operations — the classic example is the fixed bit permutation at the input and output of DES, which is frequently speculated to be nothing more than an attempt to artificially slow down software implementations.

With the advent of the dark-silicon age, where we can fabricate far more transistors on a chip than we can afford to remove the heat from at reasonable clock speeds, it seems that adding new instructions that are less mainstream in their applications should be worthwhile. So far, this has mostly taken the form of instructions to implement particular cryptographic algorithms (like AES-NI) and multimedia-oriented and 3-D-oriented instructions like those in 3DNow!, MMX, various versions of SSE, AltiVec, and modern GPUs. These have the advantage of adding various kinds of vector algebras.

(SSE 4.2 also added a CRC32 instruction.)

Unfortunately, the available instructions in the multimedia and 3-D sets are generally fairly incoherent, making them awkward to program with.

We could add more random instructions that compute functions like "$\lceil\sqrt{(x^3 - 50x + y)}\rceil$ over the integers", but this would be kind of a waste of time — they can already be evaluated on mainstream hardware without unreasonable overhead on top of their inherent computational cost; and they don't compose in a useful way — which is to say, they don't form an interesting algebra. Or we could add an instruction that computes the derivative of a closed-form symbolic expression, but because the set of closed-form symbolic expressions is infinite, you can't fit a symbolic expression into a fixed-size register — only *finite* algebras are of interest.

So, what other finite algebras might it be advantageous to add hardware support for because mainstream hardware currently imposes punishing overhead on them?

Things we already pretty much have

We might consider lattices of finite set containment, but in fact the bitwise operations already give us that. Many other interesting lattices are already provided by the vector-math stuff we added for 3-D rendering.

Similarly, both signed and unsigned integer math are already reasonably well supported by the $GF(2^n)$ operations thath have been in CPUs for decades.

Bitwise rotation instructions have been totally mainstream for decades (since at least the 4004); I was always mystified by these, but it turns out they're necessary for multi-precision bit-shift operations, and they're also handy for individually testing bits that you're shifting out of numbers.

We also got saturating arithmetic as part of the whole multimedia and vector instruction shit sandwich.

Linear feedback shift registers seem like they would be a pain, since in theory the next bit to shift into the register is the XOR of the tapped bits from the register. But in fact you can also implement an LFSR by shifting a bit *out* of the register and using it to conditionally XOR *all* of the tapped bits *into* the register.

Shuffling, permutation, symmetric groups, and substitution ciphers

Much of this section concerns, in one or another sense, the symmetric group on some small number of elements, typically somewhere between 16 and 256 elements. This may seem like a strangely arbitrary choice, but in fact the symmetric group on N elements is in some sense the canonical group on N elements — every other group on N elements is a subgroup of it! So, in a sense, operations that can handle arbitrary symmetric groups can handle arbitrary groups.

Computers are, at their core, code machines; they operate on arbitrary information enciphered into a representation they can deal with. Classical (pre-computer) ciphers were broadly classified into substitution and transposition ciphers. In substitution ciphers, the pieces of the plaintext tell where to put pieces of the key schedule to form the ciphertext; in transposition ciphers, the pieces of the key schedule where to put pieces of the plaintext to form the ciphertext. These are in a sense the same operation, and it is somewhat surprising that they are so poorly supported in modern hardware. Perhaps this is accounted for by the fact that they are very well supported indeed by RAM (and the 8086 has an XLAT instruction that uses RAM for this in 11 clock cycles); it's just that RAM is falling progressively further and further behind the capabilities of CPUs.

Knuth's MMIX has instructions called MOR and MXOR, which "regard the 8 bytes of a register as a 8×8-Matrix and compute the result as a matrix multiplication", or "set eacy byte of $X by looking at the corresponding byte of $Z and using its bits to select bytes of $Y" (Knuth's Fascicle 1). In effect, each bit specifies whether or not to include one of the 8 input bytes in the computation of one of the 8 output bytes. If only one bit is set in each byte of the second argument, the output is just a rearrangement or shuffle of the input bytes; but, by setting more bits, we can activate the OR or XOR mentioned in the name.

Being generalized matrix multiplications using associative operators, these operations are of course associative. I know of no manufactured CPU that implements them, but something similar is commonplace in FPGA routing — the bits in $Z amount to a crossbar switch.

As Mytkowicz, Musuvathi, and Schulte showed in their 2014 paper , the shuffle instructions such as PSHUFB in SSE3 and SSE4.2 can be

used to accelerate and parallelize finite-state-machine execution. The PSHUFB instruction (`_mm_shuffle_epi8`) and its variant VPSHUFB is something like a cut-down MOR, using indices encoded as binary integers instead of bit vectors, thus offering no possibility of combining multiple input bytes into one output byte, but it can operate on 128-bit and 256-bit registers as well as 64-bit ones. Still, because it applies a permutation to the input (which may be a permutation) it's associative too, which is how Mytkowicz et al. used it to accelerate finite-state machines.

(More recent versions of SSE include PSHUFW and PSHUFD versions, which shuffle wider chunks around.)

If what you wanted was specifically to compose permutations like that, you'd be faced with a couple of problems:

• Even a 256-bit VPSHUFB wastes 3 bits out of every index, in the sense that the maximum valid byte index is 31, so only the low 5 bits of the index are used. If you were to cut the indices to 6 bits, you could fit 42 of them into a 256-bit register; if you could somehow use arithmetic coding, you could get the number a little higher still.
• There are times when you would like to compose permutations on sets of more than 32 or 42 elements, and it isn't immediately obvious how to efficiently decompose such permutations on larger sets into permutations on smaller sets, although related questions have been explored in some depth in the context of building circuit-switched telephone systems out of crossbar switches.

Another issue is that there are a number of kinds of bit permutation that byte-shuffling (whether 8-bit or SIXBIT) doesn't help you with. Massalin famously called for a perfect-bitwise-shuffle operation in her dissertation, which is still not available in current hardware.

So here are a couple of new shuffling designs that might be worth exploring:

• A 64-bit value v is considered to consist of 16 4-bit nybbles v0 through v[15]. xbar(a, b)[i] = a[b[i]], thus providing the same permutation-composition power as a 128-bit PSHUFB in half the bits. Additional nybble-perfect-shuffle operations "hi" and "lo" provide a way to combine nybbles from different registers without requiring fundamental operations of greater than binary arity: lo(a, b)[i] is a[(i-1)/2] when i is odd but b[i/2] when i is even, while hi(a, b)[i] is a[8 + (i-1)/2] when i is odd but b[8 + i/2] when i is even.

These nybble-perfect-shuffle operations make it straightforward to use this 16-nybble-xbar instruction as a 16-byte xbar, a 16-wyde xbar, etc.; and they make it relatively straightforward to leverage them into computing permutations of larger groups of nybbles.

• The N-bit value mux(x, y, z) is computed from three N-bit values x, y, and z as follows. The bit mux(x, y, z)[i] is x[i] if z[i] is 1, otherwise y[i]. The N-bit value shuf(a, b) is computed from an N-bit value a and an N/2-bit value b as mux(a[0:N/2], a[N/2:N], b) || mux(a[0:N/2], a[N/2:N], ¬b) where || is string concatenation and ¬ is bitwise negation. The N-bit value 2shuf(a, b) is computed from two N-bit values a and b as shuf(shuf(a, b[0:N/2]), b[N/2:N]). Arguments from sorting networks suggest that it's possible to perform any permutation of N bits by providing the appropriate `b` values to (lg

N)(lg N + 1)/4 2shuf operations; when N is 128, for example, 14 2shuf operations suffice, and when N is 1024, 28 2shuf operations suffice. Arguments from `mux(x, y, z) = x & z | y & ~z` suggest that you can do this on existing hardware and don't need new hardware. XXX no this is totally wrong because the bits aren't being perfect-shuffled! I want a perfect shuffle! rewrite

• MNAND or MNOR, which are analogous to the MOR and MXOR instructions, but permit specifications of a large class of small parallel computations as a fixed sequence of $Z values.

Lerp

As Darius Bacon pointed out, you can think of linear interpolation as being a sort of generalization of C's ternary operator.

GPUs automatically provide linear interpolation between texels. Iterating linear interpolation provides Bézier curves. Alpha-blending linearly interpolates between pixel values. The usual way to do linear interpolation is with two separate multiplications and a subtraction:

```
x := 1.0 - y;
z := y · a;
z += x · b;
```

However, this uses slightly more than twice as many bit operations as necessary; you can adapt the standard binary long multiplication algorithm to lerp instead. The standard algorithm to add the product y · b to z, consuming both, is:

• Go to step 4.
• If y is odd (that is to say, if `y & 1` is nonzero), z += b.
• y ← ⌊y/2⌋; concurrently, b ← b · 2.
• If y is nonzero, go to step 2.

While implementations of multiplication on transistor-starved hardware of the distant past often used this algorithm literally, taking more time to multiply by a larger multiplicand or one with more 1 bits — the 8086 would take anywhere from 128 to 154 clock cycles for a 16×16-bit multiply, while the 80486 would take 13 to 26 — modern hardware instead unrolls the loop into a cascade of carry-save adders. To multiply two 16-bit integers, for example, it has 256 carry-save adders. This means the loop gets rewritten as follows for a 16-bit multiply:

Repeat 16 times:

• If y is odd, z += b.
• y ← ⌊y/2⌋; b ← b · 2.

The modification to make this algorithm interpolate linearly between a and b is very simple:

Repeat 16 times:

• z += (b if y is odd else a).
• y ← ⌊y/2⌋; b ← b · 2; a ← a · 2.

This is easiest to understand if we think of a, b, and y as initially representing 16-bit fractions between 0 and 1, with 1 being represented by 16 1 bits, and the 32-bit result as likewise representing a fraction between 0 and 1, but with 1 being represented by 15 1 bits

followed by 6 0 bits followed by as 1, which is admittedly a somewhat awkward representation. You can restore it to a proper 16-bit form, if lerping is what you want, by taking the leftmost 16 bits and adding their most significant bit to their least significant bit.

So, for example, the 13-instruction multiplication code example on p.54 of the 8080 Programmer's Manual would become 15 instructions, and would execute 11 or 12 instructions per iteration instead of 10 or 11.

(Most current hardware multipliers reportedly use Booth's multiplication algorithm, which is not quite as easy to adapt.)

Of course, you can replace "16" and "32" in the above with "N" and "2N".

Decimal arithmetic

In the 1950s and into the 1960s, business computers invariably used (binary-coded) decimal arithmetic, thus avoiding the need to laboriously convert numbers from binary into decimal for output using a long, slow series of divisions by 10. The IBM Type 650 was a typical example of this class; a random memory access on it took 2.5 ms, and it typically ran about 1000 instructions per second, but needed 16.9 ms for a division, so converting a single six-digit number from binary to decimal for output would have required a full tenth of a second — much longer than was required to punch it into a card.

As computers got faster, conversion to decimal ceased to be the computational bottleneck, and IBM moved its decimal-computer customers over to the binary System/360, but included instructions in the 360 instruction set for operating on binary-coded decimal numbers as well, stuffed two digits per byte, but only in memory (not in registers).

The primary instruction set we use today, AMD64, is derived ultimately from the instruction set of a 1971 Japanese pocket calculator, whose 4004 CPU was developed by Intel. The 4004 had a DAA instruction, "Decimal Adjust Accumulator", used to convert the result of a 4-bit binary arithmetic operation into a 4-bit BCD result; Intel's manual explains:

> The accumulator is incremented by 6 if either the carry/link is 1 or if the accumulator content is greater than 9. The carry/link is set to 1 if the result generates a carry, otherwise it is unaffected.

The history of this instruction is as follows:

- 1971: DAA is introduced with the opcode FB_{16} in the 4004, which, though dozens of times faster than the IBM 650, still would have suffered performance problems from having to do a series of divisions in order to update the calculator display.
- 1972: dropped in the 8008, which was a more powerful follow-on to the 4004 but with fatal flaws in its instruction set design. The 8008, with its 8 8-bit registers, has its opcodes divided into octal fields. The opcode space previously occupied by DAA and many other 4004 instructions, $37*_8$, is now dedicated to instructions that store registers into memory. All eight opcodes of $0*7_8$, one of which will be DAA on the 8080, are RET.
- 1974: DAA is reintroduced in an 8-bit version in the 8080, with the opcode 27_{16} (047_8), which had the same registers as the 8008 but was an actually usable CPU. It didn't have multiply and divide

instructions, but its stack was in RAM. Its opcodes are divided into the same bitfields as the 8008's. Like the 8008, it has 8 opcodes dedicated to subroutine returns, but now they're $3*0_8$ and are different conditional returns; maybe that was planned for the 8008 but had to be cut. Unconditional RET is a ninth return instruction, 311_8.

• 1978: renamed "decimal adjust for addition" in the 8086, and accompanied by DAS, AAA, AAD, and AAM instructions that do related things — the AAD and AAM instructions were added to enable decimal arithmetic with the new multiply and divide instructions. The 8086 had 16-bit registers instead of the 8-bit registers on the 8080, but DAA and friends still only work on 8 or even 4 bits of data. I'm not sure what to make of this — they added four new instructions to support decimal arithmetic, but then didn't bother to take advantage of the double-sized accumulator register? The 8086 isn't binary-compatible with the 8080, but DAA retains the same opcode, 27_{16}. There are four RETs: $C2_{16}$, $C3_{16}$, CA_{16}, and CB_{16}, none of which is the 311_8 from the 8080, which is unassigned on the 8086; it's later reassigned to the fairly useless LEAVE on the 80188 and 80186 in 1982.

• 1985: the entire group of instructions is preserved unchanged in the i386, which expanded the registers to 32 bits. At this point it's clearly only being preserved for backwards-compatibility — doing multi-precision arithmetic with it involves using almost an order of magnitude more instructions than doing the arithmetic in binary. This is the first of these updates that's binary-backward-compatible — the 8086 was more or less compatible with the 8080 at the assembly level, but you couldn't run unmodified 8080 binaries on it. The i386 has a variety of processor modes to enable this backward compatibility.

• 2003: AMD introduces the 64-bit Opteron, which expands the registers to 64 bits, and adds a mode bit so it can continue executing unmodified 32-bit code. In 32-bit mode, DAA and friends work as always; in 64-bit mode, they generate an invalid-opcode exception. Intel was trying to escape from backward compatibility with such nonsense with the Itanium, playfully dubbed "Itanic", because of how it sank. Intel followed AMD's lead in 2004.

So, today, if you have some decimal numbers, such as "321.23", "289.528", "9076", and "8", and you want to do some arithmetic on them on a computer, you have a few different options. You can go through a series of multiplications to convert them to binary floating-point and then do arithmetic on them in binary, accepting the inevitable rounding errors on numbers like "0.1"; you can pick a prescale, like 100, and convert them to binary integers instead (for example, "8" becomes 800, and "321.23" becomes 32123) and then do arithmetic on those; or you can use a software implementation of decimal math like the Python `decimal` module.

(The computationally expensive part, actually, is not converting from decimal to binary on binary hardware; that's only a series of multiplications, which are fast. It's converting back the other way, which is a series of divisions, which are slow.

John Cowan quoted Wikipedia:

> IBM POWER6 includes DFP in hardware, as does the IBM System z9.1 SilMinds offers SilAx; a configurable vector DFP coprocessor.2

b484dfaa84a14c3b366574f300d21d974a61c348e42376fobe9bd0a64479oo
oe095 - is the sha256 of this line: Author: Kragen Javier Sitaker. Salt: "Ez)g['7Hbvy

Polynomials over GF(2)^n

Computing properties of LFSRs and CRCs require doing computations with polynomials over GF(2)^n. Addition and subtraction of these polynomials are just XOR, but multiplication and division are more complicated operations.

SSE4.2 added a CRC32 instruction which performs an XOR and a polynomial division by the polynomial $11EDC6F41_{16}$, the CRC32C polynomial; it runs a bit over twice as fast as a software implementation, which makes me think that hopefully better alternative uses for the silicon are available. As far as I can tell, though, you can't use the SSE4.2 instruction to, for example, do Rabin-Karp string search or rsync sliding-window duplicate probing.

Arithmetic Coding

???

Cellular Automata

The Cytocomputer was some custom silicon designed in the 1970s at ERIM for some machine-vision applications; essentially it was a pipelined two-dimensional cellular automaton which processed one pixel (or cell) each clock cycle, with one line plus two pixels of latency per pipeline stage, using FIFO buffers. Each generation of the cellular automaton was run on a separate hardware stage of the pipeline, which could therefore easily be programmed for a different transition rule. Typical rules did things like dilation, erosion, and edge detection.

Cellular automata, especially those with a small number of states, are just about the worst possible case for per-instruction overhead.

XXX...

Lattices

XXX

Complex numbers

Base -1+i, "proposed by S. Khmelnik in 1964 and Walter F. Penney in 1965," allows us to represent the Gaussian integers as finite bitstrings without needing sign bits. The addition and multiplication operations on these numbers form a commutative ring, indeed, a Euclidean domain, so even division can be sensibly defined. I am not sure if there is a way to preserve these properties with a finite bit length, the way ordinary unsigned binary arithmetic does with GF(2^n). The fourth power of the base is real ($(-1+i)^4 = -4$) and so, in the complex plane, bit shifts by multiples of four amount to scaling a 2-D plane by a power of 4 with some number of half-turns.

Complex numbers can be used to represent geometrical points, translation (by addition), and rotation with scaling (by multiplication).

So it might be worthwhile to add hardware to implement the

relevant addition and multiplication algorithms for base -1+i.

p-adic numbers

p-adic numbers (for some prime p) form a field. This gives Hehner and Horspool's "quote notation" for finitely representing rational numbers with simple arithmetic algorithms, although that notation is not now considered practical due to exponential size explosions on some common cases.

Other rings

XXX

Topics

Paper editing

Kragen Javier Sitaker, 2017-06-15 (3 minutes)

Often, drawing tools on computer screens suck, especially on the tiny computer screens we're starting to use more often ("cell phones", "tablets"). But computers are coming with better and better cameras, printing on paper remains high-quality and relatively cheap, and there are a wide variety of tools available for drawing and otherwise making images on paper — pens, markers, pencils, erasers, rulers, compasses, paintbrushes, paint, ink, and so on. Furthermore, it's easier and easier for a computer to recognize barcodes and things like that on photographed or scanned paper, which allows it to precisely coregister things drawn on preprinted paper, both spatially and in colorspace.

Existing similar things

There's a web site called Fontifier where you print out a template with a box for each letter, draw letters in all the boxes, then scan in the template and upload it to their web site. The website converts the uploaded scan into a TrueType font and sells it to you.

Scantron machines optically detect the positions of pencil marks on preprinted paper slips, used to encode the answers to multiple-choice exams, marking each one correct or incorrect.

Presumably human-readable bureaucratic forms are commonly processed by OCR nowadays.

Possibilities

So, what new possibilities does this modality of interaction offer us?

Most obviously, you could print out a document that needs illustrations with blank pre-sized boxes for the illustrations, illustrate it by hand, then scan or photograph the document to automatically add the illustrations to the document. Barcodes on the paper would orient the illustration-cropping software, and if the printer is colorimetrically calibrated, color calibration patches on the paper can correct for unknown lighting color, camera white balance, and camera focal plane sensitivity.

Additionally, though, if you can print out an image in a single color channel, such as magenta, then you can use it as a reference for another image you draw on top of it in black — whether merely marking points or areas of interest on the underlying image with "X" or "O" or similar marks, marking arbitrary contours, or even drawing an arbitrary overlay image. If grayscale isn't needed, you can even do this on a monochrome printer by printing the reference image in light gray, then thresholding it away to leave only the human-made black marks.

For example, you could draw accent marks for a font on a preprinted letter glyph to provide reference, places of interest on a map (perhaps with their names or other data such as hours of operation), coloring or shading on an outline drawing, connections on a network diagram, data points on a scatterplot with a preprinted reference graticule.

Topics

- Human–computer interaction (p. 3493) (76 notes)
- Cameras (p. 3364) (8 notes)
- Tangible interfaces

Notes on the value restriction and Modula-3

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

(Warning, this note is kind of rambling with no real point.)

Finkel says, describing Modula-3's subtyping rules:

> If every value of one type is a value of the second, then the first type is a called a 'subtype' of the second. For example, a record type TypeA is a subtype of another record type TypeB only if their fields have the same names and the same order, and all of the types of the fields of TypeA are subtypes of their counterparts in TypeB.

Initially, this struck me as violating ML's "value restriction" and therefore being unsafe, but I was wrong. Suppose we say

```
type Ushort = 0..65535;
     Long = -2147483648..2147483647;
     TypeB = record
       Data: Long;
     end;
     TypeA = record
       Data: Ushort;
     end;
```

Now it seems that `Ushort` is a subtype of `Long`, because every value of type `Ushort` is a value of type `Long`. So it is safe to assign a value of type `Ushort` to a variable of type `Long`. `TypeA` and `TypeB` have fields of the same names in the same order, and the single field in `TypeA` has a type that is a subtype of the single field in `TypeB`.

However, I guess if you're passing by value, you can safely copy a record of type `TypeA` into a variable declared as a record of type `TypeB`, and you're still safe with no run-time type checks.

Mutable Aliases Introduce Problems

It's only once you get into aliasing that you start running into problems; if you have a record that is mutably accessible through both `TypeA` and `TypeB` pointers, you can set its `Data` to `-1` through the `TypeB` pointer and then access it through the `TypeA` pointer.

It seems that if you could ensure that the `TypeB` pointer were `const` --- that is, didn't allow modification of the pointed-to value --- you could avoid this. That would allow the following subtyping rule:

```
if                  TypeA is a subtype of TypeB
       ------------------------------------------------------
then   pointer to TypeA is a subtype of pointer to const TypeB
```

That allows you to copy a pointer to `TypeA` into a variable declared as pointer to const `TypeB`. Which is pretty much equivalent to copying a `TypeA` into a variable declared as `TypeB`, but more efficient.

Arrays

When it comes to arrays of the same element type but varying sizes, the definition of "A is a subtype of B" that works in the above

subtyping rule is "A's indices are a superset of B's indices". That is, if you have an `int[100]` array, it's perfectly OK to copy a pointer to it someplace that is expecting a pointer to a `const int[10]` array; no run-time type checks will be needed. Actually, though, that's true even if that pointer isn't `const` --- which I guess is Finkel's point when he distinguishes "extensions" from "subtypes".

Lack of Conclusions

Anyway, just some interesting things I hadn't realized before about safe static typing. There's a subtyping relation that still applies after you take mutable references, and another one that doesn't, and the one that doesn't can be pretty broad.

Topics

Two-thumb quasimodal multitouch interaction techniques

Kragen Javier Sitaker, 2018-04-26 (11 minutes)

Most of the time people use their multitouch Android phones with one or two thumbs, the better to type on the keyboard at the bottom of the screen. But the direct-maniplation possibilities of this interaction paradigm are not very well explored, beyond tapping items to select them and scrolling vertically.

One-finger scrolling is such a ubiquitous interaction that not supporting it — or turning it into something incompatible — is going to be painful.

However, you could quite easily use one-thumb quasimode shifts to identify an action to be taken using objects selected using the other thumb. It's easy enough to support three or so separate buttons per thumb, and sliding after touch could select from a pie menu or adjust two parameters. Ideally actions like "undo" should require a slide, so that you can touch the button to see what you can slide to without fear of invoking an action. If you have three buttons per thumb and six slide directions, you have 36 actions or quasimodes available plus 6 further quasimodes.

Also, when you tap an object to select it, you can pop up another radial menu around the object, which can display detailed properties of the object; tapping on the menu items can pop up submenus accessible by dragging, invoke quasimodes, or permit a two-dimensional continuous adjustment of two parameters. A common pattern might be to display the numerical value of an attribute, drag to adjust it continuously, or tap to enter a new value with the keyboard. Under the same six-direction constraint, this allows another 42 object-specific actions.

Initially I thought that perhaps the object context menu should disappear as soon as your touch ends, so that the menu itself was a sort of quasimode; but that introduces three problems. First, you can't select from that menu by dragging, because that would interfere with one-finger scroll, so you have to use your other thumb — but half the time the menu item you want will be on the wrong side of the menu, so you have to cross your thumbs, or start over. Second, part of the menu is hidden under your thumb, so you can't see what the options are. Third, you give up the possibility of quasimodal menu items on the object menu.

Here are some example applications.

Interactive geometry

(See also Interactive geometry (p. 508).)

This is a program like KSEG, with points, lines, line segments, rays, circles, labels, and measurements, computed in a DAG, each possessed of color and layer properties; layer visibility can be toggled.

Panning, zooming, and rotation of the canvas are conducted using one-finger drags and two-finger pinches and rotations. Tapping an object highlights it and brings up a context menu depending on the object type; additionally it may bring up other menu options

elsewhere on the canvas.

All operations that you might want to apply to a number of objects at once or in quick succession, or on no objects, are necessarily on global quasimode options rather than individual object context menus.

The lower left corner has translucent quasimode buttons for "move", "trash", and "create point". Tapping in an empty space flashes the "create point" quasimode button, since maybe that's what the user meant to do. While that button is held, any taps on the canvas create points, which are initially selected and thus display their context menus. While "trash" is held, skulls and crossbones hover over all the objects visible on the canvas; tapping them will delete them. "Move" causes chaos symbols to hover over all the objects visible on the canvas, and the objects can then be moved with another finger — indeed, you can move more than one object at a time this way.

While you are holding down one of these buttons, the lower-right quasimode button cluster is replaced with a popup view of the radial menu of the button you're currently on, with a shaded cone connecting it to that button to clarify that it's a sort of magnified view. Dragging up and left on the "trash" button executes "undo"; dragging down and right executes "redo".

The lower right corner has a similar set of quasimode buttons. One is the "layer" button, which displays the name of the current layer and whether it is visible. Holding it invokes a quasimode that moves any object then tapped to that layer. Dragging it down and left and up and right cycle among the layers, with the possibility of cycling through more than one layer if the motion is extended. Other drag directions let you change the name of the layer, toggle its visibility, or add a new layer (initially labeled with the ten Heavenly Stems). Another is the "style" button, which shows the current style applied to new objects on this layer (or objects moved to this layer), and which either primarily shows color, line style, or point style (selectable by dragging up and right or down and left), and invokes a quasimode to change its focused aspect of objects while held. Other drag directions on "style" allow you to change attributes of the focused aspect of the style (for example, by invoking a modal popup color selector, or changing the thickness and dash pattern of lines) or to hide or show labels (itself a quasimode that causes all objects' labels to appear in gray, allowing you to tap objects to hide or show their labels). The submenus associated with these pop up in the lower left-hand corner, replacing the left-hand buttons.

The "layer" button in particular allows you to easily create a new layer with one drag, hold it to point to the objects you want to move onto it (tapping them a second time to undo the action, perhaps), and then tap it again to hide them all. Layers are automatically unhidden when switching to them. This provides a conveniently reversible way to hide objects.

Constructions with two objects as inputs are handled entirely through object context menus, because you can't really tell where you're going to start a drag — fine selection requires that you be able to correct your course by dragging after making initial contact. So, for example, to create a triangle, you create three points with the point quasimode, then tap one of them to bring up its context menu.

Holding the "segment" option on the context menu to activate a segment quasimode, the other points light up (as would, for example, tangent points on a circle and perpendicular points on lines and segments) and you tap two of them, then release "segment". Then you tap one of them, hold its "segment" item, then tap the other and release. While the quasimode is active, possible objects to construct appear dimly, lighting up when your finger is touching in the right place to construct them.

Other context-menu two-input constructions include perpendicular through a point, parallel through a point, circle from center and radius, circle from center and point on circumference, perpendicular bisector, angle bisector, or line or ray through two points. Also, you can constrain or unconstrain a point to lie on some other object, go to the object's layer, make the object's style the current style, or change the object's style.

So, for example, circle-from-center-and-radius is just a tap, a hold, and a tap, and creating two circles with the same radius and different centers just requires one additional tap, as long as you had the foresight to start the construction from the radius rather than the center. In Geometer's Sketchpad, each circle is two clicks to select the inputs and two clicks in a pull-down menu, so you pay eight clicks for two circles instead of three taps and a hold.

However, intersection points, midpoints, and points constrained to an object are constructed using the point quasimode — you just drag your finger around until you get the option you want, displayed in a callout.

Constructions of more than two inputs use a different interface. The third lower-right button opens a cartoon of the construction on top of the canvas; you then specify correspondences between the formal-parameter objects in the cartoon and the corresponding actual parameters in your sketch by touching them at the same time — effectively the formal parameter objects are quasimodal. Once only one formal parameter remains unspecified, you can see previews for possible formal-parameter values as you drag around the canvas.

Dragging on that constructions button scrolls around a universe of possible constructions to try.

A drawer that can be slid out from the right side with a tab contains a step-by-step textual description of the sketch, and objects can be selected in either the canvas or the text.

Other miscellanea include creating text objects, creating formulas, creating loci, creating new constructions (including by refactoring), entropy-minimizing point coordinates via implicit slow snaps, and file management.

Whenever an object is selected, its parents and children are also highlighted in different colors.

Prototype plan

To see if that's a good idea, I need to write a prototype, and DHTML is the easiest way to do that. But right now I don't have hardware that I can usefully get multitouch events from in DHTML: I can't find my new iPhone, my modernish Android phone has a power fault, I don't have a charger for my old iPhone (and I don't know if it will charge), and although my ancient Android phone delivers touch events in its browser, it seems to unavoidably collapse

into pinch-zoom mode as soon as there's more than one touch on the display at once.

So I'm going to start by prototyping on my laptop, using the keyboard for the quasimodes.

The very most minimal functionality that can possibly be interesting to look at at all is:

- Making points.
- Making lines between those points.
- Moving the points.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Multitouch (p. 3591) (12 notes)
- Quasimodal (p. 3674) (2 notes)

Can you eliminate backpatching?

Kragen Javier Sitaker, 2019-12-17 (8 minutes)

Conventional compilers, assemblers, and linkers require some non-sequential access to their output stream in order to insert forward references: places where a piece of code must contain the address of some later piece of code or data. Normally this is done by maintaining the output program in RAM along with a database of unresolved references and "backpatching" it when one of those references is resolved.

But can we avoid this, so that all code can be output as soon as it is generated, and still have a practical programming system? I mentioned this in a conversation with Jeremiah Orians, and I think I see a couple of ways.

How assemblers backpatch

In assemblers this is conventionally fairly unrestricted: you can mention an undeclared label `foobar` wherever, and this generates an undefined relocation for that label. If you define `foobar` later on, either as a label or with EQU, the assembler may go back and backpatch all the previous `foobar` references, but if you don't (and maybe even if you do), it leaves the job to the linker.

In conventional assemblers, this single mechanism provides for procedure call, looping, conditionals, exception handling, other local control flow, function pointer generation, and references to statically-allocated variables, and in some cases even to things like thread-local variables.

How Forth backpatches

Forth systems normally don't do a lot of this. Forth words are defined purely in terms of words (subroutines and variables) defined previously, textually prior to them, so they don't need to do any forward referencing for procedure calls, access to statically-allocated variables, or function pointer generation, and so no backpatching is needed in these cases. Mutual recursion can be arranged by defining a "DEFERred" word which is later updated to invoke a word defined textually later in the source; in a sense this is backpatching, but DEFER permits this definition to be changed multiple times, including at runtime --- it's a statically-allocated function pointer variable.

For control flow within a subroutine, though, Forth systems conventionally do backpatch in three cases: loops that may run zero times (such as ?DO loops), loops that may exit in the middle (such as BEGIN WHILE REPEAT loops), and conditionals (IF ELSE THEN). For these purposes, they maintain a compile-time stack both to compile backward jumps (which require no backpatching) and to backpatch forward jumps.

Is there no way to avoid this?

A sorting linker

As I said before, if your compiler or assembler relies on the linker to put in the forward references, it doesn't need to backpatch them itself,

so it can emit code immediately, at least if it doesn't care too much about optimizing it. But that just palms the distasteful random access off on the linker; it doesn't eliminate it. Can you do a linker without random access?

Well, sure, if you accept a lot of sequential access instead. You can shred the code into snippets that start at each symbol reference, each tagged with its start address (which may mean that you have some zero-length snippets), and mergesort them by the symbol name, while you mergesort the symbol definitions in a separate file. Then you can do a sort-merge join between the two files, formatting each symbol in the appropriate way for each relocation, to produce a file of properly linked snippets, still in symbol-name order. Then you can sort this file by start addresses and merge the overlaps where there were multiple symbol references at the same address, probably of different relocation types. Easy. Lots of "compilers" in the 1950s worked this way, I imagine.

But it's not highly efficient, if your objective really is to compile programs substantially larger than your RAM; you may need quite a few passes over your sequential files to finish the link.

Conditional returns

Suppose instead we structure all of our conditionals as conditional returns. Consider abs(x) = x if x > 0 else -x. You can compute this with the following sequence of operations:

```
result = x
x > 0?
if so, return
negate result
return
```

The "if so, return" conditional requires no backpatching to implement, perhaps a forward jump over a return instruction, and indeed instruction sets like ARM (see My very first toddling steps in ARM assembly language (p. 1684)) implement it directly.

However, this comes at a heavy cost. The more general case f(x) = g(x) if h(x) else j(x) can be implemented straightforwardly like this:

```
result = g(x)
h(x)?
if so, return
result = j(x)
return
```

But this involves computing g(x) in cases where we don't want it; this is at least computationally wasteful, and if g(x) has some observable effect, actually incorrect. You can do it correctly at moderate cost with an auxiliary function:

```
f'(x, c) =
    c?
    if not, return
    result = g(x)
    return
```

```
f(x) =
    c = h(x)
    result = f'(x, c)
    c?
    if so, return
    result = j(x)
    return
```

Note that this can be compiled purely in sequence from the definition "f(x) = g(x) if h(x) else j(x)", as long as we allow ourselves to revise our notion of f's entry point once we get to the "if"; and this is true for arbitrarily long inline computations in place of g, h, and j, *as long as j contains no conditionals*. That is, it doesn't generalize to the usual conditional cascade form "g(x) if h(x) else (j(x) if k(x) else m(x))", since the test for h(x) would need to come after the test for k(x) in the object code to prevent forward references. Although this is much less useful, it does generalize to the form "(g(x) if h(x) else j(x)) if k(x) else m(x)", which arises from PHP's incorrect associativity of `?:`:

```
f''(x, c') =
    return unless c'
    result = g(x)
    return

f'(x, c) =
    c' = h(x)
    result = f''(x, c)
    return if c'
    result = j(x)
    return

f(x) =
    c = k(x)
    result = f'(x, c)
    return if c
    result = m(x)
    return
```

All of these pieces can contain arbitrary exit-at-the-bottom loops and other computations, though.

If we replace the `return` at the end with a jump back to the top of the function, what we have is a normal exit-at-the-top while loop where the code before the conditional can do arbitrary computation, like Forth BEGIN WHILE REPEAT.

So this approach requires you to write your code in a pretty strange style with, usually, at most one conditional per function, and the conditional written *after* the consequent but before the alternate, as in Python conditional expressions. But it does seem feasible.

Conditional call and return

If we suppose that we have a conditional call instruction (or instruction sequence), we can do this a little more reasonably:

```
f'(x) =
```

```
    result = g(x)
    return

f(x) =
    c = h(x)
    result = f'(x) if c
    return if c
    result = j(x)
    return
```

If we have conditional call, we don't really need conditional return:

```
f'(x) =
    result = g(x)
    return

f(x) =
    c = h(x)
    result = f'(x) if c
    result = j(x) unless c
    return
```

However, this restricts the form of the j(x) expression; it needs to be nothing more than a function call. No such restriction applies to g or h, and if you have both conditional call and conditional return, it doesn't apply to j either.

Topics

- Programming (p. 3658) (286 notes)
- Independence (p. 3520) (63 notes)
- Small is beautiful (p. 3714) (40 notes)
- Assembly language (p. 3328) (25 notes)
- Forth (p. 3461) (19 notes)

Using the method of secants for general optimization

Kragen Javier Sitaker, 2019-07-22 (updated 2019-11-26) (13 minutes)

The method of secants is an algorithm the humans have been using for some 3000 years to solve a fairly wide variety of inverse problems, or, to their modern way of thinking, find zeroes of a fairly large class of functions. Given a function f from some vector x to the underlying field of that vector, such as $\mathbb{C}$ or $\mathbb{R}$, we compute a sequence of iterative approximations:

$$x_n = x_{n-1} - f(x_{n-1}) \cdot (x_{n-1} - x_{n-2}) / (f(x_{n-1}) - f(x_{n-2}))$$

You can think of this as a variant of Newton–Raphson iteration, using the secant approximation to the tangent line.

Unlike Newton–Raphson iteration, it doesn't require computing the derivative of the function, and it has slightly faster convergence under the usual assumptions used to prove the convergence of Newton–Raphson iteration — under those circumstances, the error after each iteration is the previous error to the power $\varphi \approx 1.618$, but each iteration only requires computing a single new value of the function being solved for, while Newton–Raphson iteration requires computing a new value of the function and also of its derivative, which is usually about twice as much work. So in some sense it converges about 31% faster.

However, it needs two guesses to get started instead of one, which makes its convergence conditions somewhat more complicated to describe.

Some ideas occurred to me about how to use the method of secants, so I thought I'd write them down.

I used the method of secants as an extended example in Separating implementation, optimization, and proofs (p. 780).

(This is probably crushingly naïve compared to all the work out there on optimization methods I don't understand yet, like Nelder–Mead, Broyden's method, and the Levenberg–Marquardt algorithm, not to mention the stunning successes in recent years with variants of gradient descent; but I thought it would be worth writing down.)

(Unrelated: the Method of Wecants, a technique for declining to do something you don't want to do while blaming someone else.)

Vector-domain functions

The method of secants is normally described as a one-dimensional root-finding method, but above, I said that you can generalize the domain of the function to be a vector, as suggested by the form of its recurrence relation. What happens in practice if you try that?

Consider $f : \mathbb{R}^2 \to \mathbb{R} = \lambda(a, b).a^2 + b^2 - 1$, a paraboloid which is zero everywhere along the unit circle. If our initial starting guesses are $x_0 = (1, 1)$ and $x_1 = (2, 0)$, the values just sort of randomly oscillate:

- 1, 1
- 2, 0
- 0.5, 1.5

- −1, 3
- 0.8, 1.2
- 1.04545455, 0.95454545
- 4.29411765, −2.29411765
- 0.89511609, 1.10488391

That's because each x value in the method of secants is an affine combination of the previous two values, so there's no way for them to get off the line $b = 2 - a$; and, as it happens, that line doesn't intersect the unit circle. If you're looking for an intersection of that line with the circle, or more likely some hairy implicit function, that might be great — although, if there are multiple intersections, there's no guarantee about which one you'll get, unlike with signed-distance-function raytracing. But if you're trying to find *any* solution, it's not so great that you need to start by picking two points that are collinear with it.

Other pairs of starting points converge just fine for the same function:

- 1, 0.5
- 2, 0
- 0.90909091, 0.54545455
- 0.86206897, 0.56896552
- 0.80697224, 0.59651388
- 0.80049797, 0.59975101
- 0.80000430, 0.59999785
- 0.8, 0.6

There are different approaches to solving this problem. Perhaps the simplest possible one is to use a secant, not through the last two points, but through the first and last of the last m points:

$$x_n = x_{n-1} - f(x_{n-1}) \cdot (x_{n-1} - x_{n-m}) / (f(x_{n-1}) - f(x_{n-m}))$$

In theory this should allow the last m points to be a simplex of an $m-1$-dimensional space, so their affine combinations would be that $m-1$-dimensional space, at the expense of somewhat slower convergence. This seems too dumb to work, but it does seem to. Here's a Python implementation:

```
def secnd_seq(f, x):
    x = list(x)
    y = [f(xi) for xi in x]

    while True:
        yield x[-1], y[-1]

        div = y[-1] - y[0]
        if not div:
            return

        x.append(x[-1] - y[-1] * (x[-1] - x[0]) / div)
        y.append(f(x[-1]))
        x.pop(0)
        y.pop(0)
```

Given random points from $[-5, 5]^2$, this converges to a point on the unit circle about ¼ of the time with 2 starting points (the orthodox

method of secants), but almost always with 3 or more starting points, because three points is enough to span the whole 2-D parameter space almost surely. However, the algorithm frequently takes several thousand iterations to converge! Increasing the number of starting points to 4, 5, or 6 makes it less frequently need more than 100 iterations or more than 1000 iterations, but since it usually converges in less than 50 iterations with 3 points, it might make just as much sense to do a random restart if the algorithm is failing to converge. Still, increasing the number of points more makes the completion time more consistent.

```
unit_circle = lambda (x, y): x**2 + y**2 - 1

def test_nd(d=3, n=1000, maxiter=100, eps=1e-15, f=unit_circle):
    ok = not_ok = 0

    for i in range(n):
        x = [numpy.random.random(2) * 10 - 5 for j in range(d)]
        items = list(itertools.islice(secnd_seq(f, x), maxiter))
        if abs(items[-1][-1]) < eps:
            print "ok:", x, items[-1][0], len(items)
            ok += 1
        else:
            print "not ok", x, items[-d:]
            not_ok += 1

    return ok, not_ok
```

(After several thousand trials, it did find a three-starting-point state from which convergence failed after 10000 iterations: three points reported as [array([-2.57650664, -4.90971528]), array([-0.17240513, -3.30215595]), array([-4.9655625 , 3.50688737])]. Unfortunately, that point converges in 82 iterations; like Lorentz's, my attempt at reproducibility has apparently been defeated by rounding.)

This experimental result suggests that this approach may be usable in high-dimensionality spaces to find zeroes, perhaps, like gradient descent is for finding minima. But I wonder how it compares to running gradient descent, or one of its modern variants like AdaGrad or Adam, on the square of a function?

Finding a zero of a vector-valued function

Suppose you want to find an intersection (x, y) of two circles: $(x - x_0)^2 + (y - y_0)^2 - r_0^2 = 0 \wedge (x - x_1)^2 + (y - y_1)^2 - r_1^2 = 0$. You could think of this as finding a zero of a vector-valued function $f(x, y) : \mathbb{R}^2 \to \mathbb{R}^2 = ((x - x_0)^2 + (y - y_0)^2 - r_0^2, (x - x_1)^2 + (y - y_1)^2 - r_1^2)$. But we can't directly apply the method of secants, because we need to divide by the difference of two function outputs, and you can't divide vectors.

However, we can take a norm of the result vector to get a scalar which will only be 0 when the vector is 0; for example, the $\mathbf{L_1}$ norm: $|(x - x_0)^2 + (y - y_0)^2 - r_0^2| + |(x - x_1)^2 + (y - y_1)^2 - r_1^2|$. Or, in Python:

```
def circle_intersection(x0, y0, r0, x1, y1, r1):
```

```
    def f(xv):
        x, y = xv
        return (abs((x - x0)**2 + (y - y0)**2 - r0**2) +
                abs((x - x1)**2 + (y - y1)**2 - r1**2))

    return f
```

This of course has a discontinuous derivative whenever we cross one of the circles, and so although the procedure above is able to find intersections successfully, it takes several hundred iterations to do so. But, surprisingly, things get even worse if we try to use the L_2 norm:

```
def circle_intersection_L2(x0, y0, r0, x1, y1, r1):
    def f(xv):
        x, y = xv
        return (((x - x0)**2 + (y - y0)**2 - r0**2)**2 +
                ((x - x1)**2 + (y - y1)**2 - r1**2)**2)

    return f
```

I think this is because the quadratic convergence condition of both Newton–Raphson iteration and the method of secants requires the function to have a nonzero derivative in the neighborhood of its root, and the L_2 norm instead has a zero derivative. Still, by using enough initial points, we can usually get to the solutions this way; this converges on 997 out of 1000 attempts, but usually takes between 1000 and 2000 iterations:

```
test_nd(d=20, maxiter=10000,
        f=circle_intersection_L2(0, 0, 1, 1, 0, 1),
        eps=1e-13)
```

The L^∞ norm is just as bad.

Optimization

Thomas Simpson pointed out that, if you can compute the first and second derivatives of a function, you can use Newton–Raphson iteration to numerically approximate its critical points — saddle points, local minima, and local maxima. Its global minimum must be one of these or, if its domain is compact, a point on the boundary of the domain. (If its domain is finite and open it may not have a minimum, just an infimum.) In low dimensionalities with sufficiently polite functions, this can enable you to quickly find the global minimum, even by manual computation. This approach has expanded into a whole field of "quasi-Newton methods", but these involve maintaining an approximation of the Hessian matrix of the function being optimized — and in n dimensions, the Hessian has n^2 elements.

Similarly, you can use the method of secants to numerically approximate a critical point of a function if you can compute the function's derivative — for example, using automatic differentiation. Earlier, I suggested that maybe you could square a function, at least a real one, and use generic optimization algorithms to find its zeroes. Now I'm suggesting almost the opposite: differentiate a function and

use generic root-finding algorithms to find the zeroes of its derivative, then test them to see which one is lowest.

If we are attempting to find a minimum of a function $f: \mathbb{R}^n \to \mathbb{R}$, we can start by using automatic differentiation to compute points of ∇f, which are in $\mathbb{R}^n$. Then we can, perhaps, use the extended method of secants in the way described above — taking some norm of that gradient and attempting to extrapolate to where it becomes zero, using something like $2n$ or $3n$ points — with only on the order of $3n + O(1)$ operations per iteration, rather than the $O(n^2)$ required by quasi-Newton methods. (However, I suspect that convergence, if it happens at all, may be slower per iteration with this approach, so overall it may be faster or slower than quasi-Newton methods.)

Genetic algorithms

"Genetic algorithms" is a catchy name for a popular metaheuristic based on Darwinian and Mendelian metaphors. You have a "population" of "chromosomes" over which you compute "fitnesses"; then, to create a new "generation", you "crossbreed" members of the population chosen randomly (with higher probability increasing with fitness) by combining their "genes" with "crossover" and apply "mutations" to the results. As long as the crossover and mutation operations aren't too destructive to fitness, each generation of the population will tend to have higher and higher "fitness".

It occurred to me that the method of secants sort of fits into this mold, but with an especially powerful "crossover" mechanism — it attempts to extrapolate from the differences between the two "genomes" it's crossbreeding to find the optimum. The offspring is necessarily within the space of affine combinations of the two parents, but its fitness may be much higher than theirs. (But sometimes it's not.)

The above approach of just drawing secants from further back in the history of approximations, in order to handle higher dimensionality, can be seen as a sort of degenerate genetic algorithm in which we only crossbreed two individuals at a time, and they're always the oldest and youngest. Presumably more judicious choice of parents would yield faster convergence.

As an example, I was thinking of approximating an image (see Image approximation (p. 2394)) with a set of Lambertian spheres of the same color and some lighting, and using the method of secants as above to generate new combinations of spheres. The function being optimized would be a combination of the difference between an image rendered from a given configuration of spheres and a penalty for the complexity of the configuration; the configuration would consist of an (x, y, z) direction for the directional light, an (r, g, b) color for the ambient light, and parameters $(x_i, y_i, z_i, R_i, r_i, g_i, b_i)$ for each sphere in the configuration. Mutation operations would add noise to everything and occasionally clone a sphere. Initial renderings would be low in resolution to speed up the initial search.

Dunno, maybe something like that could work for the structure-from-shading or even structure-from-motion problem, though in the second case you additionally have to estimate the camera path relative to the object. And maybe gradient descent and its variants are better fits.

Topics

A stack of stacks for simple modular electronics

Kragen Javier Sitaker, 2017-06-27 (5 minutes)

(See also files Graph construction (p. 3226) and Circuit notation (p. 1161).)

I was reading about Ekawahyu Susilo's "SnapBloks" modular circuit design kit, each of which has an STM32 microcontroller inside, and I had an idea.

Stack machine instruction sequences are nice and flat, even as they express nested expression trees. Even three or four levels of stack are adequate for fairly complicated trees.

What about physical circuit building blocks you can plug together into a linear sequence? Things much more primitive than an entire microcontroller. Like resistors. Maybe you could have, say, nine wires, representing a stack of seven signals plus two power rails; simple passive wiring components would be adequate for DUP, SWAP, DROP, ROT, pushing Vcc and GND onto the stack, and shorting the two top stack-input lines together; two-terminal components like diodes and capacitors would be packaged to connect between the "input" and "output" on the top-of-stack line, running everything else straight through; logic gates and transistors and the like would fit into the scheme reasonably easily. Physical distances between components could be fairly short if each layer in the stack were only, say, 1 mm thick.

A Forth-style two-stack version with enough wires would be adequate for any series-parallel circuit, I think, as well as many that aren't.

If you could change the orientation of processing modules in the stack, you might be able to get some extra expressiveness. Diodes, of course, could be turned around, but perhaps you could flip things around to select which of two stacks you were operating on.

A "vectorized" version in which each signal line was replaced by, say, a four-wire bus, would be no less expressive (you could always ignore the extra wires, though driving them would consume power) and more expressive for some uses. You'd want some extra "permuting" and "reducing" kinds of operations to interconnect the bitplanes.

If you had eight "stack items" of four wires each, plus power and ground, you'd have 34 lines. If the spacing was the standard 1.27 mm SOIC pin spacing with a 6×6 square matrix, the whole module size would be 6.35 mm square plus the width of a pin — say, 7 mm × 7 mm. This is a practical size to recognize by eye and manipulate by hand. An assembled circuit containing 50 elements and 50 passive "stack operations" might then be 7 mm × 7 mm × 100 mm (and cost about US$50 in parts, plus maybe another US$1000 in modules that were sitting around that you didn't use). A huge variety of circuits can fit within these constraints, especially when the available components include things like programmable microcontrollers.

About orientations: a square pin array like that could be oriented in eight orientations (four in-plane rotations and two flips), while the

slightly denser hexagonal packing would have twelve possible rotations. Getting these rotations to do something useful would be an interesting challenge, the alternative being to add a forcing function to reduce the possible number of rotations.

By way of size comparison, a AA battery is 14 mm × 14 mm × 50 mm. In that size, you could fit four separate such stacks of 50 items, totaling 200 modules and about 100 electrical component modules (i.e. not just passive wiring). So this seems like it might be a very reasonable alternative to breadboarding.

If the hypothetical 34 lines were distributed around the edges of the board at the same 1.27 mm spacing, you'd need 43.18 mm circumference, 13.74 mm diameter, the same as a AA battery. This could conceivably enable a larger number of possible rotations, but you'd have to figure out how to hook up the power rails, maybe with coaxial pins in the center of the board.

Is 1 mm a reasonable thickness? Common 0805 surface-mount resistors are 2 mm × 1.2 mm × 0.45 mm. A TQFP is nominally 1.0 mm, not counting the leads bending down below the body; an MLF is also nominally 1.0 mm. Standard PCB thicknesses (including 4-layer and 6-layer PCBs) are 0.031" and 0.062", apparently without regard to the number of copper layers, which work out to 0.79 mm and 1.57 mm. So 1 mm is maybe a bit low, but 3 mm should be doable. (Of course, not all the modules have to have the same thickness.)

I don't know what kind of connectors the boards would have to use between them. I assume zebra strip would have unacceptably high resistivity for many uses.

Topics

Why Minetest is so addictive

Kragen Javier Sitaker, 2019-04-20 (8 minutes)

I haven't played Minecraft, but I'm terribly addicted to Minetest, an open-source clone of it. What is it that makes the game so captivating?

When I start a fresh game, there's a carefully graduated slow exponential increase in the character's power. At first they can only dig dirt, break trees with their hands, and transplant plants; once they've acquired some wood to make a wooden pickaxe, they can mine stone and make a stone pickaxe, which allows them to mine stone more quickly, and then a stone axe, which allows them to chop trees more quickly; and some coal, which allows them to make torches, which makes it practical to mine underground and in caves, which makes it far easier to find coal and then metals; and so on. The sequence varies a bit depending on what environment your character finds themself in. I had a world where I had to travel a long way to find any wood, for example, and where I never did find any roses, which are necessary to make a bed. In the world I'm currently playing, it took me a long time to find a jungle, which is where you find the jungle grass you can convert into cotton seeds in order to start farming cotton, also necessary for beds.

There's a certain balance needed between surface farming and deep mining. There's a progression of tool types — wood to stone to steel to bronze to crystals (diamonds and the softly glowing mese crystals) — which roughly corresponds to how deep you need to go to find the necessary materials. But to dig deep quickly you need ladders, and ladders require lots of wood, which can only be obtained on the surface. (Mina likes playing with mods that add monsters, make ladders cheaper, and add hunger, which also generally requires that you go to the surface to satisfy it.) At different points in the game, different resources are the limiting resource for whatever you want to do, and you can play for many hours before you run out of things in the world that surprise you.

And, in the mining process itself, you're always looking for something that's fairly rare, and you will run across its random occurrences at a frequency proportional to how hard you look. At the surface, even coal may require searching for quite a while, and iron is not found at all. Later, once you have iron tools, you can easily dig down far enough that you can find more coal and iron than you need, but if you know bronze tools are better, it takes a while to find enough copper. And once you're deep enough to find enough diamond and mese that you don't need to use metal tools any more, it can still take a lot of searching to find each new crystal deposit, and you're frequently in danger of stumbling into lava. This proportional variable reward mechanic means that, in addition to the overall macroscopic reward schedule for continuing to play, you have a microscopic reward schedule where you know that the very next stone you punch with your pick could have diamonds or mese crystals behind it, making the whole journey down from the surface worthwhile. This makes it easy to keep playing for just one minute more. In fact, it makes it easy to forget to eat in real life. Or sleep.

Or go to class. Or go to work.

All this is to say that the reward schedule is well calibrated, somewhat randomized, and robust to player strategy. But I think there's another reason as well.

My apartment is three meters wide, 2.6 meters tall, and thirteen meters long. At one end, it has a glass window, which is about two meters square. The walls, ceilings, and floors are white, and there is a bit of a recess around the edges of the ceiling. I pay US$300 a month to live here, despite the inconstant and often uncomfortable temperature and troublesome noises, especially on weekend nights.

I just constructed a sort of replica in Minetest, in a world where I had already been playing and thus already had crystal tools, about 40 meters underground rather than in the sunlight. Replicating my real-world living space as described above took me 33 minutes; although that doesn't include the time to cultivate, harvest, and mine the wood, diamond, sand and coal consumed in the process, I think the direct labor on the building was the bulk of it. Merely digging out a 13×3×3 space out of gray rock would have been much quicker, less than five minutes, but I made the extra effort to reduce the ceiling height from the usual 3 meters to 2.5, adding the glass window at the end with a light source behind it, and making the walls, ceiling, and floor white with a sandstone pattern like the one printed on my real-world ceramic floor tiles.

Of course, the Minetest replica doesn't have a bidet, air conditioning, internet access, hot water, natural light, or conveniently walkable 24-hour hamburger shops like my real-world apartment does. But it's a lot tidier, and I can remodel it completely in another half-hour if I want. Moreover, I'm not likely to run out of space to put things, which in real life can be a problem; this world currently has about 1 km × 2 km generated horizontally, and I've dug almost 500 meters down, and it's only 49 megabytes, very little of which is the underground palace I added the apartment replica onto.

Aboveground, the palace does have its own stand of lumber trees, a giant outdoor fountain I built and am using to irrigate a cotton crop, and a watch tower with a beautiful view of the sunrise; indoors, it has a mushroom cultivation area, a 49-square-meter dojo with a mosaic floor, chests full of copper and gold ingots, ovens baking bread from wheat cultivated upstairs, a fireplace, and a room illuminated through its glass floor by lava running below. (In another Minetest world, I built a continuously erupting volcano visible from my balcony that, unfortunately, burned down a forest.)

So, like nearly all video games, Minetest offers a sense of competence and progressively increasing power. Minetest is also a medium of expression like that provided by painting and CAD programs. But, probably most addictively, Minetest is a sort of animated dollhouse: a vehicle for a convincing fantasy of living a good life, though it is a life involving a great deal of smashing rocks and hauling them up mine shafts. Like literal dollhouses, you can play in it alone or with friends or even strangers; it supports sharing your virtual reality over a LAN or the internet.

Also, in some ways, it goes beyond literal dollhouses: it provides an illusion of travel. Like any first-person three-dimensional game, the screen contents are in almost constant apparent motion toward you, triggering your orienting response and making it easy to pay

attention, and you can travel around, explore the world, learn where things are, and find more things.

Topics

Alastair thesis review

Kragen Javier Sitaker, 2013-05-17 (1 minute)

Reading Alastair Porter's master's thesis.

It provides a comprehensive overview of how current audio fingerprinting algorithms work, what they are used for (both for good and for evil), and a more detailed comparison of the open-source Echoprint and Chromaprint algorithms and an open-source Matlab implementation of the Landmark or Shazam algorithm.

It's a gold mine of references to things I didn't know about, or that I hadn't heard of before Alastair told me about them: mel-frequency cepstral coefficients, the freesound archive, the Shazam and Soundhound smartphone apps for song identification, the three open-source audio fingerprinting algorithms he evaluated, a list of other companies that do audio fingerprinting, the Fourier-Mellin transform, an adaptive FFT, Query By Humming, the Codaich dataset, the Acoustid and Echoprint servers, etc. Its overview of the history of audio fingerprinting for automated censorship in Napster, YouTube, and other applications may be shocking for those who aren't familiar with the situation.

The research involved making a fingerprint database of the thirty thousand two hundred eighty-three songs in the Codaich dataset and running twenty thousand queries (trying to match some audio fragment) against it. But they didn't just do that once; they did it many times with different fingerprinting algorithms and different kinds of degradation on the queries.

Topics

- Algorithms (p. 3310) (123 notes)
- Audio (p. 3331) (40 notes)
- Music (p. 3593) (18 notes)
- Book reviews (p. 3347) (5 notes)
- Shazam

Multitouch and accelerometer puppeteering

Kragen Javier Sitaker, 2019-08-29 (updated 2019-09-01) (12 minutes)

In Dercuano drawings (p. 64) I've talked a bit about possible tools for illustrating Dercuano without bloating it, and in Two-thumb quasimodal multitouch interaction techniques (p. 1765) and $1 recognizer diagrams (p. 1264), among others, I talked about possible multitouch interaction idioms that could make this more practical. But now I want to talk a bit about possible ways of doing *animations*.

Bret Victor's turning-leaf animation demo

Bret Victor's lecture *Inventing on Principle* contains a demo of a prototype multitouch animation application designed to circumvent the limitations of things like Flash by, among other things, recording the motion paths of fingers and using them to define motion paths for cels, not just in position but also other dimensions such as rotation. I had never seen an animation program work like this before, but it turns out that the pioneering animation program GENESYS already worked this way in 1970, I think on the TX-1, though using a light pen rather than a capacitive multitouch screen.

There's a 1970 video of GENESYS on YouTube; watching that and *Inventing on Principle* are essential inspirations.

Two-dimensional puppeteering

GENESYS's motion paths were essentially recorded puppeteering, but were only capable of supplying two dimensions at any given time. This requires an additional pass over the animation to supply other information, such as when to switch the image of a character between different images — in GENESYS's case, as in traditional cel animation, discrete images, but as I pointed out in $1 recognizer diagrams (p. 1264), morphing between pen strokes is a straightforward thing to do.

Dimensions you might want to supply to a character path in an animation are nearly unlimited — even with a single sprite image and enough computrons to remap space in real time, they could include X, Y, rotation, scale, stretch angle and direction, transparency, brightness, contrast, tint, and two dimensions of perspective distortion. If an animation character has multiple images available, they can be arranged in a multidimensional space using dimensions like position-in-stride, mouth openness, age, fury, angel wings, deadness, left hand X and Y, right hand X and Y, head position, and explodedness, or a smaller number of dimensions could be used with a sort of K-nearest-neighbor interpolation instead.

More dimensions

In Victor's demo, he concurrently supplies a third dimension — rotation — by using a second concurrent finger. As I noted in Two-thumb quasimodal multitouch interaction techniques (p. 1765), this is typically the limit of what people do with modern

hand computers — although typical screens and OSes track up to five touches, it's unusual for people to use more than two at a time. Moreover, it's common for one or both of the touches to be thumb touches confined to an area near a corner of the screen, while the rest of the user's hand is supporting the computer from behind.

(Some of them also track touch size and orientation, but these variables are not very controllable, and are not available on all touchscreens.)

I think it might be feasible to get people to puppeteer an animated characters using two fingers on handles near the character, as with pinch-zoom but with less finger occlusion, while their thumb near the corner navigates a couple more dimensions of animation space. Moreover, more than two handles may be available, so the position at which they bring down a second finger onto the screen can indicate which dimensions they would like to be puppeteering at that moment.

(Choosing among different candidate characters to puppeteer is another reason for using handles positioned near the characters onscreen.)

Some animated characters, like Victor's leaf, will rotate freely, but many characters have a preferred orientation: a car or a text string might be horizontal, while a human figure might be vertical. Even rotatable characters might not rotate all the time. This means that with two fingers on handles near the character, the user can both move the character freely and continuously vary two more behavioral dimensions (rather than zooming and rotating with the second finger), such as mouth position and eyebrow position. The thumb in the corner can simultaneously be varying another pair of dimensions or invoking other operations.

Marionettes are commonly operated primarily with a 6DoF controller consisting of two crossed sticks, with the option to pull additional strings with a finger. The accelerometers in modern hand computers similarly offer two more degrees of freedom, which can perhaps be used advantageously.

This suggests that it should be possible to use a modern hand computer for real-time puppeteering with 8 degrees of freedom varying simultaneously in real time, with some of them chosen from among some 10 to 16.

Interaction with previous recording

The above is perhaps fine for when you're recording a new animation, or puppeteering live for an audience or to play a game, but when you're editing an existing animation, there's the question of how new puppeteering movements interact with the previously recorded ones. Perhaps they overwrite them — this should at least be an option — or perhaps they normally just add to them. This is especially troublesome if the primary handle you use to select a character to puppeteer is also the handle you use to move it around: does that mean you have to overwrite the existing movement path in order to add new eyebrow movements? Perhaps there's a fuzzy transition from merely selecting to overriding movements.

We could think of these degrees of freedom as "tracks" that we're recording, like in a multitrack digital audio workstation; as in GENESYS and Victor's demo, we'll need to be able to see the tracks

on the screen, select parts of them to be copied, moved, or erased, and so on.

Track inference

Probably by default some characters should infer tracks like moving through walking paces, making jumping motions, breathing, and turning to face different directions from lower-dimensionality data, like position and movement direction and speed. Butterflies flutter, falling leaves twirl, cars rock back and forth, humanoids turn to face the direction they're walking, toons wind up and screech to a halt, balls bounce, frogs hop, and so on. Users can always control these tracks explicitly if they like.

To some extent you could think of this as being like the "artistic style transfer" problem that so many artificial neural network papers have made so much spectacular progress on recently: for a character to walk realistically, they need to move not only their legs but also their hips, their arms, their shoulders, and their spine, and ideally those movements can be generated from a model of the movements the puppeteer is controlling explicitly. I suspect some kind of K-nearest-neighbors kind of thing based on an existing motion corpus, mixed with some linear prediction, might be sufficient, but maybe you'd end up needing a more sophisticated model like a neural network. Regardless, the space is of very small dimensionality relative to the spaces recent papers are training neural networks on, so the problem should be significantly more tractable once you have some data.

Face tracking

An alternative to puppeteering facial expressions with multitouch is to use the hand computer's front camera to detect a human user's facial expressions in real time so they can be used to provide some of the tracks. (Latency may be a problem at present.) This is most intuitive for humanoid character facial expressions, of course, but it could be used for a variety of other tracks as well.

Multiscreen puppeteering

Many popular puppets are operated by more than one person simultaneously in order to handle more degrees of freedom than a traditional marionette, without losing the immediacy of real-time performance. Different people, each using a personal hand computer, could also do this.

Extending this further, you could strap several old cellphones with accelerometers (and perhaps with broken screens) to different parts of a person's body in order to get a real-time feed of rotoscoping-like information, with two degrees of freedom per cellphone; then you might be able to entirely avoid the use of touchscreens.

Yaw-axis and position detection

Above I suggested that the accelerometers in a modern hand computer offered two degrees of freedom, because most of the signal they provide just tells you which direction gravity is. In fact they usually provide six readings: three axes of accelerometer and three of "gyroscope", but these do not provide absolute position and yaw

information, just pitch and roll, as it were. Typically they also have a flux-gate compass as well, but its readings are slow and noisy, so you have to filter it over a considerable period of time to get a useful reading.

However, you might be able to use the slow filtered compass reading to provide a sort of low-frequency yaw signal, then the faster (but relative, so high-frequency-filtered) gyroscope signals to update it in real time. Similarly, it might be reasonable to assume that on average (over, say, a minute) the computer is stationary, and use the shorter-term variations in accelerometer readings as adjusted by the gyros to compute where in the local space it is.

Of course, none of the above is new — inertial navigation systems have worked like this since, I think, the 1940s, though using actual gyroscopes and larger accelerometers. But using it for puppeteering or rotoscoping purposes seems to be new.

Photogrammetry could provide an additional source of high-precision ground truth for absolute yaw measurements.

Non-sprite virtual puppets

In addition to simple 2-D sprites, you could use this approach to puppeteer 3-D models of varying degrees of sophistication, as well as 2-D models more complex than simple sprites: 2-D models with skeletons, say, or assemblages of separate sprites for different body parts. Particle systems can be attached to the puppets, perhaps just in certain circumstances. Drawings can be progressively traced.

Dancing in more abstract spaces

Of course, interactively defining a time-indexed parametric curve in spaces of 16 dimensions or so, with real-time feedback, has many applications other than imitating Walt Disney. Modern video-game character models commonly have more dimensions than this, but the games have only very clumsy ways of navigating them, navigating from slider to slider.

Ad-hoc mathematical models of phenomena — physical system simulation, like Victor's active-filter example in the same talk, or otherwise — can easily have that many dimensions to explore.

Fractal graphics commonly have many continuous parameters which are interesting to explore.

A music synthesizer, too, has many such parameters, and recording how they change over time is a crucial part of recording music. This, especially with the accelerometers, is of course the core of Onyx Ashanti's inspiring work on Beatjazz.

Plots of multivariate data can only display a few dimensions at a time; varying plot parameters dynamically, as with one of Victor's other demos in the same talk, is useful not only for exploration but also for explanation to an audience.

Theater lighting is another case where you need to record a path through a continuous multivariate space and then play it back, typically with some degree of interactive control over timing.

Manually planning a motion path for an industrial robot or CNC machine is another possible use. Of course, you can also use this approach for control of such a machine in real time.

Topics

- Human–computer interaction (p. 3493) (76 notes)
- Multitouch (p. 3591) (12 notes)
- Hand computers (p. 3492) (10 notes)
- Animation

A sketch of a minimalist bytecode for object-oriented languages

Kragen Javier Sitaker, 2017-03-20 (updated 2017-06-20) (13 minutes)

Suppose you have a programming system with a garbage-collected object-graph memory where objects are type-tagged immutable vectors of fields. You really only need two stack-bytecode operations for the object memory:

```
NEW ( class [ val0 val1 ... -- ref )
AT ( ref idx -- val )
```

Here `[` is a PostScript-style stack mark for variadic function invocation.

`AT` fetches a field value, and `NEW` creates an object containing the given fields. Alternatively, you could bake `idx` into AT's bytecode and have it always operate on `self`, the receiver containing the executing method, and NEW could be an ordinary method call on `class` which is implemented with magical native code. Either way, you need bounds-checking either at compile-time or run-time.

With a reasonable generational garbage collector, this can be the most efficient way to do many things. Other algorithms, however, also need mutation, so you need a third operation to mutate fields: ATPUT.

For a Pythonish language (see Thredsnek: a tiny Python-flavored programming language (p. 1172)) you need built-in dict, list, int, float, and probably string classes, plus reflection, dynamic dispatch of variadic method calls, an iterator protocol, and exceptions; and, of course, control flow.

So a minimal OO Pythonish bytecode might need INVOKE, AT, ATPUT, VAR, VARPUT, LIT, IF, GOTO, LOOP, [, and RET: 11 operations. I'm not quite sure how to implement exceptions; catch and finally blocks are rare enough in garbage-collected languages (unlike in C++ or Rust, where every new variable sort of introduces one) that doing it the straightforward way with PUSHJMPBUF and POPJMPBUF opcodes might be adequate.

An alternative block-structured design might unify method scope, object scope, and global scope as simply being three arbitrary levels of an arbitrary set of nested scopes, and this does have benefits to recommend it. However, this requires recovering at compile-time or even run-time further information that was available in the programmer's mind: in particular, which blocks might become closures (i.e. when objects are created) and which variables are captured by those blocks (i.e. the instance variables of the implicitly created objects).

Looking at existing bytecode may be instructive. Here's disassembly of the CPython bytecode for a somewhat arbitrarily chosen Python library function, the `difference_update` method of the deprecated `sets` module.

```
Disassembly of difference_update:
```

```
479           0 LOAD_FAST                0 (self)
              3 LOAD_ATTR                0 (_data)
              6 STORE_FAST               2 (data)

480           9 LOAD_GLOBAL              1 (isinstance)
             12 LOAD_FAST                1 (other)
             15 LOAD_GLOBAL              2 (BaseSet)
             18 CALL_FUNCTION            2
             21 POP_JUMP_IF_TRUE        39

481          24 LOAD_GLOBAL              3 (Set)
             27 LOAD_FAST                1 (other)
             30 CALL_FUNCTION            1
             33 STORE_FAST               1 (other)
             36 JUMP_FORWARD             0 (to 39)

482     >>   39 LOAD_FAST                0 (self)
             42 LOAD_FAST                1 (other)
             45 COMPARE_OP               8 (is)
             48 POP_JUMP_IF_FALSE       64

483          51 LOAD_FAST                0 (self)
             54 LOAD_ATTR                4 (clear)
             57 CALL_FUNCTION            0
             60 POP_TOP
             61 JUMP_FORWARD             0 (to 64)

484     >>   64 SETUP_LOOP              33 (to 100)
             67 LOAD_GLOBAL              5 (ifilter)
             70 LOAD_FAST                2 (data)
             73 LOAD_ATTR                6 (__contains__)
             76 LOAD_FAST                1 (other)
             79 CALL_FUNCTION            2
             82 GET_ITER
        >>   83 FOR_ITER                13 (to 99)
             86 STORE_FAST               3 (elt)

485          89 LOAD_FAST                2 (data)
             92 LOAD_FAST                3 (elt)
             95 DELETE_SUBSCR
             96 JUMP_ABSOLUTE           83
        >>   99 POP_BLOCK
        >>  100 LOAD_CONST               1 (None)
            103 RETURN_VALUE
```

The original source code:

```
def difference_update(self, other):
    """Remove all elements of another set from this set."""
    data = self._data
    if not isinstance(other, BaseSet):
        other = Set(other)
    if self is other:
        self.clear()
    for elt in ifilter(data.__contains__, other):
```

```
        del data[elt]
```

Here's a longer method, from another obscure standard library module, `netrc.netrc.__repr__`:

```
Disassembly of __repr__:
130           0 LOAD_CONST               1 ('')
              3 STORE_FAST               1 (rep)

131           6 SETUP_LOOP             137 (to 146)
              9 LOAD_FAST                0 (self)
             12 LOAD_ATTR                0 (hosts)
             15 LOAD_ATTR                1 (keys)
             18 CALL_FUNCTION            0
             21 GET_ITER
        >>   22 FOR_ITER                120 (to 145)
             25 STORE_FAST               2 (host)

132          28 LOAD_FAST                0 (self)
             31 LOAD_ATTR                0 (hosts)
             34 LOAD_FAST                2 (host)
             37 BINARY_SUBSCR
             38 STORE_FAST               3 (attrs)

133          41 LOAD_FAST                1 (rep)
             44 LOAD_CONST               2 ('machine ')
             47 BINARY_ADD
             48 LOAD_FAST                2 (host)
             51 BINARY_ADD
             52 LOAD_CONST               3 ('\n\tlogin ')
             55 BINARY_ADD
             56 LOAD_GLOBAL              2 (repr)
             59 LOAD_FAST                3 (attrs)
             62 LOAD_CONST               4 (0)
             65 BINARY_SUBSCR
             66 CALL_FUNCTION            1
             69 BINARY_ADD
             70 LOAD_CONST               5 ('\n')
             73 BINARY_ADD
             74 STORE_FAST               1 (rep)

134          77 LOAD_FAST                3 (attrs)
             80 LOAD_CONST               6 (1)
             83 BINARY_SUBSCR
             84 POP_JUMP_IF_FALSE      114

135          87 LOAD_FAST                1 (rep)
             90 LOAD_CONST               7 ('account ')
             93 BINARY_ADD
             94 LOAD_GLOBAL              2 (repr)
             97 LOAD_FAST                3 (attrs)
            100 LOAD_CONST               6 (1)
            103 BINARY_SUBSCR
            104 CALL_FUNCTION            1
            107 BINARY_ADD
```

```
              108 STORE_FAST               1 (rep)
              111 JUMP_FORWARD             0 (to 114)

136       >>  114 LOAD_FAST                1 (rep)
              117 LOAD_CONST               8 ('\tpassword ')
              120 BINARY_ADD
              121 LOAD_GLOBAL              2 (repr)
              124 LOAD_FAST                3 (attrs)
              127 LOAD_CONST               9 (2)
              130 BINARY_SUBSCR
              131 CALL_FUNCTION            1
              134 BINARY_ADD
              135 LOAD_CONST               5 ('\n')
              138 BINARY_ADD
              139 STORE_FAST               1 (rep)
              142 JUMP_ABSOLUTE          22
          >>  145 POP_BLOCK

137       >>  146 SETUP_LOOP              85 (to 234)
              149 LOAD_FAST                0 (self)
              152 LOAD_ATTR                3 (macros)
              155 LOAD_ATTR                1 (keys)
              158 CALL_FUNCTION            0
              161 GET_ITER
          >>  162 FOR_ITER                68 (to 233)
              165 STORE_FAST               4 (macro)

138           168 LOAD_FAST                1 (rep)
              171 LOAD_CONST              10 ('macdef ')
              174 BINARY_ADD
              175 LOAD_FAST                4 (macro)
              178 BINARY_ADD
              179 LOAD_CONST               5 ('\n')
              182 BINARY_ADD
              183 STORE_FAST               1 (rep)

139           186 SETUP_LOOP              31 (to 220)
              189 LOAD_FAST                0 (self)
              192 LOAD_ATTR                3 (macros)
              195 LOAD_FAST                4 (macro)
              198 BINARY_SUBSCR
              199 GET_ITER
          >>  200 FOR_ITER                16 (to 219)
              203 STORE_FAST               5 (line)

140           206 LOAD_FAST                1 (rep)
              209 LOAD_FAST                5 (line)
              212 BINARY_ADD
              213 STORE_FAST               1 (rep)
              216 JUMP_ABSOLUTE          200
          >>  219 POP_BLOCK

141       >>  220 LOAD_FAST                1 (rep)
              223 LOAD_CONST               5 ('\n')
              226 BINARY_ADD
```

```
           227 STORE_FAST               1 (rep)
           230 JUMP_ABSOLUTE          162
       >>  233 POP_BLOCK

142    >>  234 LOAD_FAST                1 (rep)
           237 RETURN_VALUE
```

Here's the original source code:

```
def __repr__(self):
    """Dump the class data in the format of a .netrc file."""
    rep = ""
    for host in self.hosts.keys():
        attrs = self.hosts[host]
        rep = rep + "machine "+ host + "\n\tlogin " + repr(attrs[0]) + "\n"
        if attrs[1]:
            rep = rep + "account " + repr(attrs[1])
        rep = rep + "\tpassword " + repr(attrs[2]) + "\n"
    for macro in self.macros.keys():
        rep = rep + "macdef " + macro + "\n"
        for line in self.macros[macro]:
            rep = rep + line
        rep = rep + "\n"
    return rep
```

I feel like these two functions are relatively typical Python code.

From a few arbitrary files including the above, I disassembled about 2650 bytecodes. The top ops are:

```
LOAD_FAST (630 bytecodes),
CALL_FUNCTION (242 bytecodes),
LOAD_GLOBAL (238 bytecodes, usually in order to call a global),
LOAD_ATTR (228 bytecodes, largely split between
    self attributes (preceded by a LOAD_FAST of self) and
    method calls (followed by a CALL_FUNCTION)),
LOAD_CONST (179 bytecodes),
STORE_FAST (150 bytecodes),
RETURN_VALUE (132 bytecodes),
POP_JUMP_IF_FALSE (95 bytecodes),
POP_TOP (89 bytecodes), and
COMPARE_OP (80 bytecodes).
```

These add up to 2063 bytecodes, about 78% of the total. COMPARE_OP makes it there because it's a catchall for a variety of comparison operators, including exception-matching, `==`, and `is`, but also `in`, `>`, and the like. Most other binary operators have their own bytecode, like `[]` (BINARY_SUBSCR, 30 occurrences), and `+` (BINARY_ADD, 26 occurrences).

Stack operation density

I've tried a number of things, but so far I haven't been able to find a way to get tighter code than with a stack machine. On average, a binary operation (two inputs and one output) on a stack machine has about one associated stack manipulation operation, which is typically between 5 and 8 bits.

Typical two-register code has a 3- or 4-bit field for each operand, so 6–8 bits, and sometimes has an additional MOV instruction associated to prevent one of the inputs from being clobbered; three-register code avoids that at the cost of a third operand field, which in extreme cases (like Lua bytecode) means 24 bits of operands per operation, but is more typically 12–18 bits.

The Mill architecture's "belt" is a third alternative; like stack machines, the output of each instruction is implicit, but unlike them, inputs are not consumed. Consequently operands must be implicit. I did some speculative analysis (Golomb-coding operands as belt offsets likely won't increase code density much (p. 1605)) of variable-length encoding of a two-address belt code and found that in the code I looked at, the spans were very nearly geometrically distributed, so nearly the optimum was Golomb coding with a bin size of 1, which reduces to unary coding, and this works out to a bit over 5 bits of operand information per two-operand computational instruction (would be 4 if the geometric distribution was exactly correct), which is essentially exactly the same code density as the stack code, just much more expensive to decode.

(An encoding variation I haven't explored in more detail: since about half of the operands are the output of the immediately previous instruction, make that implicit for two-operand instructions, and insert an extra copy instruction in the other cases. It probably won't work to improve density further, but it might.)

The TRIPS or EDGE ("explicit data graph execution") architecture is sort of a mirror image of the "belt", where instruction inputs are implicit, but outputs are explicit, and point to other instructions in the same block. Presumably this works out about the same.

So I think that probably stack code is the best approach; it typically uses about 10–12 bits per binary computational operation, of which about 5–6 bits is operand information (in the form of stack manipulation operations), which is comparable to the other alternatives; I'm not yet convinced that anything is better. It's faster to decode and interpret than variable-length instructions, though slower than fixed-width register-based instruction sets. And it easily accommodates operations that consume or produce unusual numbers of values.

Truncation and Packing

The F18A core in the GreenArrays GA4 and GA144 chips has four 5-bit instructions per 18-bit memory word (when there are no jump targets; immediate literal constants are pushed on the stack by a `@p` instruction anywhere in the previous instruction word). This means two bits are missing from the last word; the instruction encoding is designed so that among the 8 expressible instructions for this last slot are `;` (return), `.` (nop), `dup`, `+`, and `unext` (which loops back to the beginning of the instruction word a number of times stored in the R register).

In this way, you can always NOP out the truncated instruction slot if you can't do anything useful with it, but you *can* fit many of the most common instructions — including particularly the ones that are most likely to be useful at the end of an instruction word. If you can do this ¾ of the time, you can fit 15 instructions into four 18-bit

words, an average of 4.8 bits per instruction — and, as I suggested earlier, typically you need about two instructions per computational operation, giving 9.6 bits per computational operation, not counting jump targets.

You could use a similar approach with 32-bit words of bytecode on a 32-bit or 64-bit machine. If each opcode is 6 bits, for example, you can fit 5 whole ones and one 2-bit truncated one into a 32-bit word, or 4 whole ones and two 3-bit truncated ones.

However, jumps will use up the rest of the instruction word; the above **repr** example code contains 99 bytecodes, of which 4 were `FOR_ITER` or `JUMP_FORWARD`, for an average of about 24 bytecodes between jumps. This is probably roughly typical, or slightly high, so a substantial fraction of words will be taken up with such things.

I'm not counting method invocation as a jump (though it is on the F18A), because I propose to carry it out in a different way, as explained below about "Facets and Dispatch".

Type tagging

For a Pythonish language, we really need some kind of dynamic dispatch that allows us to use the same bytecodes to operate on native integers as on, for example, rational numbers. One way to do this — the approach CPython takes — is to make integers heap-allocated objects too. But it really helps performance, not to mention reliability and timing data leaks, if ordinary integers don't need to be stored in memory like that.

The standard approach to support runtime polymorphism between integers (held in registers) and pointers is to use one to four bits of the address word as type tags. On byte-addressed machines with four-byte alignment for objects, the lowest two bits of a valid object pointer are always 00, so you can set them to, for example, 01 to indicate integers. Then

64-bit machines with byte-addressed memory have

Facets and Dispatch

Constant literals

Static Checking

Topics

- Programming (p. 3658) (286 notes)
- Programming languages (p. 3656) (47 notes)
- Instruction sets (p. 3526) (40 notes)
- Compression (p. 3384) (28 notes)
- Python (p. 3671) (27 notes)
- Stacks (p. 3730) (21 notes)
- Object-oriented programming (p. 3606) (10 notes)
- Mill (p. 3584) (7 notes)
- Bytecode (p. 3356) (6 notes)
- Minimal Instruction Set Computing (p. 3587) (3 notes)

Append only unique string pool

Kragen Javier Sitaker, 2016-07-27 (2 minutes)

A couple of systems I've been dealing with recently involve reading in a bunch of strings (up to a few million) as fast as possible, storing them in minimal space, and then retrieving them by index. It may turn out in these cases for space reasons to be advantageous to deduplicate the strings, but we can't afford a lot of hashing or hash search time.

The most space-efficient solution to the problem (without resorting to compression) is just to concatenate the strings with delimiters in between, or better, preceded by a variable-length count. Then, retrieving string i requires iterating over the i preceding strings.

That's unacceptably slow both for indexing and for searching for duplicates. If you store a separate array of the indices of the starts of the strings, you can quickly index, but on a 64-bit machine, that array may be rather large; it may be more practical to store the start index of, say, every 8th or 16th string, and then, in a separate array of 16-bit values, the lengths of all the strings. These lengths only cost one more byte per string than inline delimiters, make the system 8-bit clean, and allow for strings up to 64KiB. Now we have constant-time indexing.

Finding duplicate strings requires some kind of additional index, of which a hash table is the simplest. It has the problem that it potentially needs a lot of space if it's full of pointers to strings, and I don't have any particularly clever solution to that, but it shouldn't use much extra time if collisions can be kept low enough; in particular, you can compute the hash function as you're copying the incoming string into the place it will be occupying in the concatenated strings if it turns out not to be a duplicate. For very long duplicate strings, this copy will result in extra memory bandwidth, but for short strings and non-duplicate strings, it won't.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)

A 7-segment-display font with 68 glyphs

Kragen Javier Sitaker, 2017-02-21 (4 minutes)

Seven-segment displays are easily salvaged from lots of electronics. They are somewhat limited in what they can display, but there are 128 possible glyphs, not counting the decimal point which usually accompanies them. Many of them are recognizable letters or punctuation.

The most popular 7-segment LED display on Digi-Key right now is http://www.digikey.com/product-detail/en/kingbright/ACSC56-41CGKWA-F01/754-1047-1-ND/1747764, a Kingbright 2.1V 20mA (pulsed to 150mA in 100µs pulses) common-cathode slightly slanted green LED display with a decimal point, with the totally ridiculously high price of US$2.70. Its datasheet "numbers" the segments as follows:

```
  a
 --
f|g |b
 --
e|  |c
 --
 d  DP
```

with the totally absurd pinout 7, 6, 4, 2, 1, 9, 10, 5, with the last being DP.

If we make a the MSB of a 7-bit word, then the standard glyph for 0 is 0x7e, with 0x7f being 8. The numeric weights in hexadecimal are as follows:

```
  40
 --
2|1 |20
 --
4|  |10
 --
 8
```

A more complete font looks like this:

```
' ' 0x00                          '⌝' 0x60
'-' 0x01                          '⊃' 0x61
''' 0x02    '"' 0x22    '⌝' 0x42    'ⁿ' 0x62
',' 0x04                '⊂' 0x43    '°' 0x63
'r' 0x05
                                  '?' 0x65
'|' 0x06                'Γ' 0x46
'⊢' 0x07    'µ' 0x27     'F' 0x47    'P' 0x67
'_' 0x08
'=' 0x09                '≡' 0x49    '⊇' 0x69
```

```
                    '⊆' 0x4b    'º' 0x6b
'ㄴ' 0x0c
'c' 0x0d    '¿' 0x2d            '2' 0x6d
'L' 0x0e            '[' 0x4e
't' 0x0f            'E' 0x4f    'e' 0x6f
'i' 0x10    '1' 0x30            '7' 0x70
'¬' 0x11    '⊣' 0x31

            '4' 0x33    'ς' 0x53    'q' 0x73

'n' 0x15            'ñ' 0x55
                                '∩' 0x76
'h' 0x17    'H' 0x37            'A' 0x77
'ㅣ' 0x18                        ']' 0x78
                                '3' 0x79

            'y' 0x3b    '5' 0x5b    '9' 0x7b
'u' 0x1c    'J' 0x3c    'ū' 0x5c
'o' 0x1d    'd' 0x3d    'ō' 0x5d    'a' 0x7d
            'U' 0x3e    'Ḡ' 0x5e    '0' 0x7e
'b' 0x1f    '∀' 0x3f    '6' 0x5f    '8' 0x7f
```

I've tried to omit duplicates here. These 68 glyphs don't include all the letters in even the English alphabet (missing are 'k', 'm', 'v', 'w', 'x', 'z', and very sadly 's'), but they include a fairly complete repertoire of logical operators (at least if we interpret set arithmetic as logic) except for ∃. They include a perhaps unreasonable number of bracketing characters.

The absence of 's' and 'w' eliminates many of the most common English words: 'is', 'was', 'with', 'have', 'this', 'his', as well as most plurals. 'm' also eliminates 'from'. So you can't use this font to write in English, unless maybe you use 'ς' for 's'. You can't even write "sin" and "cos".

Many old typewriters didn't have separate characters for "l" and "1", or "0" and "O"; the number row started at "2" and ended at "9". Due to typewriter legacy, even today, we often use the same character "'" for "‘" and "’", and the same character '"' for "“" and "”", and the same character "-" as a hyphen, minus sign, em dash, and en dash. If you use the same approach here, allowing the "ς" and "2" glyphs to double as "s" and "z", then you get back trig operations and most common English words. The most common ones missing are "was with have from which we were would will what who more them some him two time my like me now".

These glyphs do not include any arithmetic operators other than '-'. Some calculators use 0x05, 0x21, or 0x25 for "/" for a fraction bar.

For the more limited purpose of rendering decimal or hexadecimal digits, you want the list {0x7e, 0x30, 0x6d, 0x79, 0x33, 0x5b, 0x5f, 0x70, 0x7f, 0x7b, 0x7d, 0x1f, 0x0d, 0x3d, 0x6f, 0x47}.

Topics

Audio logic analyzer

Kragen Javier Sitaker, 2019-11-12 (3 minutes)

Suppose you want a logic analyzer, and you have electronics, but you don't have a screen; you only have a speaker or headphone jack and perhaps an LED or two. What can you do?

It seems sensible to map the time domain to the time domain. (However, you will often want about six orders of magnitude slowdown: a logic analyzer needs to be measuring signals of at least 20 MHz, and the humans can only hear signals up to about 20 Hz. Then you can loop the capture if desired.) Then what do you map the different logic channels to?

Probably the most sensible thing to do is to map them to different pitches in the same octave, synthesizing those pitches with rich harmonic content in order to make the perception of both pitch and envelope more precise. Emphasizing onsets with an attack-decay envelope, or the variation in harmonic content that comes inevitably from Karplus–Strong or other waveguide synthesis or that can easily be produced with FM synthesis, might help with some kinds of debugging.

Unfortunately the choice of pitches has a tradeoff between interpretability and aesthetic quality: dissonant pitches are easier to distinguish, particularly if they happen to start and end simultaneously, which will happen frequently on a logic analyzer. Additional features that may help to distinguish the notes might include different speeds, depths, and phases of vibrato, and the kind of flanging effect produced by the beating of the harmonics of nearly-equal-delay strings, as in a piano or a 12-string guitar.

(A counterintuitive feature of modern microcontrollers is that even low-end microcontrollers are more than powerful enough to do real-time multichannel Karplus–Strong synthesis with all-pass filters for variable fractional-sample delays, and additionally they can do digital logic analyzer data acquisition at megasamples per second.)

You probably need some way to configure a Boolean function to trigger a capture. Since you're fiddling around with wires anyway in this scenario, the simplest approach is to use jumper wires to hook up some of the input lines to particular pins — some designated such that their conjunction will trigger a capture, others designated such that any of them will inhibit it. This is not universal but might be enough. However, here in CMOS land, we need to hook those input pins up to *something*; probably the best approach is to just enable pullups or pulldowns and hope that doesn't disturb the DUT too much.

Topics

Antialiased line drawing

Kragen Javier Sitaker, 2018-11-13 (updated 2019-09-01) (4 minutes)

In using matplotlib to do a series of plots, I've found a number of annoying aliasing phenomena.

One phenomenon is that, although the edges of lines and markers are antialiased, the antialiasing is relatively soft. So the lines still contain substantial energy above the Nyquist frequency of the display. When a single line is drawn, this is rarely important, but when many parallel lines are drawn, even if they are perfectly vertical and equally spaced at non-integer pixel spacing, the aliased frequencies can be visible and indeed produce a dominant moiré pattern over the whole area.

Completely eliminating this phenomenon with even a single hatching pattern would require bandlimiting the line's pixel image to strictly below the Nyquist frequency for the display, and of course a strictly bandlimited signal necessarily fills all of space — worse, a bandlimited impulse is a sinc, which drops off in intensity fairly slowly with distance, only inversely proportional to the distance. But it might be reasonable to touch, say, a swath of 8–16 pixels along each edge of the line, rather than 2.

Eliminating moiré for *multiple* hatching patterns would require combining the patterns linearly, which is to say, without opacity. Although this departs from the historical practice developed from pen-and-ink graphics, experience shows that this is a very useful way to plot data on, for example, an oscilloscope display, but it may not be entirely practical for all kinds of data graphics. However, there's a whole range in between, easily accessible in a system that uses premultiplied alpha. RGBA(0, 0, 128, 255) is pure opaque dark blue; RGBA(0, 0, 128, 0) is perfectly transparent luminous dark blue with no opacity; and RGBA(0, 0, 128, 128) can be viewed equally validly as half-transparent bright blue or as a partly luminous, partly opaque dark blue. Indeed, that color is representable even in a system where alpha is not premultiplied (it's RGBA(0, 0, 255, 128)) but other, brighter colors are not.

Note that in this case you want to be able to represent not only overunity brightnesses but also negative brightnesses if you want to plot dark lines on a light background.

The other phenomenon is that, although my LCD uses RGB subpixel ordering, and my font rendering takes advantage of it, matplotlib does not. This results in wasting something like half of my display's horizontal resolution, two thirds for dim blue-tinted graphics. This is really frustrating when I'm trading off legibility for data coverage.

LCD subpixel ordering is not the limit of subpixel antialiasing, either, and I think that even the soft antialiasing matplotlib uses provides visually-subpixel positioning of the line.

Matplotlib of course does not insist on drawing one-pixel-wide lines. You can get it to draw lines several pixels wide. It's just that the particular line profile it uses is a slightly antialiased boxcar: a constant opaque color inside the line, dropping off to perfect transparency outside it. You could easily imagine a better

compromise between precision and visibility — for example, a strong thin line with a diffuse background around it, possibly in a contrasting color, making it easy both to find the line and see its overall shape and to see its precise value.

Topics

- Graphics (p. 3483) (91 notes)
- Displays (p. 3414) (13 notes)
- Numpy (p. 3600) (6 notes)
- Aliasing (p. 3315) (4 notes)
- Matplotlib

Do visually expanding images evoke an orienting response, or the startle response, and what does that mean for ZUIs?

Kragen Javier Sitaker, 2016-06-03 (14 minutes)

In the sci-fi user interface criticism blog [???] the guy complains that it's really bad user interface design for a HUD in a flying machine to have things expand in place, because it triggers a human "startle response", which stresses people out, makes them blink, and distracts them.

Presumably the underlying phenomenon here is that if something is expanding rapidly in place in your visual field, it's usually an object flying rapidly at your face, which means it could hit you soon.

I was alarmed and concerned, because I've been playing a bit with ZUIs, and in ZUIs it's really common for things to expand in place. I started thinking of workarounds: could we maybe zoom into a thing by first zooming in on a point to the right of the screen, then a point to the left of the screen, thus avoiding this response?

But does this visual "startle response" to zooming in exist at all? If so, will it cause problems? Or is the related "orienting response" more relevant? Here's a cursory literature review.

Relevant literature

The Wikipedia article on "startle response" says:

> In animals, including humans, the startle response is a largely unconscious defensive response to sudden or threatening stimuli, such as sudden noise or sharp movement, and is associated with negative affect.1 Usually the onset of the startle response is a startle reflex reaction. The startle reflex is a brainstem reflectory reaction (reflex) that serves to protect vulnerable parts, such as the back of the neck (whole-body startle) and the eyes (eyeblink) and facilitates escape from sudden stimuli. It is found across the lifespan of many species.

Startle and Surprise on the Flight Deck, Rivera et al., 2014. This is a five-page paper on the difference between the "startle reflex" and the "surprise emotion". It says that startle was "studied extensively in the 1970s and 1980s", but refers primarily to the acoustic startle response, although it says, "The startle reflex can be elicited through auditory, visual, or tactile stimuli," citing Carlsen et al. 2008, Davis et al. 1982, and Yeomans et al. 2002. With regard to effects we might care about in ZUIs, it says, "startle has been found to impair information processing on mundane tasks," and "may induce brief disorientation and short-term psychomotor impairments which are likely to lead to task interruptions, or brief confusion," and that "after startle", "decision making can be significantly impaired, especially higher-order functions".

It does mention ways the startle response can be evoked visually: sudden illumination of flight crews by lasers, for example.

The "Startle Reflex" article in the Encyclopedia of Autism Spectrum Disorders, by Sterling, defines it as an "automatic response to the sudden onset of an acoustic, visual, or tactile stimulus… that

varies systematically with the individual's emotional state… characterized by a fast series of muscle contractions around the head, neck, and shoulders;" this says very little about what kind of stimuli will cause it, other than "sudden", or what kinds of cognitive effects it might have.

Human On-Line Response to Visual and Motor Target Expansion, Cockburn and Brock 2005 or so, says that expanding mouse targets in place as the mouse approaches them is "visually appealing" but also "improves acquisition performance". Their main result is that it's easier to click on things when they expand when you move the mouse toward them, and that this is mostly due to the visual feedback, only slightly due to the larger click target. While this is basically not about startling users at all, but rather about how long it takes to click on things when they're expanding (related to Fitts' Law), it provides a sort of useful negative evidence — in their experiments, users were able to click on things faster when they expanded in place, while if expanding stuff in place was invoking a strong "startle reflex", you'd expect to see their performance worsen. But maybe they didn't expand fast enough or far enough, or maybe the performance improvement would have been even greater without the "startle response" confounding it.

The Wikipedia article on "fear-potentiated startle" mentions bright light as a stimulus that elicits "startle", and says that the "startle" is stronger when the organism is afraid, or when the person is traumatized or not depressed, and mentions some symptoms of "startle": blinking and faster pulse. However, apparently, usually the experiments use loud noises.

The Uncanny Valley in Games and Animation, by Tinwell, p.106, section "Lack of Visual Startle Reflex and Psychopathy," says that we make surprised faces when "we experience a sight or sound that frightens or surprises us", except for psychopaths. It's talking about things like "seeing a shadow when we thought we were alone," though, not seeing windows expand on a screen, and it doesn't mention cognitive effects. The author is bringing this up because apparently people who act scared or surprised without showing fear "in the upper facial region" are creepy ("uncanny") and that maybe this is a reason animations can be creepy.

Boo! Culture, Experience, and the Startle Reflex, by Simons, 1996, is a 288-page book about the "startle reflex". It goes into quite a bit of philosophizing about the relationships between culture, biology, and psychology, saying things like, "Being startled changes… one's relationship with the entire experienced universe," and "Pythagoras… showed that the soul is mathematical."

The author is a collaborator of Ekman and Friesen who himself did original research on startle some years back, using pistol sounds.

It explains, "The subject of the book is a more general one: (1) How one's experience of being in the world is shaped by neurophysiology and how it is shaped by prior experience, beliefs, values, and social condition, and (2) how these different types of shaping influences might be consid- ered jointly in single explanatory formulations."

It gives some more detail on what the startle response *is*, exactly (p. 9):

The usual human startle response is precisely the set of behaviors that would be

maximally effective in minimizing the chance of a fatal encounter. The response includes both physical events and alterations in attention, thought, and mood. Instantaneously and without reflection, forward motion is checked. The raised foot is arrested or drawn back and the upper limbs are drawn in to safety. Visual attention is locked onto the eliciting stimulus, whatever had been in consciousness is obliterated, and the mind is filled with one thought: "Snake!" Startling stimuli other than snakes elicit similar immediate patterned rapid-avoidance behaviors and similar redirection of attention.

In the context of ZUIs, redirecting your attention to what you're zooming into might be *good*, and that would be consonant with the positive results Cockburn and Brock got from inflating click targets.

He confirms the 14-ms number Sourakov (the butterfly dude) cites for human startle response time, but for tightening jaw muscles, not blinking.

He cites "Hoffman, 1984, p.275", which turns out to be "Methodological factors in the behavioral analysis of startle," as a paper that mentions a "startling sight." His Table 1.2 on p.14 lists startling stimuli; the visual items include "dangerous or appalling sights" ("snakes and creepy-crawlies, other dangers, appalling sights"), "bright lights", "sudden motion", "great beauty", and "cessation of a stimulus". Later (p.21), he reports on visual stimuli that startled his informant Barbara Haldane:

I mentioned before my startle reaction to the sight of a large spider (or a picture or drawing of one); other large images of things with spindly legs, such as ants and other insects, produce a similar but less-violent startle reaction.

A real snake (but not a picture of one)…

As I told you, the unexpected sight of a human skull--of nearly life-sized or larger proportions, anyway--induces a shock in me … especially if it has dark eye-sockets.

The sight of a larger-than-life human face looking directly at me (at the camera, in the case of a photo) has a similar shock value. I recall one instance of being quite unpleasantly startled by the face of Sophia Loren on a magazine cover, and more than once by an ad in a magazine featuring eyes (larger than life) looking directly at the viewer. (Both eyes usually have to be present — one big eye doesn't have as much effect.) [T]hese reactions are invol- untary and uncontrollable unless the sight in question is antici- pated, and even then I am apt to experience an internal moment of panic.

And his hyperstartler informant Gould startles easily with a variety of visual stimuli, including boys jumping out at her at a ranch, her children stepping out into the hall in front of her, a classroom of her students having turned their desks around while she wasn't paying attention, and putting on the wrong kind of glasses.

The exaggerated startle reaction he reports of Gould, in which she screams for a while, falls to her knees, and throws whatever she's holding, would certainly be a UX problem in a ZUI; it would likely break her phone. However, it seems to be mostly related to seeing unexpected things, not to a lower-level stimulus such as a thing expanding in her field of vision, regardless of whether it's expected or not.

Emotion, attention, and the startle reflex, by Lang et al. in 1990, is a theoretical model of emotion in general, based on the "startle reflex", which is an intellectual overreach if I ever heard of one.

Irrelevant literature

Visual Pathways Involved in Fear Conditioning Measured with Fear-Potentiated Startle: Behavioral and Anatomic Studies, Shi and Davis 2001, startled rats with sounds or electric shocks but *conditioned*

them to associate the "startle response" with visual stimuli. Then they injected different drugs into different parts of the rats' brains to see which parts were involved in the association. This paper doesn't use an innate visual startle response or human subjects, so it is of no interest.

The pupil as a measure of emotional arousal and autonomic activation, Bradley et al. 2013 had people look at pleasant or unpleasant pictures, measuring their emotional response to the pictures by their pupil diameter, skin conductance, and pulse. They also found that people's pupils contracted even more when the pictures were bright than when they were unpleasant. This came up because they startled some people, but not others, with loud noises before showing the pictures. However, they didn't report on the results of this part of the experiment in the paper!

Postural and eye-blink indices of the defensive startle reflex, by Hillman et al. 2003, reports on an experiment where they startled 24 volunteers with loud noises through headphones and measured how much they blinked and cringed, rocking first forwards and then backwards. Their results were that people who blinked more also cringed further backwards.

A Double Dissociation in the Affective Modulation of Startle in Humans: Effects of Unilateral Temporal Lobectomy, by Funayama et al., 2001, found differences in the "startle response" in people who had had one of their temporal lobes surgically removed. They were startling the subjects with loud noises as a way to measure the emotional effect of looking at pleasant or unpleasant pictures or being told they would be electrically shocked. Their results are somewhat involved and they draw a bunch of conclusions about the role of the amygdala in fear that I don't care about.

Perceiving Threat In the Face of Safety: Excitation and Inhibition of Conditioned Fear in Human Visual Cortex, by Miskovic and Keil, 2013, startled 29 subjects with loud noises that the subjects could predict by the speed a light flickered at.

Greater general startle reflex is associated with greater anxiety levels: a correlational study on 111 young women, by Poli and Angrilli, 2015, startled young women with loud noises in both ears after questioning them about how anxious they felt, finding that more anxious women blinked more (but only with their left eyes) and hated the noises more. As background, it mentions that old people and men startle less.

Extraordinarily Quick Visual Startle Reflexes of Skipper Butterflies (Lepidoptera: Hesperiidae) are Among the Fastest Recorded in the Animal Kingdom, by Sourakov, 2009, is about visual startle reflexes in butterflies. I think butterfly eyes and visual cortices evolved separately from our own, so I doubt that conclusions about startle reflexes in butterflies are applicable to humans. It does, however, mention that loud noises can make people blink within 14 ms, puffs of air can make us blink within 30–50 ms, but these butterflies jump into the air in less than the researcher's camera's exposure time of 17 ms when visually startled by the flash; however, it claims that this is "twice as fast as the fastest startle reflex of humans", so, I don't even know.

Topics

Hand drawn font compositing

Kragen Javier Sitaker, 2018-10-28 (2 minutes)

I thought it would be fun to hand-draw a font on paper and scan the paper, but producing a TrueType font from this in the usual way then requires a certain amount of manual work, converting it to Béziers and whatnot. But, in a sense, you don't really need all that manual work; the input is pixels, and the output is also pixels, and the output is a sort of pasteup of the input.

Minimally you do need some way to identify the location and orientation and size of each character in the font on the page, which is best done with some kind of interactive UI. And you might want to separately indicate the character cell and the bounds of the drawn character — either because, in your drawing, some descender or something impinged upon the character cell undesirably, or because you have some kind of swash protruding.

Leaving aside the UX details of such an interface for now, there's the question of how to do the compositing of the different characters to produce the rendered image. Assuming you're drawing on white paper, you'd like to threshold the background paper to pure white, and treat that as transparent. So if you have overlapping bits of letterforms, you'd like to composite them in something like logical-AND fashion:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Now, it might be a reasonable thing to do to actually do this in binary, using oversampling. (Maybe you do 2× oversampling in both X and Y, so that each 32-bit word represents the top or bottom half of 16 different pixels; once you're done with compositing using &, you can use parallel population-count on the subpixels to decide on the grayscale value. The apparent efficiencies of this approach, with an average of 8 antialiased pixels per word instead of 1 or 4, are somewhat compromised by the amount of bit-shifting required.) Alternatively, though, you could use grayscale or even RGB representations directly. And in that case, clearly the compositing operation you want is pointwise *multiplication* followed by rescaling the result.

Topics

Linear trees

Kragen Javier Sitaker, 2016-05-19 (updated 2016-05-20) (6 minutes)

So I've been thinking a bunch lately about the Z-machine's object-tree memory model for embedded systems like the Arduino, as an alternative to the usual C approach of nesting structs and arrays inside other structs and arrays.

There are a couple of problems with the usual Lisp or ML memory model, where all values are either pointers or fit into pointers, and can be referred to from any number of places. One problem is that you almost need a garbage collector, which means that you can only use a fraction of physically available memory for your data; a garbage collector that collects garbage too often will use up the vast majority of your system's runtime. This is a big problem if you have 2048 bytes of memory available. Another problem is that the garbage collector imposes relatively large pauses on things — when you try to allocate and the GC needs to collect first, you get a pause, which can be tens of thousands of instructions long, even if you only have 2048 bytes of memory. A third problem is that the conditions under which your allocation can fail are relatively poorly defined; fragmentation and tenuring, among other things, can keep pieces of memory unavailable that ought to be available, causing allocation to fail unpredictably. A fourth, comparatively minor, problem is that the pointers themselves occupy a lot of memory space, and they are usually a relatively inefficient encoding of the facts about the world with which the program must grapple.

The more subtle underlying problem is that, in some sense, if you're allocating memory at run-time instead of compile time, it's probably because you didn't know how much memory you were going to need when you wrote the program. That, in turn, means that the program could need more memory than is physically present. And that's true even if you don't use a garbage collector.

In desktop and server applications, where programs have error-handling options that include popping up a dialog box with an error message, notifying a user that they are exiting, paging a system administrator, shedding load by dropping network requests without answering them, and logging error messages in logfiles, the occasional failure is accepted as a fact of life.

Embedded applications are different: their options for handling failure are often limited to blinking an LED and rebooting. Ideally you would like your jet engine, robot arm, chemical plant, or antilock braking system to simply not fail, rather than just having ways to report the failure gracefully.

C has other options. It inherits from COBOL, by way of Algol, the possibility of hierarchically composing structs and arrays out of primitive types and other structs and arrays; no pointers are necessary. C code for life-critical embedded systems is usually supposed to obey a set of rules called MISRA, which, among other things, entirely forbids the use of dynamic heap allocation.

However, C code to handle things made out of structs and arrays with no pointers accommodates any kind of polymorphism only awkwardly, and consequently exhibits very poor generality.

Common Lisp includes generic functions like `concatenate`, `append`, `map`, `remove-if`, `reduce`, `every`, `some`, `length`, `reverse`, `search`, and `equal`; the C++ STL includes generic algorithms like `copy_n`, `swap_ranges`, `nth_element`, and `set_intersection`; Python code has at its disposal generic dictionaries, lists, heaps, and sets.

It would be very desirable to program our embedded systems at a similarly high level of abstraction and generality without paying the heavy memory-efficiency and reliability costs imposed by the Lisp memory model.

The Z-machine object containment tree seems like a possible candidate. Each object contains three pointers: its parent, its first child, and its next sibling. (In the Z-machine itself, these were 16-bit object indices; on the Arduino, 8 bits each would make more sense. Also, it might make more sense for only active cursors into the object tree to carry the parent information, rather than every object.) Rather than destroying or creating objects, including by copying, our algorithms can only move them around and mutate them.

Despite this, most of the Common Lisp generic functions immediately make sense in the context of this rigid object tree. It's just that, unlike most of their Lisp antecedents, they destructively consume their inputs, and in some cases need to be directed as to where to place their output. The sorted set-arithmetic functions in particular, except for symmetric difference, probably reduce to a single `set_filter(comparator, targets, input, output)` subroutine which moves all the objects from `input` that are equal to an object in `targets` into `output`.

The hierarchical structure might prove valuable for persistence support, where it could segregate persistent from nonpersistent objects.

I suspect that this is related to linear or affine typing, as used in Rust, but I don't understand them well enough to really know.

There is a thing wrong with all of my analysis above. It may be only slightly wrong, but enough to make a difference. It doesn't really account for function call stacks.

In the absence of recursive functions (and they should certainly be absent if you want high assurance of no stack overflows) we could certainly statically allocate the activation records of each function, which would ensure no stack overflows. But this is probably much more costly than necessary; functions overlay one another's stack frames in a guaranteed-safe way (modulo uninitialized variables and taking the address of local variables) and give a much tighter memory-usage bound.

Can we extend this property in general?

Topics

Polycaprolactone

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

ShapeLock is actually polycaprolactone, I think, an extremely shatterproof biodegradable, bioimplantable, anti-static (? or is that only Mater-Bi?) thermoplastic with a melting point around 60°C and a relatively low price. It's weak, though, and quite viscous when melted.

Tradenames are Mater-BiZ (a mixture of polycaprolactone and starch), Anderson Andur 6 APLM, Boltaron 4300, and Dow Tone P-767 or P-787. See http://www.ides.com/grades/Boltaron_grades.htm and related pages.

There is some pricing in the RepRap blog comments: http://blog.reprap.org/2005/12/polymorph-and-polycaprolactone.html says "$5.30-3.50/lb for CAPA 6800". It also gives the name "CAPA 6800 polycaprolactone (2-Oxepanone, homopolymer; molecular weight 80,000, CAS number: 24980-41-4)".

The blog post suggests Solvay Interox Ltd. as a supplier. In early 2008, they were sold to the Perstorp Group of Sweden. CAPA was Solvay's brand. Now it is http://www.perstorpcaprolactones.com/; they claim to be a "global supplier".

http://reprap.org/bin/view/Main/Polymorph says Solvay's product cost US$9/kg in 20kg or 500kg bags.

By comparison, Forrest Higgs says:

> Right now you can buy HDPE filament for about $4.50/lb. ABS costs about a dollar per pound more, if I remember correctly. That means that the filament to make the printed parts for a Darwin will cost you about $12.50 while for a Tommelise the cost will be closer to $30.

There's another blog post on the topic at http://createdigitalmusic.com/2006/08/29/prototyping-custom-gear-friendly-plastic-aka-shapelock/ that says it's hard to find the stuff in Australia.

Someone suggested Consilium Designs, an eBay merchant that sells many specialty substances. http://www.mutr.co.uk/ is another UK company that carries essentially the same items.

According to Perstorp's datasheet, their product crystallizes at 25°C, has a glass transition temperature at 60°C, and has 77 joules per gram heat of fusion, more or less. And it supercools. Yield stress is around 16±1MPa, modulus is around 470±30MPa, and strain at break is >700%.

When melted, the higher molecular weight polycaprolactones are 4x as viscous as the lower molecular weight ones, but they all decrease in viscosity by another factor of 6 or so when heated to 150°C.

It's very permeable to CO_2, water, and oxygen.

One person reported that polycaprolactone is better than HDPE for FDM accuracy at great length; it warps less, has less "die swell" (that is, the molten filament isn't much bigger than the extruding hole), sticks to more things, and is "more compliant" (??). But it has a lot of other problems. http://hydraraptor.blogspot.com/2008/03/chalk-and-cheese.html

Topics

http://pad.partidopirata.com.ar/p/sxf3VU4fNT

Stuff I've posted to kragen-tol over the years about post-HTTP

Kragen Javier Sitaker, 2014-02-24 (12 minutes)

I've argued that we need a browser that supports accessing resources stored in decentralized content stores like Tahoe-LAFS or Git, rather than named according to a path defined by the administrator of a particular domain name over time. I haven't done a very good job of gathering my thoughts on the matter into one place yet.

This all seems a little silly, given that other people have contributed so much more to solving this problem than I have, but there may be one or two things in here that aren't currently easy to find, but that are important.

I've only dug through the archives back to 2000. There might be stuff in 1998 or 1999 that's relevant, but surely not very important.

Reasons why

"What's wrong with HTTP?"

In this essay, the first of a pair on browser apps, I explore how they are better than traditional desktop apps in some ways, but worse in others. Some of the disadvantages of browser apps are deeply rooted in the use of HTTP URLs for naming. In the second essay, I will present a design sketch for a new platform, a replacement for HTTP combining both styles' advantages.

I still haven't published the second half, and it needs thorough revision now.

"The equivalent of free software for online services"

Explains why it's crucial to implement communication systems as cooperating software running on users' computers, rather than on centralized servers.

So people use free software because of its guaranteed low cost, because it does what its users want, and because it's trustworthy. And they use web services because they get low system administration costs, they can use huge databases without downloading them first, they can get software updates quickly, they can do very-CPU-intensive things, and they can collaborate with their friends easily. How can we get both of these sets of advantages at once?

I think there is only one solution: build these services as decentralized free-software peer-to-peer applications, pieces of which run on the computers of each user. As long as there's a single point of failure in the system somewhere outside your control, its owner is in a position to deny service to you; such systems are not trustworthy in the way that free software is. ...

So we need a platform, something like a web browser, that supports a universe of constantly-changing code written by a multitude of authors, which migrates to where it's being used, and simultaneously supports individual control over what version of the code is running on your system and no-hassle updating when someone else has a change you want; that replicates your data transparently to other machines so that you don't have a single point of failure, but without allowing the owners of those other machines to spy on you or corrupt your data; that runs programs in a high-level language; that supports conflicting updates to different replicas of the data and allows a human being to resolve the conflicts; and that makes it easy for you to share particular bits of your code or data with anyone, everyone, or no one. Maybe we could even start with a web browser and add the other stuff to it.

If we don't build such a platform, we will eventually lose the advantages of free

software, because we will use web services instead.

There's actually a lot of detail in this essay about specifically how you could structure such a system.

"Why I do not want to work at Google"

Google has an orientation that is opposed to my agenda. ...

I imagine a different future, where if Alice wants to talk to Bob and Bob wants to talk to Alice, there's no unaccountable intermediary that can interfere with their communication, whether they're speaking text, or video, or 3-D models, or simulation. If Alice's email gets marked as spam, Bob ought to be able to find out why — and fix it! I imagine a future where every human being can participate in creating the culture they live in, without needing permission from anybody, and without fearing repercussions. ...

I believe that warehouse-scale client-server computing will, in the end, undermine the kind of democratic freedom of communication that we need to deal with today's global menaces. It's more practical than peer-to-peer computing at the moment, but that pendulum has swung back and forth several times over the decades. (Some of my friends were among the first employees of a hot cloud-computing startup, in 1964, called Tymshare.) The proper response to the current impracticality of decentralized computing is not to sigh and build centralized systems. The proper response is to build the systems to *make decentralized computing practical again*.

Things contributing to building the solution

There's a lot of software out there now that wasn't there when I wrote this stuff; I should probably make a list of it here too. See above, too, the item about "The equivalent of free software for online services."

"Imagine decentralizing Wikipedia with Codeville"

Codeville was an early decentralized version control system, like Git or Mercurial, that didn't take off. Functionally the systems are equivalent: they replicate the entire version history to every user and provide hash-based retrieval.

So support for individual points of view amidst general disagreement is one of the benefits of del.icio.us over dmoz or Yahoo, and it's built into the architecture of the system --- it's not just a social practice. Could Wikipedia's architecture change to support divergent points of view better?..."arch", darcs, monotone, Codeville, git, and other decentralized version-tracking systems aim to support a wider array of development models; in particular, they aim to allow each person's tree to stand alone as a first-class citizen, easily sharing its changes with other similar trees.

Imagine that we applied one of these systems to Wikipedia. We would have several benefits: tolerance of controversy, disconnected operation, higher availability, and potentially organizational decentralization.

We could tolerate controversy better because Holocaust deniers would have their own version of Wikipedia, which they could modify to their heart's content. This would reduce their desire to modify the Wikipedia that everyone else reads, but it would not eliminate it.

"DHTML persistence: a design for a generic Ajax server-side"

Probably the first essay I wrote advocating what we were doing at KnowNow in 2000, which kind of went mainstream in 2004 with Gmail and 2005 with Google Maps: putting all the application logic on the client side in JavaScript, relegating the server to basically being a dumb data store. Once you do this, of course, you no longer need a server as such; you need a dumb data store, which can be provided by

a peer-to-peer network — but this essay doesn't talk about that at all.

I'm not convinced that I actually achieved a usable protocol design here.

In my view, the most sensible thing to do is to write the application entirely in JavaScript from the beginning and run it all on the client side. Doing this prevents you from having to rewrite bits from time to time, and puts the application code on the machine of its user, who can then use bookmarklets and Greasemonkey to customize its functionality.

"Decentralized chat using CouchDB"

I was just chatting with Noah on IRC about how IRC sucks and we need to replace it and whether we could do that using CouchDB.

More broadly, what's needed for decentralized secure chat, which is to say, pub/sub, or event notification. Pub/sub is one of the fundamental services needed for distributed, including decentralized, applications.

CouchDB is one of the current systems that contemplates Lotus-Notes-style mobile-code secure applications, which it calls CouchApps. Unfortunately, I think the discussion that followed this email showed that it's not capable of providing the kind of support for secure collaboration that we need — its security model is too simple.

"distributed posting list joins"

One of the hardest problems in decentralized systems is how to query a decentralized database with acceptable efficiency. In this post, I finally found a solution that allows you to build a *distributed* if not decentralized full-text web search engine. This followed some work in http://lists.canonical.org/pipermail/kragen-tol/2004-February/threaod.html.

"rumor-oriented programming"

Suppose we want to build a distributed application with automatic change synchronization. Here's a persistence system with coordination functions somewhat similar to mod_pubsub or Linda, but specifically designed for replicating the state of an application.

I actually wrote some things kind of like this, but never built the full system described. In fact I never finished describing it :(But it's kind of like CouchDB or Meteor.

"Peer-to-peer overlay networks are a bad idea on a DSL-based internet."

Peer-to-peer overlay networks are inefficient on ADSL networks. ADSL networks are almost twice as efficient as SDSL networks. Better alternatives require redesigning the physical layer.

"mailing lists, blog posts, and Git: what to do next with kragen-tol?"

Lamenting that neither Git nor the Web provide distributed autentication of publication date, which is a thing I want for kragen-tol, which is why kragen-tol is still a mailing list.

"web services, operations as a strategic advantage, and decentralization"

Suggesting that if we can decentralize web services onto individual

users' machines, then maybe we'll be able to reduce deployment headaches. In retrospect, I think this is kind of a dead end — instead we have devops — but it contains the concept.

Just because the software runs on its users' machines doesn't mean it can't be providing a networked service; consider BitTorrent or Skype or, for that matter, Sendmail, ircd, or INN.

"the end-to-end principle in human society: scholarly writing and freedom of speech"

Describes web browsers as "mere conduits" for information; suggests content-centric networking.

"offline web reading"

Nothing earthshaking but does mention I was able to use Google Maps offline with WWWOFFLE because of its RESTian architecture.

"lazy evaluation in a distributed system"

Some notes on how to build an event-notification/pub-sub/cache-invalidation system that supports decentralized operation --- for changeable resources that live at an identifiable network node.

"level-triggered 'event notification': condition notification"

More notes on event-notification and pub-sub systems.

"P2P resource discovery"

I suggest storing current physical location information for mobile P2P nodes in a DHT, so that you can route packets to them. Really, that's it; you don't need to read the post.

"distributed mailserver"

How to build a fault-tolerant distributed SMTP/IMAP server, supporting mailing lists (pub-sub!) using distributed transactions.

"DWOF"

Earlier, sketchier notes on how to build a distributed mailing list server.

Topics

- Systems architecture (p. 3691) (48 notes)
- Protocols (p. 3668) (21 notes)
- Decentralization (p. 3404) (13 notes)
- REpresentational State Transfer (p. 3684) (8 notes)
- HTTP (p. 3509) (4 notes)

Polynomial-spline FIR kernels by integrating sparse kernels

Kragen Javier Sitaker, 2014-04-24 (12 minutes)

I think I have a method for reducing the computational expense of a large and interesting class of FIR filters by an order of magnitude or so, but I haven't tried it out yet, and it seems like the kind of thing people would have tried by now, so it probably either won't work or is already known.

In my case, this questionable insight came out of, among other things, considering how to write timesheet software using functional reactive programming, sweating through the night in the Buenos Aires heat wave and blackouts, and considering whether it's possible to approximate dense FIR kernels by convolving multiple sparse FIR kernels together.

I've tried to write this with some humor, although I think the result is basically that I sound insane. Hopefully that provides some entertainment. Don't take it too seriously.

Background

(This section is basically me regurgitating shit from Wikipedia and dspguide.com, so feel free to skip it if you know DSP. Or, better yet, read it and correct me.)

FIR filters are common tools in DSP because they can easily be designed to get zero-phase high-performance filters: you take the inverse FFT of your desired frequency response, giving you the impulse response, which you window to give it compact support without fucking up the frequency response too much, and that gives you the weights for your filter. (Not the only method, but a common one.)

However, in many cases, you end up needing tens, hundreds, or even thousands of multiply-accumulates per sample. As a result, we often use IIR filters to get better efficiency, even though we don't have a general theory of how to design IIR filters.

The moving-average filter

The moving-average filter is a special case of a FIR filter: all the weights within its support window are equal, so its impulse response is a pulse, like one cycle of a square wave. It's used for a couple of reasons: (1) it's optimal in minimally degrading time-domain step response while rejecting high-frequency noise, and (2) it's highly efficient.

If you implement it in the general FIR fashion --- multiplying each of the last N input samples by a weight, then adding them --- it's not any more efficient than any other FIR filter with the same support. But you can implement it much more efficiently than that, because composition of time-invariant linear operators is commutative.

Specifically, the impulse response of the moving average filter is the integral (or prefix sum) of a very sparse kernel: a single positive impulse at the beginning of the window, and a single equal negative impulse at the end. Integration (or prefix sum) is a time-invariant

linear operator, as is convolution with a given kernel. So what you do is that you first convolve the input signal with the derivative (or finite forward difference) of the desired impulse response, which requires only two multiply-accumulates per sample, since that derivative is so sparse; then you integrate (or prefix-sum) the result of the convolution; and the commutative property guarantees you that the result is the same. So you get by with two MACs and an addition per sample.

(Actually, because the two impulses are equal in magnitude, you can wait until the very end of the process to multiply by the weight, giving you three additions/subtractions and a single multiply per sample. Sweet.)

Polynomial splines

A spline is when you approximate a function by breaking it up into chunks along the X-axis and approximate each chunk with a different polynomial. Typically the polynomials produce the same value at the points where they join up, the "knots", which is to say, the spline is at least continuous, and usually has a continuous derivative too. (Normally you have the constraint that a spline made out of Nth-order polynomials has continuous derivatives up to order N - 1.)

Typically you can get a pretty good approximation of an analytic function with second- or third-degree polynomials, and without all that many of them. This is a lot less computation than using a high-order approximating polynomial, and as a scrumptious bonus, it also avoids Runge's phenomenon. So this is actually the approach used by a lot of math libraries these days.

(This suggests that the Difference Engine could have been quite a bit smaller if equipped with a facility to reload the highest-order difference from a table periodically, at knots; it could perhaps have used three or four columns instead of eight.)

Efficiently approximating FIR kernels with splines

The standard efficient method for computing the moving average filter can be generalized to a wide class of FIR kernels, producing a worse but still very substantial computational speedup.

Step function OTFs

A moving average filter is of the family of step functions: piecewise constant functions. That's why its derivative (or forward difference) has such sparse support.

But you could in principle compute a convolution with any piecewise constant OTF by convolving with the derivative and then integrating.

Step functions and splines

If you integrate a piecewise-constant function, you get a continuous piecewise-linear function; if you integrate that, you get a continuous piecewise-quadratic function with a continuous derivative; and if you integrate that, you get a piecewise-cubic function with continuous zeroth, first, and second derivatives. Which is to say, you get a polynomial spline.

If it happens that your desired impulse response can be

approximated by an Nth-order spline with a small number of points, then you can fit that spline to it; take the N+1th forward difference to get a sparse kernel; apply that sparse kernel to your input data; and integrate its output N+1 times to get the filtered output.

For each sample, this requires one multiply-accumulate per spline knot plus N+1 additions for an Nth-order spline.

Desired frequency response containing no low frequencies

This section is incomplete.

This is going to suck if your desired impulse response is very wiggly, though, because I think that in general (certainly with a second-order spline), you'll need two knots for each oscillation. If the wavelength of your oscillations is only like six or eight samples, then you're not going to be saving much.

I think, based on basically no experience, this kind of wiggliness often shows up because you're trying to put together a high-pass or bandpass filter, and so all your low frequencies are zero. Which is to say, your desired frequency response is the convolution of an impulse at some high "base frequency" with some window function giving the shape of the frequency response above that frequency.

If you shift the window function down closer to zero, you'll reduce the wiggliness a lot, and you'll be able to get by with a lot less knots.

So what good is that? You'd have to somehow shift the signal down in frequency, and then back up. RF circuitry does this kind of downconversion all the time by just multiplying the high-frequency signal by a reference signal at a nearby frequency, thus producing sum and difference frequencies[1], typically much lower (either near zero, "baseband", or in some lower but still RF band, "intermediate frequency"). This is easy enough to do (it requires one multiply per sample, not even a multiply-accumulate). But I have the impression that this kind of downconversion and upconversion requires high-pass or bandpass filtering before and after, which seems to be begging the question, so I suspect this may be making the problem harder rather than easier. (Maybe you can use an inverted low-pass filter for your high-pass filtering? That shouldn't run into the wiggliness problem, although you want to do the inversion separately from the spline approximation.)

[1] If this is puzzling, remember the sum and difference identities from trig class. Or re-derive them from Euler's Formula.

Anyway, if you somehow manage to frequency-shift your input signal down to baseband, then you can use a much less wiggly filter kernel on it, and then multiply it by a carrier wave to upconvert it back to its original frequency.

Another incomplete approach for kernels containing no low frequencies

Another possibly valid approach for wiggly kernels: note that N cycles of a sine wave are the convolution of a comb filter with N impulses spaced one cycle apart with a kernel consisting of a single cycle of the sine wave. You can thus convolve a kernel consisting of N cycles of a sine wave with your signal (as if that were a useful thing to do, but bear with me) by first convolving it with the comb filter, then with the single cycle, for an almost N-fold reduction in

computation if the number of samples per cycle is large, or an almost samples-per-cycle-fold reduction if N is large.

If you want instead to shape the sine wave with some kind of envelope, you could make each impulse in the comb a different height, but that is going to give you discontinuous derivatives and totally fuck up your frequency response; those discontinuous derivatives are almost like impulses in their wideband-noise nature. You can do better by making your second kernel be, rather than a *single* cycle, *two* cycles, but windowed with a triangular window. This way, in between the comb points, you're linearly interpolating between two sine waves with different amplitudes.

Getting back to the wiggly kernel, I think you can approximate it as such an amplitude-modulated sine wave, with the frequency of the sine being the "base frequency" I mentioned earlier.

So how does this save you work over the spline approach? Well, I think you can use the spline approach to do the N-points modulated comb filter in a lot less than O(N) work by round-robining among a bunch of different intermediate signal buffers, one for each cycle per sample. But I haven't worked out the details yet, so I'm not sure it will work.

Related work

I wrote the above while my internet connection was off, so I couldn't search to see if anybody had already done this.

B-spline image interpolation might appear to be related, but it is different. This is a common algorithm for image interpolation (upsampling, resolution enhancement) aka cubic B-spline interpolation, where you construct a bicubic B-spline patch between adjacent pixels in order to approximate the values between the pixels. It turns out that you can do this by applying a FIR filter to the image; that is to say, the interpolated pixels of the upsampled image are a linear function of the neighborhood pixels of the original image, even though the splines are nonlinear. This, approximating the ideal image with a spline, is a very different concept from approximating the FIR kernel itself with a spline.

"Kernel B-Splines and Interpolation", by Bozzini, Lennarduzzi, and Schaback, 2005, appears to be about a variant of the image interpolation problem (interpolating an unknown function between some set of known points).

There's a substantial body of theory interpreting sampled signals as ways of representing splines rather than bandlimited sums of sinusoids; the methods mentioned above belong to that theory. The approach I'm discussing here does not, I think, belong to that family of methods, but I'm not very familiar with the theory of spline image processing. So maybe it's in there somewhere.

Topics

- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Splines (p. 3727) (6 notes)

10tcl ui

Kragen Javier Sitaker, 2019-12-06 (17 minutes)

I was talking with the entity known as The Doctor. They mentioned that they really liked the idea of having a tiny programming language in your boot ROM, but had found the Forth in OpenBoot to be fiddly, requiring a lot of effort to do simple things like changing the MAC of a Sun NIC.

Usually what I've done with the boot ROM is to boot, though, maybe occasionally from alternative boot media. I found that to be kind of fiddly in OpenBoot too. (I also wrote a graphics demo in OpenBoot on an OLPC, but because I couldn't figure out how to abort back to the interpreter, my first infinite loop ended the experiment†.)

Tcl

This reminded me of Yossi Kreinin's wonderful 2008 essay I can't believe I'm praising Tcl, about why Tcl is a terrible programming language but a good command language:

> Well, in Tcl that's as simple as it gets. Tcl likes to generate and evaluate command strings. More generally, Tcl likes strings. In fact, it likes them more than anything else. In normal programming languages, things are variables and expressions by default. In Tcl, they are strings. `abc` isn't a variable reference – it's a string. `$abc` is the variable reference. `a+b/c` isn't an expression – it's a string. `[expr $a+$b/$c]` is the expression. Could you believe that? `[expr $a+$b/$c]`. Isn't that ridiculous? ...
>
> And the nice thing about my retarded debugger front-end is that it looks like shell commands: `blah blah blah`. As opposed to `blah("blah","blah")`. And this, folks, is what I think Tcl, being a tool command language, gets right. ...
>
> And eventually, simple commands become all that matters for the [interactive] user, because the sophisticated user grows personal shortcuts, which abstract away variables and expressions, so you end up with `pmem 0 bkpt nextpc` or something. *Apparently, flat function calls with literal arguments is what interactive program usage is all about.* ...
>
> I have a bug, I'm frigging anxious, I gotta GO GO GO as fast as I can to find out what it is already, and you think now is the time to type parens, commas and quotation marks?! Fuck you! By which I mean to say, short code is important, short commands are a must.

Tcl is also, it bears mentioning, quite a bit saner than bash as a programming language.

In Tcl, `foo -x $bar -y $baz` invokes `foo` with four arguments: the string `-x`, the value of `bar`, the string `-y`, and the value of `baz`. The equivalent command in bash sometimes does this and sometimes calls `foo` with some other random number of arguments, depending on what is in `bar` and `baz`.

In Tcl, if your code has an error, your program will stop there with a backtrace (possibly popped up in a Tk window, if you're into that kind of thing), while in bash, it will probably continue as if nothing untoward had happened, unless you're running with `set -e`, which will crash perfectly working programs with some older versions of sh, because sh commands indicate failure in the same way that sh boolean expressions indicate falsehood: by returning nonzero. And not only won't `set -e` give you no backtrace, it will give you no error message at all --- you may not notice that anything has gone wrong.

And Tcl supports not only lists, which bash sort of does, but also nested lists and associative arrays ("dicts" or "hashes"), although like

Perl 4 and Awk, the associative arrays are not first-class.

(Tcl's nested lists are kind of shoddy, but they do exist. An unusual thing about them that they sort of share with bash lists is that a list of one string is the same as that string; a string is a single-item list of itself, in the same way that Python uses single-character strings to represent characters, or Octave uses 1x1 matrices to represent scalar numbers. So you can't distinguish between the string `squizz` and a list of one item that contains a list of one item containing the string `squizz` .)

Also, in Tcl, you can define subroutines that return values, including nested lists. In bash instead your subroutines can just write bytes to their output channel, which by default displays the result on the terminal.

Like Bash, Tcl unfortunately doesn't support named arguments or any other kind of name-value-pair interface that would make it reasonable to preserve backward compatibility, except by using some ad-hoc command-line parsing like Tk does:

```
button .x -text foo -command {puts bar}
```

One way bash usability beats the *shit* out of Tcl usability, though, is in tab completion, which in modern bash setups is fairly context-sensitive, understanding the syntax of most commands well enough to complete the appropriate kind of thing for the context you're in, most of the time; so, for example, `sudo` will tab-complete to available commands, `sudo apt` tab-completes to the 11 `apt` commands, and `sudo apt install` tab-completes to the available `apt` packages (!).

This is a huge timesaver; modern IDEs implement something similar, with dropdowns, by using static typing information which isn't present in Tcl, or for that matter bash. IPython does it without static typing information but only for properties of a variable that's already defined. Fecebutt and Slack provide dropdowns with substring search when you start to @mention somebody in a text box.

Tab completion is extremely useful for navigating hierarchies; for example, when you're booting, there's often a hierarchy of possible boot media --- USB vs. hard disk vs. network, different USB mass storage devices, different partitions on the mass storage device, different files in the filesystem, etc. Or, when you're debugging, you may have nested structs, or arrays of structs, or linked structs, that you maybe want to examine pieces of; or you might have nested dicts ("objects" in JS) with arbitrary sets of names. In Tk you have a hierarchy of widgets. (Though I think IMGUI is probably a better design than Tk on modern fast machines; see IMGUI programming compared to Tcl/Tk (p. 2333) for thoughts on that.)

Noun-verb ordering rather than Tcl's verb-noun?

Verbs act on nouns. Yossi's `pmem 0 bkpt nextpc` is a verb `pmem`, print memory, that acts on CPU 0 and, I think, prints out its breakpoints and next program counter value. `mv Un_Yanqui_Enseñando_Dichos_Argentinos_a_otro_Yanqui-hBhrHaoYONs.mp4 humor/.` is a verb `mv`, move, that acts on a video named by the first argument,

moving it into a named directory.

Verbs and nouns have some type-compatibility requirements, so if you know one of them, you can narrow down the candidates for the other, making it easier to choose. `mv` above only acts on filenames, and if it's given more than two arguments its destination argument needs to be a directory, while `mpv` only acts on *some* filenames, those that name directories or video or audio. `pmem` presumably only acts on CPUs for its first argument.

If you have more verbs than nouns, tab completion is probably easier if you specify the noun, or *a* noun, first. If you have more nouns than verbs, tab completion is probably easier if you specify the verb first. So, for example, my /usr/bin directory has 4652 verbs in it, most of which are only applicable to a few kinds of nouns, but none of my own directories have that many nouns in them. `tiff2bw` is only applicable to TIFFs, `pdftotext` is only applicable to PDFs, and `avrdude` is only applicable to AVR flash memory images. So probably a noun-verb order would be more usable.

In either case, there's the possibility of needing a space-filler of whichever syntactic category comes first. If I don't want to do any particular operation on some file, just see it, I still need to invoke Unix `cat` or `less` or `xdg-open` or `ls`. Similarly, in Smalltalk, verbs that don't really operate on any object have to get arbitrarily attached to some class as class methods.

If you have a lot of nouns --- or for that matter verbs --- you probably want a better way of choosing one than reading through a list of all of them or typing a memorized name. As alluded to above, a hierarchy is one possibility, while search (for example, using substrings, as in Fecebutt, or arbitrary database queries) is another. Bash tab-completion does a prefix search which doubles as hierarchy navigation. But bash tab-completion is copied from csh and tcsh, which come from a time when avoiding process context switches was an important consideration (so, for example, csh completions were triggered with ^D!) and terminals were commonly 2400 baud --- 3 lines of text per second. Most modern computer systems have much larger display bandwidth, often in the gigabits per second, and can thus afford to be more proactive about presenting candidates.

Most OO and OO-influenced programming languages put object properties (forming, sometimes, a hierarchy) and operations (verbs) in the same namespace; Smalltalk uses the same syntax for `anArray size`, which returns a number, and `anArray inspect`, which opens an inspector window; or for `anArray at: 3` and `anArray at: 3 put: 4`; and, in Python or JS, `foo.bar` can be either the instance variable `bar` or a method `bar` of the object `foo`, although they do distinguish between merely *reading* the property and *invoking* an arbitrary operation.

Making such a distinction, like the HTTP distinction between GET and POST, is crucially important for tab completion: if evaluating `foo.bar` can cause serious side effects, you don't want the UI to do it peremptorily. But, of the things that *can* be evaluated without serious side effects, you would like to maximize the number that the UI can look at peremptorily.

Another usability-maximizing design feature (missing from Tcl! but not from bash!) is being able to write to things in a way consistent with reading them. That is, if reading `foo.bar.baz` gives you 3, it is often useful to define `foo.bar.baz <- 3` or something similar as a way to

establish the same state of affairs in the future. It's much worse for usability to have to wander around looking for an operation on `foo.bar` or perhaps even `foo` that has the effect of changing `baz`.

An excellent example of the utility of writability and hierarchies was The Doctor's original example; under Linux:

```
$ cat /sys/devices/pci0000:00/0000:00:1c.0/0000:01:00.0/ieee80211/phy0/macaddress
00:24:2c:97:d8:58
```

Sadly, this kernel does not provide the ability to change the MAC address by writing to that pseudo-file.

Most of Yossi's hardware-debugging example from before could quite reasonably be implemented as a mere object graph like this, with custom getters performing the operations he'd decided were safely side-effect-free. (He mentions in the article that reading memory-mapped I/O regions wasn't always safe, which is a pretty common situation in device drivers.) Occasionally you'd maybe need to apply a verb to it. Here's a fragment of a recent GDB session as I was tracking down a bug, a fragment which consisted almost entirely of such navigation:

```
(gdb) p symbol
$1 = (HCFChoice *) 0x1e
(gdb) frame 1
#1  0xb79ecacd in collect_nts (grammar=0x83e92e0, symbol=0x83e9258)
    at build/debug/src/cfgrammar.c:121
121         collect_nts(grammar, *x);
(gdb) p symbol
$2 = (HCFChoice *) 0x83e9258
(gdb) p *symbol
$3 = {type = HCF_CHOICE, {charset = 0x83e92b0, seq = 0x83e92b0,
    chr = 176 '\260'}, reshape = 0xb79f0202 <h_act_first>, action = 0x0,
  pred = 0x0, user_data = 0x4c3b433d}
(gdb) p *symbol->seq
$4 = (HCFSequence *) 0x83e92c0
(gdb) p **symbol->seq
$5 = {items = 0x83e92d0}
(gdb) p (*symbol->seq)->items
$6 = (HCFChoice **) 0x83e92d0
(gdb) p (*symbol->seq)->items@10
$7 = {0x83e92d0, 0xb7f89450 <main_arena+48>, 0x3c, 0x11, 0x1e, 0x0, 0x41,
  0x29, 0x0, 0x83e431c}
```

You could imagine this whole transcript collapsing down to `stack.1.symbol.dest.seq.dest.dest.items dump 10`, or even `stack.1.symbol.seq.0.items d 10`.

Similarly, most of the booting I do could be accomplished by navigating through a device tree and finally applying a "boot" verb.

A thing that is mentioned in this transcript, but missing from my commands and from bash, Forth, and Tcl, is GDB's ability to refer back to previous results; for example, instead of `p **symbol->seq`, I could have written `p *$4`.

This kind of navigation does not entirely replace the need to pass string --- or other! --- parameters to verbs.

GUIs

Noun-then-verb interaction is of course entirely standard in GUIs; even SKETCHPAD had you select onscreen objects with the light pen before applying actions to them by flipping switches. It's still uncommon in command-line interfaces.

10tcl

Originally I was thinking of something fairly traditional: a simple Lisp dialect, but with more Tclish syntax, in which symbols and lists are quoted by default and require some kind of explicit sigil to unquote them --- perhaps "," rather than Tcl's "$" --- and in which the outermost parentheses are unnecessary. And called "10tcl" as in "tentacle". And maybe with dicts, like Clojure. But then I started thinking about how to handle tab completion, and the noun-then-verb thing popped up, and the RESTish distinction between properties and verb invocations.

This suggests a connection with Darius Bacon's language Cant, a dialect of Scheme in which the basic procedure-definition system includes a pattern-matching system, so that it is easy to define a procedure which returns `#no` if invoked with the argument `.interactive?` and a different form if invoked with two arguments the first of which is `.pick-move`:

```
(make greedy-player
  (to ~.interactive? #no)
  (to (~ .pick-move board)
    (for min-by ((move board.gen-legal-moves))
      (greedy-evaluate (update move board)))))
```

Specifically, you could imagine that invoking `sys.class.block.sda1 boot` would invoke the procedure denoted by `sys.class.block.sda1` with the symbol `boot` as its single argument. This is the same as Tcl as far as it goes, except that `sys.class.block.sda1` is actually an expression interpreted as it would be in Python or JS: as a series of property accesses. But a facility for defining actors like the Cant `greedy-player` above would make it convenient and idiomatic to define entities that responded to such invocations.

However, a significant difference is that these 10tcl objects *additionally* have properties which can be, by convention, safely enumerated and read; they might be a statically determined set or a set computed by arbitrary code invoked at runtime, like Python `__dir__`, `__getattr__`, and `__getattribute__`. That is, like JS or Python functions, 10tcl objects can have *both* behavior *and* attributes.

As in Tcl, the first word of the command is evaluated under different rules than the rest of it: namely, the first word is *evaluated* (rather than used to look up a proc, as in Tcl), while if there is more to the command, it is *quasiquoted*.

The grammar might look something like this:

```
command ::= expr (hwsp quasiquoted)* newline
hwsp ::= (' ' | '\t')*
expr ::= '(' command ')' | name | expr '.' name | number | obj
quasiquoted ::= '(' command  ')' | name | number | ',' expr | obj
```

Here "obj" is intended to represent things like lists and dictionaries, whose syntax I haven't thought about yet.

I think the Common Lisp, Forth, and Scheme approach of defining new control structures through compile-time metaprogramming is probably better than the Tcl and MACLISP approach of defining them through fexprs, partly because you can inspect the results of the compile-time metaprograming more easily.

Footnote

† It turns out that to interrupt an infinite loop in OLPC Open Firmware/OpenBoot, the answer is that the DEL key or the key with a rectangle on it in the upper right will abort, although after that you have to type `enable-interrupts` to run it again, except on later models of the XO.

Topics

- Programming (p. 3658) (286 notes)
- Independence (p. 3520) (63 notes)
- Programming languages (p. 3656) (47 notes)
- Small is beautiful (p. 3714) (40 notes)
- Forth (p. 3461) (19 notes)
- Tcl

Could you do DDS of comprehensible radio signals with an Arduino?

Kragen Javier Sitaker, 2017-03-31 (4 minutes)

Could you do DDS of comprehensible radio signals with an Arduino? Maybe with some filtering.

You can spit out data words with bitstreams at the full clock rate with the SPI controller (and some trickery), which is a square wave of up to 8MHz. You can get 10MHz (20mbps) if you use a 20MHz crystal.

AM radio is 525 kHz to 1705 kHz with like a 10kHz bandwidth. FM radio is 88 to 108 MHz with like a 200kHz bandwidth and ±75 kHz deviation.

The mechanisms for these two bands would be quite different.

For AM, we would be synthesizing more or less the frequency we actually want, but with different amplitudes (and unwanted harmonics). A 530kHz cycle is 1.89 µs, about 30 bit times. That should give us something like 10 or 15 distinct amplitudes, maybe 4 bits. Not as good as magnetic tape, but probably enough for comprehensibility.

For FM, we can't hope to get close to the actual carrier frequency; instead, we can hope that our square waves are square enough and well-timed enough to have harmonics up there. We may be able to improve the situation by reshaping them (say, with a Schmitt trigger) to sharpen the edges. At 8MHz, though, you can't get very narrow variation of frequencies. At 4MHz, your 23rd and 25th harmonics would be within the FM band. But I'm not convinced you can do an adequate job unless you're actually bending the Arduino's own clock.

Bending the clock might not be that hard, though, especially if we're running on the internal 8MHz RC oscillator for a change. Keep in mind we're looking for a deviation of less than 75kHz out of about 100MHz: 0.075%. And it doesn't even have to be very linear. It just has to be roughly controllable.

(The numbers hereafter are from the ATtiny2313 datasheet.)

The OSCCAL register on the ATtinies can adjust the internal oscillator speed, but only by increments of about 1% (about 2% at the high end of the frequency range) which would take you to an entirely different radio station. However, the oscillator frequency also varies with Vcc in a much more subtle fashion. So maybe by putting a resistor on the Vcc lead we could vary the Vcc between, say, 1.8V and 5V, by varying the current drawn by the chip and therefore the voltage across the resistance. The voltage coefficient of frequency varies on both sides of zero over the voltage range, but it seems to be usably large in many places. For example, going from 2V to 2.5V, eyeballing the chart, it looks like the frequency at 25° declines from about 8.02MHz to about 7.99MHz, a 30kHz variation (0.375%), about 60Hz/mV (0.00075%/mV). This means that by varying the voltage by 100mV we can vary the frequency over the appropriate range.

Active supply current Icc increases with Vcc; over that range, it

varies from 1.3 mA at 2V to 1.8 mA at 2.5V, supposedly. Idle supply current is about 0.3 mA to 0.5 mA over that range, so you can generate a proportionately large variation in current just by executing instructions. If we want 1mA to vary our current by 100mV, then we want about 100Ω of resistance.

We can't be sourcing 20mA of FM signal, because the resulting current and thus voltage and thus frequency variation would overwhelm the signal we are trying to generate. We might be able to sink it, though, with the output load being connected between an output pin and Vcc.

Scott Harden reports that he is able to get good results transmitting 1MHz square waves with a class-C amplifier consisting of a 2N7000, a resistor to Vcc, a couple of coupling capacitors, and modulating the signal into the transistor's drain below the resistor with an audio signal. But he was just generating square waves. If I understand it right, this seems like it might be a useful alternative to direct digital synthesis, maybe using a filtered PWM output.

Topics

Better be weird

Kragen Javier Sitaker, 2019-06-17 (updated 2019-06-24) (9 minutes)

(Revised from some comments I posted on the orange website.)

Why is it crucially important, as Feynman thought, to disregard others' opinions in order to be productive? In his case, part of the problem was that he was suffering from a fear of failure due to others' high expectations of him. He was so worried about looking foolish that he couldn't play with new ideas in the way that made him productive. But there's a deeper and broader reason, one that goes far beyond performance anxiety.

In fields like art, programming, and physics, you'd *better* be doing something *weird*. If you're doing something *mainstream*, the same thing thirty other people around the world are doing, you're all competing to make the same thing — paint the same painting, write the same text editor, prove the same theorem about black holes. Twenty-nine of you are going to get scooped by whoever is the hardest-working, the smartest, the best-supported institutionally, or whatever combination of those turns out to be the deciding factor. Your chance of being in that 97% who have totally wasted their efforts? 97%.

And if you're doing something *really* mainstream, like writing the next big client-side JavaScript framework, the one that will replace React, watch out! Your chances are a lot worse than that, because there are *hundreds of thousands* of people who fight with React every day and are frustrated with its shortcomings. Your chances are literally millions to one, unless you work at Microsoft or Google and have management support to beat those Facebook fuckers.

But if you're doing something offbeat, working on a problem† that's mainstream enough to be interesting if you're successful but not mainstream enough that dozens of people are already spending their weekends trying to solve it, you have a much better chance of finding a niche for your project. Maybe it's non-mainstream because people take for granted that it can't be solved (in which case they might be right, as with the reactionless drive — the objective here is to be weird in your *project goal*, not your *epistemology*); maybe because they don't understand why it would be important to solve it ("Where's the market?"), and you do; maybe, as with OpenSSL, it's an important problem, but there's no way to get paid for solving it.

A thousand hackers writing a thousand versions of the same library in the same way are only epsilon more productive than one hacker. A thousand hackers writing a thousand different libraries are almost a thousand times as productive.

Winning the lottery? Well, that's pretty much out of your control — but if you do decide to waste your money on the lottery, don't pick a number lots of other people are picking. Then you'll have to split the already-improbable winnings N ways.

But what if you're determined to solve a mainstream problem anyway, one where a thousand people are also trying to solve it? Then you need all the *outcome* variance you can get! If, of those thousand hackers, 500 are using a very conservative approach that is guaranteed to solve the problem with some quality metric 10 ±1 in 26

weeks ±2 weeks (these being the standard deviations, not some kind of 95% confidence interval), while the other 500 are using all kinds of wild approaches that give them quality metric exp(ln(4)±ln(4)) in time exp(ln(52)±ln(4)) weeks, it's pretty much guaranteed that the "winner" is going to be someone with a totally insane approach that hacked together a library with quality 49 in only 14 weeks. It's not going to be one of the 26-week plodders, because 14 weeks is 12 standard deviations out on their distribution (vs. 0.95 out on the crazy hackers' distribution), and quality 49 is 39 standard deviations out (vs. 1.3 out on the crazy hackers' distribution).

In fact, it's even worthwhile to sacrifice *expectation* to get higher *variance* in these situations. If your only hope of winning is to beat everyone else in a single round, you should do whatever will increase your minuscule chance of a home run, regardless of how it affects your chances of striking out.

It's still not socially optimal for people to behave this way — note that here you have 1000 hackers whose aggregate productivity is only about 20× the productivity of an average plodder — but if you've gotten suckered into competing for a mainstream niche, that's the way to play the game.

All of the above is for the simplified situation where you're working on a project by yourself. In a teamwork situation, the relevant actor is your team, not you individually. Do not write your code in Clojure if the rest of the team is working in Ruby. Do not try to solve only problems that nobody else on the team thinks are important.

And this advice definitely does not apply to a situation where doing the same thing someone else already did is valuable. If you're making a sandwich, there's no reason it needs to be different from the sandwich someone else is making across the street. They're two different sandwiches. If someone eats the sandwich across the street, it's gone and it can't feed your customer. They're going to be happy if you make them a sandwich they like, even if it's a little worse than the sandwich across the stret; even if there are many just like it, this sandwich is theirs. This is very different from the situation in software, where one sandwich feeds everybody in the world at once, except people with celiac disease. Nobody is going to be happy that you wrote a web browser from scratch for them instead of just installing Firefox. Web browsers are winner-take-all. Sandwiches aren't.

You might think that after you work through a few iterations, the distribution would start to approach a standard normal distribution, and the mean would start to matter more. But this isn't the case as often as you might think. If you're hacking on free software, as you should be, then after the first iteration, everyone can start using the clearly-much-better thing that is already working, rather than wasting months finishing their own inferior versions. And the person who wins the race that time around may not be the same one who won the first time.

The other thing is that there are distributions like the Cauchy distribution that are so heavy-tailed that they don't even have a mean, or even a variance. The Law of Large Numbers doesn't apply to them at all! And even for more ordinary heavy-tailed distributions that do have means, like the lognormal distribution (relevant here

since it's empirically the distribution of how much we misestimate tasks by) the Law of Large Numbers takes a lot more than "a few times" to start making the distribution of the sum look normal. Try it! Pop open Jupyter and convolve the lognormal distribution with itself a few times! How long does it take before it even starts to look Gaussian? How long does it take before it still looks Gaussian in a log-linear plot?

On the other hand, if you took my first piece of advice and you're working on something sufficiently weird, you no longer need to worry about increasing your variance further to beat the pack. There is no pack. Instead, anything you achieve will be a positive contribution; so, instead of grasping at straws to avert the almost-certain failure of competitors in a winner-take-all game, try to maximize your *expectation* of performance. That might mean increasing variance or it might mean decreasing variance, and it might depend on your utility function as well as the objective probabilities.

† I recognize that a painting is not "solving" a "problem", but many of the same principles apply.

Topics

Macroscopic capacitive DLP

Kragen Javier Sitaker, 2019-04-08 (1 minute)

I was writing in Paper/foil relays (p. 3273) about macroscopic electrostatic relays made from paper and graphite. Metal foil is another potential material for such devices, which deflect an insulator bearing conductive contacts using the electrostatic force on a conductive plate mounted on the insulator. Metal foil can also be mirror-reflective, which means that by deflecting it in this way, you can also redirect light. Indeed, as described in Caustic business card (p. 255), even submicron deflection of the foil surface can be sufficient to produce visible reflection effects nearby. You could produce such deflections fairly quickly and with fairly low voltages.

Topics

Parametric polymorphism and columns

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

In a relentlessly monomorphic language with good type safety, you could imagine reducing the size of object pointers to the size needed to distinguish the members of their class. If there are never more than 16 Rectangle objects, for example, you don't need more than 4 bits to identify a Rectangle; you can store their xmin, ymin, xmax, and ymax attributes in arrays of size 16. This is actually a practical thing to do in Verilog, where you actually can have a 9-bit variable (as opposed to a 16-bit one).

Now, maybe your Rectangle object is instead actually made of Point objects ul and lr. If you want to pass the ids of those Point objects to Point functions, you have two options:

• The Fortran option: make the x and y attribute arrays of the Point class explicit parameters to the Point function.
• The Smalltalk option: store all the Point attributes in the usual Point attribute arrays, then put the ids of the Points in question into ul and lr.

So far so good, although in #2 maybe you are spending more space on the Point pointers than on the Rectangle pointers.

Okay, now here's a thing that bothers me. What do I do if I want parametric polymorphism? Consider the case of an 'a list made out of car and cdr attributes, where the car has type 'a and the cdr has type 'a list, and where maybe we use -1 or something for a null cdr. Can I write a polymorphic list length function?

I have somewhat corresponding options.

If I have a Rectangle list type, for example, its car array can be of 4-bit Rectangle ids, in an alternative analogous to #1. But the length function doesn't actually use that array at all; it only needs the corresponding cdr array. So that works fine. In an alternative analogous to #2, I store all types of 'a list in the same car and cdr array, so the car array needs to be wide enough to accommodate pointers to any object type.

Topics

Agenda hypertext

Kragen Javier Sitaker, 2018-07-14 (updated 2018-07-15) (2 minutes)

Lotus Agenda was an interesting program that you could use for organizing research notes, tracking expenses, scheduling appointments, and the like. It gave birth to the "Personal Information Management" or "PIM" category of software, but it's more flexible than the other programs in that category.

An unusual thing about Agenda is that it mostly managed text, but at an intermediate granularity, roughly the granularity of a sentence. Each "item" was categorized into some set of "categories", which were like tags, but were mostly implicitly applied based on matching words in the text. The item-category association could have an associated value, sort of like a spreadsheet cell, with items being the rows and categories being the columns.

You could have several different views of your items, selecting which categories you wanted to display as columns, which you wanted to use to divide the view into sections, and which you wanted to add section totals to.

Web browsers allow you to navigate and organize text at the level of pages, while word processors and text editors allow you to navigate and organize text at the level of letters. Agenda was born about the time of Hypercard, long before hypertext became popular, but I've been thinking it would be interesting to have a hypertext system that worked at the intermediate level where Agenda did.

The key questions here are:

- Do links lead to views or to items?
- Are they transclusion links, explicitly activated links, or some of each?
- Are they only embedded in items, or can they also be tag-values?
- Are categories different from items, or is there just a single type of data that embraces both? An item belonging to a category can clearly be represented as a link, of course.

Topics

- Hypertext (p. 3512) (13 notes)
- Granular hypertext (p. 3482) (3 notes)
- Lotus Agenda

Midpoint method texture mapping

Kragen Javier Sitaker, 2019-06-01 (3 minutes)

I was thinking about Zdog and my similar `<canvas>` hack the other day for Dercuano drawings (p. 64), and today watching Κορη play a slightly-3D game, it occurred to me that a little bit of texture-mapping would go a long way to help the illusion of depth.

But both my hack and Zdog are based on, basically, spheres. How do you texture-map a sphere? The mapping from screen space to texture uv-coordinates has a couple of singularities (where the line of sight is tangent to the sphere) and so can't be reasonably approximated with a polynomial; Babbage's Method of Finite Differences is out of the question.

Consider the plane that includes the viewpoint and a scan line on the screen. The intersection, if it exists, between a sphere and this plane is a circle; along that circle we will find the points on the sphere that are visible in that scan line. If we consider only orthographic projection for a moment, we can use the standard midpoint algorithm for rasterizing circles to find the coordinates of the points in this circle. We may be able to use texel-sized steps if the sphere is close enough.

If we're using a solid texture, that's all we need — we apply the texture function (suitably strength-reduced) to the coordinates and get the colors to paint, but if we're using a flat texture, we may need to map from 3-D space to the texture's uv-space. Ideally we can use a polynomial fit (again, strength-reduced) to map into that space.

Adding perspective back in is a matter of stretching the middle of the scanned texture pixels while compressing the ends, some all the way to zero. We should be able to do this with a simple polynomial approximation, probably cubic, of the transformation from the orthographic view to the perspective view.

Actually, though, can't we do this all the way through? The transformation from screen pixel space to texel space has two singularities in it, but the inverse transformation, from texel space to screen pixel space, is singularity-free and actually relatively calm. Perhaps we can just walk along the texels on the appropriate path along the sphere's texture map, with a stride size small enough to transform at least two texels to every screen pixel, transforming each one to screen space and painting it into a screen buffer. A strength-reduced cubic spline should be perfectly adequate.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Graphics (p. 3483) (91 notes)
- Math (p. 3564) (78 notes)

• Circle midpoint algorithm (p. 3377) (2 notes)

Maximal-flexibility designs for printable building blocks

Kragen Javier Sitaker, 2019-04-20 (18 minutes)

I played Minetest a lot for a while (see Why Minetest is so addictive (p. 1781)) and I've been thinking a lot about lego-like construction sets and Minecraft voxels, in particular the voxels used for flowing water and lava, which have varying heights. More like real-world sand, Minetest water or lava (and, I assume, Minecraft water and lava) has a certain angle of repose; if you have a water source on top of an otherwise flat surface, the water forms a very obtuse cone around it, out to a certain maximum radius. The water height within any given voxel is an affine function of X and Y, and the water heights in adjacent voxels are equal along their common edge, at least in equilibrium. The result is that you can get a smoothly sloping surface, with a crude approximation of curvature, out of a finite number of distinct voxel types.

This led me to thinking of 3-D printing and marching cubes or marching tetrahedra. If you wanted to print out cubical building blocks that snapped to a voxel grid to do this kind of smoothly sloping heightfield, you'd need to quantize the height at the points of the X-Y lattice; the minimal number of heights would be 2, and this yields five types of building blocks, with empty and full corners expressed in the order (0, 0, 0), (1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 1, 1), (1, 1, 1), (1, 0, 1), (0, 0, 1): FFFF FEEE, FFFF FFEE, FFFF FFFE, FFFF FFFF, and EFFF FEFF. Here a "full" corner has material inside the cube touching the corner, though possibly only in some directions, while an "empty" corner has no material around it, and indeed no material anywhere nearby. The intended shape is something like the convex hull of the "full" points.

The fourth block is just a cube; there's a sixth "block" EEEE EEEE that is just empty space and thus doesn't need to be printed. FFFF FEEE, FFFF FFEE, and FFFF FFFE are different kinds of ramped surfaces, two with three √2 edges forming a slanted triangle, and one in the middle with two √2 edges and two 1 edges forming a slanted rectangle, all three with 1-1-√2 triangles on some vertical sides. The final block, EFFF FEFF, is missing two opposite corners, so all six of its paraxial faces are right triangles; it's a triangular antiprism, and it isn't needed for heightfields. These five blocks can be rotated and somehow stuck together at grid nodes to approximate any surface mesh produced by marching cubes, rounded to the nearest grid node.

If we're looking for maximal expressiveness with minimal inventory, the many symmetries of the cube are helpful; four of these five blocks are asymmetric and can be oriented in different ways to produce different shapes. (FEEE and FFFE have 8 meaningfully different orientations, while FFEE has 12. EFFF FEFF only has 3.) This rotational symmetry dramatically increases the expressiveness of this five-block vocabulary.

Or so I thought. But that's only 1+8+8+12+3+1 = 33 total voxels that can be formed, which is actually still short by a factor of 8 from

the 256 I'd expect. I realized I was missing FFFF FEFE, which can be realized as a ridge, a valley, or a saddle, but in any case has 12 distinct rotations. And EFFF FEFE, and some others. I should probably write a program to make a full inventory.

But this led me to the conclusion that perhaps, even though the cube's 24 orientations mean that any individual block can be oriented in the voxel structure in up to 24 different ways, its 8 vertices, which give 256 different possible voxels, are a difficulty. Still, five blocks (FEEE, FFEE, FEFE, FFFE, and FFFF) are sufficient for heightfields, which are sufficient for arbitrary shapes that aren't thinner than two voxels.

Hill's polyhedron, an irregular tetrahedron into six of which you can slice a cube, is an immediately promising alternative polyhedron. As it's a tetrahedron, removing any of its vertices leaves it empty, so you don't need a potentially large selection of them. As a bonus, you can assemble it into cubes and also a number of the pieces I described having to print separately above. I don't think it covers all the possible heightfields described above, though, because its faces are $1\text{-}1\text{-}\sqrt{2}$ and $1\text{-}\sqrt{2}\text{-}\sqrt{3}$, so it doesn't have the $\sqrt{2}\text{-}\sqrt{2}\text{-}\sqrt{2}$ equilateral triangles that result from cutting off cube corners.

Another alternative for approximating marching-cube surface meshes is to approximate just the surface, rather than voxels bounded by the surface, using flat triangles connected at the edges. $1\text{-}1\text{-}\sqrt{2}$ triangles are adequate for square faces and half-square faces; $\sqrt{2}\text{-}\sqrt{2}\text{-}\sqrt{2}$ triangles provide the cut-off corners; and $1\text{-}\sqrt{2}\text{-}\sqrt{3}$ triangles provide the remaining cutting planes.

Other tilings of 3-space ("honeycombs") may also offer good tradeoffs. Although the close-packings of rhombic dodecahedra and cuboctahedra aren't particularly promising in themselves, the duals of these packings packing consist of tetrahedra and octahedra. With tetrahedra and half-octahedra, you could build a version of these packings that can be truncated at a wide variety of planes.

Octahedra have the same set of 24 rotations as cubes (it's actually called chiral octahedral symmetry; its point group is the symmetric group S_4) while tetrahedra have 12 (A_4, the alternating subgroup of the symmetric group S_4). So an asymmetric octahedral piece could have as many as 24 usefully different orientations, and an asymmetric tetrahedral piece could have as many as 12.

I'm having a hard time visualizing the close-packings at the moment, but I think each face in the octahedral-tetrahedral honeycomb is shared between a tetrahedron and an octahedron. If that's the case, a single asymmetric octahedron and eight asymmetric tetrahedra would have $12^8 = 430$ million usefully distinct configurations; by the time you have the six octahedra and eight tetrahedra necessary to surround a point in space, the configuration space is unbelievably huge.

Stud patterns

Returning to a cubic honeycomb for the moment, two square faces can be joined together in any of four orientations; if we want all such faces to be compatible with all other faces, the simplest option is to make them all identical. But they must be symmetric under not only those four rotations, but also some kind of half-turn around an axis in the plane of the face to bring a face around to face another identical

face; that is, the three-dimensional contour of the face itself must possess chiral octahedral symmetry. It could, for example, possess male organs at 1 o'clock, 4 o'clock, 7 o'clock, and 10 o'clock, and corresponding female organs at 11 o'clock, 8 o'clock, 5 o'clock, and 2 o'clock. Indeed, fully an eighth of the face could be devoted to each such organ. But how should these organs interlock?

Thinking about legos (in particular, Lego-brand legos versus inferior underdog knockoff legos like the "Loc Blocs" I had as a kid), I had an epiphany: the very short insertion distance of Lego-brand studs is a natural optimization result for such frictional connections. The strength of the two-piece assembly is proportional to the frictional force (in one direction, it is precisely the frictional force), while the energy to assemble or disassemble it is jointly proportional to the frictional force and the insertion distance. (In fact, it is their product.) The impact energy the two-piece assembly can withstand without coming apart is also, in one direction, the energy to assemble or disassemble it, so lower assembly energy means lower impact resistance, but it need not mean lower strength. That strength can be arbitrarily high despite arbitrarily low assembly energies, at least in the limit of arbitrarily rigid material shaped with arbitrarily tight tolerances. The Lego company's tolerances are around 2–10 microns.

Roughly approximating, Lego-brand studs are inserted to about half a millimeter with about 5 newtons of force, so an 8-stud brick needs about 20 millijoules to assemble or disassemble.

PLA

This is reassuring for the prospect of 3-D printing building blocks using PLA, which is somewhat weaker and enormously more rigid than the when assembled ABS used in Lego-brand legos, or even the shitty polystyrene used in "Loc Bloc" brand legos. The consequence is that PLA has dramatically less impact resistance than ABS, and also can store dramatically less elastic energy when pieces are snapped together.

Typical RepRap-style FDM has curious precision characteristics: an error of some 100 μm horizontally (in the X and Y directions) which can sometimes be reduced to 50 μm or less, but typically a worst-case error of 150 μm or more in the Z direction, due to quantization to typically 300-μm layer height to avoid unreasonably long printing times. These impose a minimum scale on interference-fit parts which depends on the geometry: the parts need to stretch or squish or bend by at least the dimensional error when assembled in order to have any contact at all in the dimensional-error worst case. Ideally, they need to deform by an amount that is large relative to the dimensional error, so that the dimensional error won't result in dramatic variations in assembly/disassembly force (and assembly impact resistance), as it does with many kinds of poorly made construction sets.

I don't remember what PLA's elongation at break is, but let's suppose it's around 1%, similar to steel's yield strain. (Nylon is around 30%, and ABS is substantially less.) That means that a simple mortise-and-tenon joint with a deformation of 300 μm needs to be at least 30 mm wide in the deformed dimension! A simple mortise and tenon is not far from the geometry certain Lego-brand legos use, in particular the one-unit-thick plates. Such a geometry will not work

with RepRap-printed PLA until you reach pieces 1000 times the volume of Legos.

Prong clips

However, cantilever beams as used in many molded-plastic snap joints should work. 1% elongation means that you can bend a uniform-thickness strip of it in a circle of whose diameter the strip thickness is 1%: 100 mm diameter if the strip is 1 mm thick, for example, or 30 mm diameter if it's 300 μm thick, or 10 mm diameter if it's 100 μm thick. Cantilever beams get slightly better performance than that using a linear taper to get a uniform stress distribution, but it's not too far; so a 300-μm-thick uniform-thickness PLA cantilever beam can deflect by 300 μm without breaking if it is at least 9.5 mm long.

This is objectionably long, but it need not protrude by 9.5 mm; it can be recessed and zigzag as desired, in ways that are impractical in molded parts, reminiscent of coil springs but potentially much more sophisticated.

I don't know PLA's Young's modulus, either, but an old snapshot of Wikipedia gives polystyrene's Young's modulus as 3–3.5 GPa, which is probably in the ballpark; the plastics feel about equally stiff, although PLA is much more fragile. This suggests that if that 9.5-mm strip is 1 mm wide and 300 μm thick, bent into a circular arc with a surface strain of 1% and thus an average strain of 0.5%, it's under a total force of about 4.5 N, half tension and half compression, working over lever arms which vary proportionally with the stress and so average about 100 μm; this means the force to deflect the beam by that much is about 45 millinewtons.

Calculating this force in another way, the specific energy of (my guesses about) PLA amounts to stretching it by 10 microns per millimeter, requiring 30 MPa of stress; ½ 30 MPa 10 μm/mm = 0.15 J/mℓ, so PLA can tensilely store 0.15 J/mℓ, or half that in the beam-bending case, 0.075 J/mℓ. This strip is 0.00285 mℓ, so that works out to 214 microjoules. If that's built up over a deflection of 300 μm, the average force should be 713 millinewtons, with a peak force of 1.4 newtons. So I biffed a calculation somewhere.

By sticking a hook on the end of such a clip, we can amplify this force with an inclined plane, but probably only by a factor of two or three — at some point the frictional force will get out of control and the thing will just break instead of sliding in and out as desired. (The hook does have the major advantage that you can make it easier to assemble than to disassemble — same energy, but lower force.) The solution is probably to put many such thin strips in parallel like the pages of a book.

Suppose you have a 3-mm-side square hole to work with. You can have two parallel prongs that fit into it, each tipped with a hook, each of whose shafts consists of many 3-mm-wide, 300-μm-thick strips with 150-μm-wide spaces between them. The shafts deform by 300 μm upon insertion, coming into near contact (100 μm of space left in case the fabrication comes out too thick) and snap back by 150 μm upon full insertion. This gives us 2.6 mm of space to divide among these strips, meaning that there can be about 3 of them in each prong — 433 microns rather than the 450 described above, so only 283 microns of thickness in each strip. And these strips are being bent

S-curve-style rather than cantilever-style, since their ends are not free to rotate relative to one another (unless we want to try for a living hinge pivot, which seems inadvisable) so the prongs need to be about 13 mm long, which could quite reasonably be half recessed without even zigzagging.

The hook ramps can reasonably give a 2:1 mechanical advantage for insertion (600 microns deep for 300 of deflection) and a 1:2 mechanical disadvantage for removal (75 microns deep for 150 of deflection), so that the removal force is four times the insertion force, plus friction. Most of the 3-mm-wide hole can be oversized so as not to contact the hooks until they are almost at depth, so only the last 675 microns of movement have friction. The prong tips are 3 mm (almost) by 1.3 mm, so they are quite robust relative to the 300-micron-tall hook on their side.

Better than prong clips

But there's no reason to put the springs outside the building blocks in long prongs like that where they're vulnerable to breakage. Nearly the entire volume of the building block can be devoted to spring flexures that permit hooked studs on the outside to move or help capture inserted studs.

But you only have a sixth of the block to devote to the springs for each face, assuming you have connectors on all six faces. (It might be better to default to connectors on three or four faces for most projects, both in order to ease assembly and in order to ease printing; passive recesses can frictionlessly accommodate excess male stud prongs). If you want to be on the order of the Lego-brand assembly energy per face, which I estimated above at 20 millijoules, the one-sixth of the block devoted to that face needs to contain 0.27 mℓ, so the block as a whole needs to contain some 1.6 mℓ of PLA. If it needs to be 25% empty space, it needs to occupy at least 2.13 mℓ. Probably it's best to use a large safety factor and allocate, say, 8 mℓ per cube, which is precisely 20 mm on a side. This is significantly coarser than Lego-brand resolution, but not outrageously so; far better than we have any right to expect, actually, given the outrageously inferior qualities of PLA for this sort of thing.

Maybe the studs (or stud parts, since they ought to have opposing motion in order to grip locally rather than globally) should have motion that isn't purely parallel to the surface of the block, so the inclined-plane effect is larger than you'd expect from the shape of the hooks.

Topics

• Physics (p. 3632) (119 notes)
• Materials (p. 3560) (112 notes)
• Manufacturing (p. 3558) (50 notes)
• Digital fabrication (p. 3411) (42 notes)
• Self-replication (p. 3703) (24 notes)
• 3-D printing (p. 3301) (23 notes)
• Building blocks (p. 3354) (3 notes)
• Minetest

Minimal distributed streams

Kragen Javier Sitaker, 2018-04-27 (5 minutes)

I want a Wiki thing that provides some kind of knowledge ratchet for the things I work on in my spare time. In particular, I want to be able to write things on my cellphone (including calculations) and not lose them or have them inaccessible from another cellphone or computer.

The absolute minimal mechanism for this is some kind of text formatter that supports wiki links and some kind of sync protocol. The simplest feasible sync protocol is probably something like this, with each speaker uttering an append-only numbered sequence of lines:

"Give me any lines from Bob in the range 206–215."
"Bob line 206 is wejgoiewjgojagoijg."
"Bob line 207 is 302t020ujgs."
"Give me any lines from Alice in the range 201–211."
"Alice line 201 is jw0t23it20s."

This requires some kind of out-of-band mechanism for noticing that the connection has been established, or deciding when to make the request, and it doesn't have any "now you are up to date" indicator or any way to indicate failure. However, it doesn't leave the door open for unbounded quantities of future messages, all the messages are idempotent, it's convergent when messages are lost or when individual nodes fail or are restarted, and it doesn't require any session state. And it works as publish-subscribe as well as fetch-store. (You could even add sender-side message filters.)

It doesn't accommodate new speakers at this level of the protocol. Much. I mean normally you would only ever send a requested line but I guess you could make an exception.

Making this work offline on a cellphone is easiest in a browser app in JS. This suggests that maybe the server should be JS too, probably using Node. If I want to run it on adjuvant, though, which is probably a decent idea for prototyping, a CGI program is probably better. (Adjuvant doesn't have working TLS at the moment, though. I could probably fix that.)

You could require that the channel names ("Alice", "Bob") be public key hashes and that the lines each be signed. This probably isn't necessary for an initial prototype.

A browser app can itself be cached with a cache manifest, but it needs some way to store the data being synchronized. LocalStorage is the easiest and gives 5 megacharacters; WebSQL on Mobile Safari gives another 5 megabytes, and IndexedDB also works on Mobile Safari and supposedly gives 50 megabytes. On my browser, using the Browser Storage Abuser, it claims to have gotten 910 MB, but half the time it doesn't load at all. And it seems to be persistent after a month and a half, which is probably long enough to reconnect to the internet at least once.

Text formatting with wiki links could be something along the lines of

```
input.replace(/&/g, '&amp;')
```

```
    .replace(/</g, '&lt;')
    .replace(/"/g, '&quot;')
    .replace(/\n\n/g, '\n<p>')
    .replace(/\[(.*?)\]/g, '<a href="$1">$1</a>')
```

although obviously you could go quite a bit further with real Markdown or something. Markdown is not ideally suited for Wiki markup but it can work.

You need some kind of interface for handling update conflicts.

IndexedDB is the tricky bit here. With IndexedDB, I can quite reasonably store 50MB of text — perhaps compressed, so more like 150MB. But IndexedDB is not a very stable interface, and it's kind of complicated. Firefox by default treats it as a cache; specifying `{storage: "persistent"}` requires using an interface that is incompatible with Chrome and probably Safari. Worse, when Firefox evicts, it evicts the entire origin; there's no way to ask that it please save a little bit of data. (Although maybe a cookie or something would work.) And the callback approach recommended in MDN doesn't work in Chromium. (Oh, wait, it does. You just have to add the callbacks early enough, and subsequent opens are blocked until the earlier ones are closed.)

The IndexedDB interface is, like, you open a database, and it gives you a database-opening callback during which you can create and delete tables ("object stores"). Then once it's open, you can open a transaction on it, and in the transaction you can execute get, put, and delete on the tables. All of this is hidden behind an event-driven interface. In some browsers you can do this:

```
try {
  let db = await indexedDB.openDatabase('db')
    , tx = db.transaction('store')
    , store = tx.objectStore('store')
  ;
  await store.put('value', 'key');
  console.log('Put is totally done');
  await tx;
  console.log('Committed');
} catch (ex) {
  console.log(ex.message);
}
```

But I'm not sure the iPhone browser I want to use is among them.

Topics

The future of the human energy market (2014)

Kragen Javier Sitaker, 2014-04-24 (19 minutes)

In 2013, a momentous, revolutionary change, decades in the making, came to pass, almost unnoticed. Silicon solar photovoltaic energy became cheaper than coal in much of the world. Consequently, the majority of world marketed energy will be solar by around 2020–2026, and the problems humanity has to face will be completely different thereafter. Many things we think of as problems today will no longer be problems, and many things we don't think of as problems yet will be problems.

What does it mean for the cost to be lower?

The cost of energy, particularly for power plants, is tricky to calculate because the investments are long-term and often have unforeseen consequences. But energy utilities do these calculations on a regular basis, depreciating investments in physical generation infrastructure over XXX years and including cost of labor and fuel.

By these calculations, XXX in many parts of the world, utility-scale solar photovoltaic energy is now cheaper than fossil-fuel energy. This is new as of 2013. It was never true before.

The major reason for this has been the precipitously dropping price of silicon photovoltaic cells, due to an explosion of new low-cost photovoltaic manufacturers in China.

In the case of solar energy plants, the fuel is free; the vast majority of the cost of the energy is just the cost of building the plant, much of which is the cost of the solar cells that went into it.

One of the significant expenses in photovoltaic cell production is energy. But cheaper photovoltaic cells will lower the price of solar energy, which will lower the price of photovoltaic cells further. The ultimate limit is the cost of the raw materials and the depreciation of the capital equipment needed to produce the cells in a lights-out plant. As explained below, the cost of the raw materials is extremely low.

Some people have published calculations claiming that the embodied energy of photovoltaic panels — that is, the amount of energy consumed in their production — is greater than the amount of energy that the panels produce over their lifetime, and therefore using fossil-fuel energy to produce panels is just a waste. This was probably correct up to the 1980s, but the current energy payback time on photovoltaic panels is probably around one year, while the panels are designed to last 30 years, but in practice usually last longer.

What are the ultimate and practical limits to solar energy?

The "solar constant", the amount of sunlight that reaches Spaceship Earth, is about 1400 watts per square meter above the atmosphere. At ground level, it's about 1000 watts per square meter, due to clouds,

dust, and air absorption at some wavelengths. But that's a square meter at right angles to sunlight, not parallel to the ground. Tropical latitudes receive more sunlight because they're closer to being at right angles to the sun, while areas near the poles receive the least.

The total amount of sunlight striking the earth is XXX

Total world marketed energy consumption is XXX

You could imagine that even if the total amount of energy available is very large, limitations on manufacturing capacity or raw materials will limit us to harvesting a small fraction of it, as they have for the past several million years.

Indeed, many solar energy technologies do have some limitations. Thin-film panels invariably need indium, gallium, or both, although in small amounts; these metals are already more precious than silver, and they're mostly produced as a byproduct of zinc mining, so their supply is very inelastic — the prices will have to go up a lot more before people start opening indium and gallium mines. (Substantial amounts of indium could be recovered from discarded LCD panels if they can be efficiently separated from the garbage stream.) And germanium and other exotic semiconductors used in high-end multijunction solar cells are also rare.

But silicon photovoltaic cells, the current mainstream solar panel technology, do not suffer from these problems. They are principally made of silicon, with aluminum conductive traces, and then doped with trace amounts of group III and V elements such as phosphorus and arsenic — all among the most common elements in Spaceship Earth's crust. Given sufficient energy and equipment, you could make silicon photovoltaic panels out of almost any random rock, although perhaps at a somewhat higher cost than using the currently popular ores.

Entire panels contain some additional elements: aluminum frames, copper or aluminum wires, and tempered glass, which is made from calcium, silicon, sodium, and oxygen, using fairly inexpensive processes.

There's still the risk that, even if the raw materials intentionally included in the solar panels are cheap, the capital equipment needed to convert them into solar panels might be expensive. For example, parts of the manufacturing process involve melting silicon, which requires very high temperatures, and typically platinum-iridium crucibles. Even if silicon is abundant, platinum and iridium are not. I don't know enough to know if this risk will materialize, but if it does, there is a lower-tech alternative: concentrating solar power.

The first solar power plant was built in 1908 (XXX?) in Egypt, and used trough-shaped mirrors to focus sunlight on pipes, boiling water to drive a steam engine. This project ended when abundant oil was discovered in the area, providing a cheaper source of energy, but now that we've used up most of the oil, there are a number of similar projects.

Steam engines do not need any scarce materials, either in the finished engine or the factory to produce it, although they can be substantially more efficient, up to 40%, if made using modern alloys. And the mirrors for a concentrating solar power plant can be made from aluminum and glass. Current mirrors are made using a vacuum-deposition process which makes them almost as cheap as plain glass. It's also possible to use Fresnel lenses molded from cheap

transparent plastic.

CSP doesn't gather sunlight on cloudy days, but it has the great advantage that it is practical to use it to produce electricity at night, by storing the gathered heat in tanks of molten nitrate salts.

An in-between option is what the famous Solyndra was pursuing: by concentrating sunlight with mirrors or lenses, you can use a tiny fraction of the photovoltaic cells that you'd need to gather the same power directly from the sun, which in today's world may allow you to use high-end multijunction cells with up to 40% efficiency, but in a hypothetical world where platinum shortages limited photovoltaic production, could substantially increase the installed power. Solyndra went bankrupt because, at the moment, simple photovoltaic cells are too cheap to compete with using complicated machinery that needs to make back R&D costs.

How fast is adoption growing?

Utility-scale solar photovoltaic installed capacity is currently doubling every year in the US; recently worldwide capacity was doubling every 22 months, but in early 2013 I saw figures that said the doubling time had shortened to 8 months, presumably as a result of the much lower costs.

I don't have any numbers for non-utility scale solar (e.g. on rooftops) or other forms of solar energy (e.g. thin-film solar and concentrating solar power), but I think it's safe to say that, for now, utility-scale silicon photovoltaic has won the race and will remain the cheapest way to harvest solar energy for the foreseeable future. The other forms are much smaller.

Specifically, Intersolar reported in July that new solar energy installations (worldwide, I presume) would go from 30 GW of new installed capacity in 2013 to 100 GW in 2014. This represents a doubling time of 8 months.

My prediction that this exponential growth will continue for another decade and beyond is apparently a lunatic-fringe opinion; everybody else I can find making plans or predictions about solar-energy growth rates seems to be expecting something more like linear growth, to a double-digit percentage of electric power supply only by the mid-2020s.

What human problems exist because of energy scarcity?

Water shortages

Much has been said about the worldwide shortage of fresh water, with predictions of wars being fought this century over it. But salt water is abundant, and production of fresh water from salt water can be carried out straightforwardly with large energy inputs: either via distillation, the traditional way, or by reverse osmosis, which uses less energy. XXX Reverse osmosis plants do require significant investment in equipment, but the majority of the cost of their water is the cost of the energy they consume; and they are already cheap enough to produce water to irrigate farmland, water pure enough that it can reverse the problem of progressive salinization that has desertified many previously-fertile lands that have been irrigated by

slightly salty water.

In short, fresh water is only scarce because energy is scarce. Abundant energy will eliminate water scarcity and the risk of water wars, except perhaps for landlocked countries.

Expensive aluminum

Since Andrew Carnegie and his competitors exploited the Bessemer process to make steel the mainstream material in the late 19th century, our society has been girded with steel: steel railroad tracks collapsing the price of transport, steel rebar holding our buildings and bridges together, steel boats carrying our goods from port to port, steel automobiles ferrying us from steel-framed building to steel-framed building, where we can be shot by people with steel guns.

But the Hall-Héroult process, discovered in Carnegie's heyday, made aluminum (previously a precious metal) into a lightweight, inexpensive substitute, and it's displaced steel in some uses: airplanes, drink cans, bicycles, engine pistons, and so on. It's lighter than steel for the same strength, and it doesn't rust. Aluminum, however, is still more expensive than steel per pound and even for a given strength, so we continue using steel.

Most of the cost of the Hall-Héroult process, though, is the cost of the energy it consumes to electrolyze the molten aluminum ore, an ore which is abundant. Abundant energy will make aluminum abundant too, and it will displace steel in most applications; it will even displace plastic in some.

Climate control

Many of today's buildings, especially here in Argentina, are expensive to inhabit because they were built in an age of energy abundance — from the 1940s to the 1970s — and so are built with little concern for efficiency of climate control, since operating air conditioners was cheap at the time. Many other buildings, like those in the world's slums, are unpleasant and dangerous to live in because they don't have adequate air conditioning or heating, and are not built with sufficient resources to enable passive climate control. (Vinay Gupta's Hexayurt design is a possible alternative that provides passive indoor climate control with much less resources than traditional designs such as meter-thick adobe walls.)

Abundant energy makes it possible to heat and air-condition easily.

Energy production centralization

Current electrical energy production is carried out in centralized power plants, either because it's hydroelectric and therefore not portable, because it's steam and therefore experiences great economies of scale, or because it's nuclear and therefore is dangerous to distribute widely. About a third of it is then lost between the power plant and the consumer, and sometimes inadequate infrastructure maintenance results in widespread power outages, which are deadly. This is okay if you have good governance, but in places with shitty governance (like any slum, war zone, or refugee camp), it sucks. It also sucks if you have shitty self-discipline and blow your paycheck on smack and booze instead of paying the electric bill.

Photovoltaic panels are portable, do not experience economies of scale in use, and are not particularly dangerous. They can

substantially ameliorate the problems of inadequately maintained electrical transmission and distribution infrastructure, fragility in the face of attack, and poor governance.

Limited transportation

Much of the cost of transportation, especially air transportation but even bus transportation, is the cost of the energy needed. This cost makes traveling an unachievable dream for much of the world's population.

Airplanes, intercity buses, long-distance trains, and ships universally use liquid fuel rather than batteries because of its much higher energy density. This has led to suggestions that solar energy cannot replace fossil fuels for transport. This is a mistake. Liquid fuels can be produced synthetically from CO_2 and water; it just takes energy, and it's an inefficient process, so it won't happen until electrical energy is *much* cheaper than fossil-fuel energy.

Climate change

Global warming is caused by releasing fossil fuels into the atmosphere, either burned or unburned, and by releasing carbon dioxide from calcite in the production of cement. With sufficiently-cheap energy, cement production can be reoriented to magnesium cements derived from seawater with no carbon emissions, and we can build plants to actively remove carbon dioxide from air, either to sequester it back underground or to reduce it into combustible material, as suggested in the previous section.

What new human problems will exist because of energy abundance?

Dependence on energy suppliers; concentration of power in the hands of those who control energy production.

Pollution.

Lack of menial labor.

What new human problems will exist because of solar photovoltaic energy?

We can expect that a greater and greater proportion of our land area will be consumed by solar panels, because building them on land is easier than building them at sea. At first, much of this will take place in deserts, but eventually anyplace that gets sunlight will be fair game.

Calories are a measure of energy; a food calorie is about 4200 joules. The price of a joule in the form of food is similar to its price in the form of electricity. But solar panels are reducing that price, and they turn a larger fraction of sunlight into usable energy than natural photosynthesis does. Typical silicon solar cells convert 16% of incident sunlight into electricity, while the most efficient plants convert 7% of incident sunlight into biomass energy XXX, which then must be burned in a heat engine to recover some 2.8% of the original energy. So in the limit, an acre of solar cells will produce some five times the usable energy of an acre of sugar cane.

So we can expect food crops, as well as nature reserves, to compete with photovoltaic cells for land once the tropical deserts are used up.

However, even with yearly doubling times, that won't happen until the 2030s. Before that, it will probably make more sense to plant crops in the shadows of solar panels. (Current practice, which I hope stops, seems to be to concrete over the entire area to be populated with solar panels.)

Historically, we have carried out only fairly small-scale semiconductor fabrication, because it's an expensive process and because integrated circuits can be very useful while still being small. XXX These small-scale processes nevertheless produced staggering amounts of toxic waste, contaminating numerous sites around the world. To convert the world energy infrastructure to photovoltaic, we will produce semiconductor wafers by the hectare, with a correspondingly large possible increase in toxic waste.

While we won't have water wars, we probably will have wars for access to tropical areas with low cloudiness, such as the Sahara and the Atacama.

It hardly seems worth mentioning, but the Economist predicts that European utility companies may go bankrupt when solar energy lowers the price of energy below the cost of operation of their existing fossil-fuel and nuclear plants; as a result, their market capitalization has already dropped by half a trillion dollars. (!)

How is this practically different from biomass or agriculture?

As I mentioned above, current solar panels can collect some five times the usable energy from sunlight that biomass and agriculture do.

How will solar abundance be distributed?

More tropically. England's kind of fucked, as eloquently calculated by David MacKay in Sustainable Energy without the Hot Air, while North Africa is sitting on a gold mine — but geopolitically lacks the power to keep it from being exploited by other powers.

XXX

Should we conserve energy?

Yes, both because right now much of our energy produces CO_2 by burning fossil fuels, causing global warming, and because energy is expensive. Miners and drillers die to bring coal and uranium to your power plants and gasoline to your cars.

Should we make long-term investments to reduce energy usage?

No. Marketed energy will become abundant in the mid-2020s. An efficiency investment that saves a dollar a year now will turn into saving an inflation-adjusted dime or penny a year then.

Some people will do it anyway. Here in Buenos Aires, I hear people making arguments about how conserving fresh water is important, while the Rio de la Plata a couple of kilometers away discharges 22000 cubic meters per second of fresh water into the salty Atlantic (which is to say, 600 000 liters per day per inhabitant of Buenos Aires), and every construction pit in the city needs a sump

pump to constantly pump fresh groundwater out into the street. They seem to be inspired by the virtue of asceticism more than any actual knowledge about the issues.

But in an energy-abundance regime, it will make as much sense to try to conserve electricity by not using it as to try to conserve sunlight by sitting in the shade instead of out on the beach.

How, where, and by whom are panels made today?

Topics

- Pricing (p. 3646) (89 notes)
- Energy (p. 3438) (63 notes)
- Solar (p. 3717) (30 notes)
- The future (p. 3746) (20 notes)
- Climate change

Fast message router

Kragen Javier Sitaker, 2017-06-15 (updated 2019-07-23) (15 minutes)

Suppose you have a bunch of small processes, like the size of httpdito (which uses 12k or 16k of memory maps XXX no it has more memory maps than that), running and sending requests, responses, and change notifications to each other. How fast could they reasonably do this?

I wrote this test called syscallovh.c to get a ballpark:

```
char c[s];
int fd = open(devzero, O_RDONLY);
if (fd < 0) {
  perror(devzero);
  return 1;
}

for (int i = 0; i < n; i++) {
  read(fd, c, s);
}
```

It finds that, on my current laptop (Intel(R) Pentium(R) CPU N3700 @ 1.60GHz under Linux debian 4.4.0-21-generic), a system call takes about 300 ns, and bulk-copying bytes into userspace takes about 171 ps per byte, or, let's say, 175 ns per kibibyte. This suggests that, to keep the system call overhead under 10%, we need IPC message buffers to usually be in the neighborhood of 16 kibibytes or bigger. Such a bufferful of data should take 300 ns + 16 * 175 ns = 3.1 µs to process.

The sample CoAP request from Appendix A of RFC 7252, the CoAP RFC, is 16 bytes long (0.01 GET /temperature MID=0x7d34) and it receives an 11-byte response (2.05 Content "22.3 C" MID=0x7d34), so sending it through a Linux system call by itself incurs about 100× overhead: 300 ns for the system call plus 2.7 ns to transmit the actual request, plus a comparable amount of work for each of the three steps where the server receives the request, the server sends a response, and the client receives the response, a total of 1.2 µs. A buffer of 1024 such requests or responses, by contrast, would require 3.1 µs, or 3.1 ns per request — 12.4 ns per request for the full request-response cycle. This would allow my four-core laptop, naïvely, to handle 32 million request/response pairs per second, or 500 000 request/response pairs per 60Hz screen refresh.

As some kind of comparison, on a machine similar to this, httpdito can serve (and ab can measure) an HTTP request in about 50 µs, which takes about 2700–3200 instructions in the spawned child process and 29 instructions in the parent process, plus some unknown amount of work in the kernel on its behalf, which is actually the great bulk of the execution time.

As another comparison, here's the usual dumb fibonacci microbenchmark:

```
fib(n) { return n < 2 ? 1 : fib(n-1) + fib(n-2); }
```

On my laptop, this takes 920 ms to calculate that fib(40) is 165580141, running 2,288,656,125 instructions (2.49 billion per second), which works out to 5.6 ns per leaf call. There are almost exactly twice as many total calls as there are leaf calls, so this is 2.8 ns per function call and return, or 6.9 instructions per function call and return.

This is probably an unusually serial benchmark. One core of the machine can presumably run about 4 billion instructions per second with more typical levels of ILP, giving these crude ballpark numbers:

task	ns	insns	reqs	insns/req
syscall/ret	300	1200	0	1200/0
4 syscall/rets	1200	4800	1	4800
httpdito HTTP txn	50000	200000	1	200000
1024-req buffer 4 syscall/rets	12400	49600	1024	48.4375

#+TBLFM: $3=$2*4::$5=$3/$4

(It's amusing that the kernel is presumably running about 200 000 instructions while being scripted by httpdito running about 3000 instructions.)

Perhaps for messages sent over a network, this enormous microsecond-scale overhead of computation per request/response pair is unavoidable, but for processes on a single machine, it seems like it should be avoidable. A message broker that accepts buffers full of messages from other processes, copies the messages around appropriately to other processes, and sends the buffers off to the other processes should be able to cost somewhere between the 48 instructions per message that copying them minimally costs and the 4800 instructions per message that the kernel charges us for more or less the same job. If we could manage 512 instructions per message, for example, that would be 128 ns, several times faster than doing a couple of system calls per message. This would scale down to pieces of work as small as 32–64 function call/return pairs, and be efficient for pieces of work as small as 512 call/return pairs.

(Even across a network, if each node has a message broker talking to other message brokers across the network, it may be feasible to reduce the overhead; alternatively, zero-copy networking hardware may be able to store incoming packets directly into the message buffers of waiting processes.)

For communications topologies that change much less often than messages are sent over them, such as with flow-based programming, Unix pipelines, or farming work out to worker threads, no message broker would be needed to amortize per-system-call overhead over many messages.

What might such a protocol look like?

You probably want the messages to have an 8-byte-aligned fixed-length header with a length field counted in 8-byte units, rather than the bytes used by 0MQ and CoAP. You need to be able to have thousands of outstanding requests at a time, which probably requires you to be able to accept results out of order. You probably want your header fields to be either single bits or entire bytes in order to avoid the need for extra shift instructions.

You probably don't want per-message authentication and

encryption, not only because it impedes routing but also because it takes too long. In The security impact of a new cryptographic library in 2012, Bernstein, Lange, and Schwabe report that NaCl can run 80 000 public-key authenticated encryption or authenticated decryption operations per second an a 6-core 3.3GHz AMD Phenom II X6 1100T, which is presumably in the ballpark of 750 000 instructions per operation. The paper mentions a faster interface consisting of crypto_box_beforenm, crypto_box_afternm, and crypto_box_open_afternm, which allows you to amortize the expensive public-key operations across many messages to or from the same correspondent.

However, even the ChaCha20 stream cipher used by NaCl needs 5.6 clock cycles per byte for a 64-byte message on an AMD Ryzen 7 1700 at 3GHz, working out to 128 ns per message. So per-message authentication and encryption could conceivably be affordable, but it will probably use more CPU time than the message routing would.

It's crucially important that, if there are message brokers, whatever transformation they must do need not examine every byte in the message, including in particular breaking the 8-byte alignment guarantee as they copy the message from an input buffer to its appropriate output buffer or buffers.

Messages should probably normally be in the range of 16 bytes (smaller messages will have space only for the 8-byte header!) to 512 bytes (significantly larger messages will probably gain no further efficiency), with 128 bytes being the normal case. A single-byte length field would support messages up to 2048 bytes.

So, a single message might contain a datum of a level of detail such as the following:

- the above example request or response about temperature
- a 512-byte disk sector (although maybe in the age of 3D XPoint memory, user processes should be interacting directly with durable memory rather than indirecting their access to it through kernels and servers)
- a 16×16 pixel RGBA tile (1024 bytes)
- part or all of a scan line of pixels (2048 bytes in RGBA is 512 pixels; in 16-bit RGB, it's 1024 pixels; in 8-bit grayscale, it's 2048 pixels; in 1-bit monochrome, it's 16384 pixels)
- a rendered glyph from a font (some of the glyphs on my screen right now are 20×34, 12×17, 7×9, 12×9, and 7×7, thus ranging from 49 pixels up to 680 pixels; some mainstream fonts have glyphs as small as 5×7, and I've designed a proportional 6-pixel-tall font with some 2×6 glyphs)
- a millisecond or two of PCM audio; at DAT quality this is 192 bytes
- a drawing command
- a Merkle tree node with 4–32 SHA-256 hashes
- 32 single-precision floating-point numbers from one column of some table
- a touch, keystroke, or mouse event
- a SQL query
- a SQL table row
- the log message for an HTTP request
- a Redis query or response

• a line of text (but ideally not a letter, word, paragraph, page, or document)
• a 128-node binary DAG with two bytes per node and 128 implicit leaf nodes
• an AST of some 64 nodes with up to 192 implicit leaf nodes
• the bytecode for a method to be compiled or the machine-code compilation result
• a B-tree node containing 32–128 child node pointers, 64 bits each, with or without keys
• an SNMP request or response
• a Git commit, but probably not an entire Git commit history.
• a Git tree entry such as "100755 blob b734283e0473b9d77f07efee066f1486a1c5a37f wifiscan.py", but probably not an entire Git tree.
• a Tweet
• an SMS
• a financial transaction
• a stock price update
• a compilation command

It probably makes sense to layer some kind of application-layer protocol defining semantics (routing, failure recovery, naming, request-reply, publish-subscribe, caching) on top of a base data representation layer.

Many-output computations

Thinking about how to wedge a "give me the 16×16 RGB tile (768 bytes) at (112, 144) from frame 1832 of foo.mp4" service into a cached RESTful architecture makes me question the idea of caching individual message responses to achieve efficiency. Computing the tile in question, or even an approximation of it, is going to require decoding a potentially large area of the previous several frames, possibly up to a few seconds' worth of video. I'm not sure if it's feasible to break that computation down into pieces whose outputs would fit into a single 2048-byte message so they could be cached, but it certainly isn't the easiest way to bring existing video decoders into the system.

However, if instead of "give me the 16×16 RGB tile (768 bytes) at (112, 144) from frame 1832 of foo.mp4", you instead request "please store the next few frames following frame 1770 of foo.mp4 at /some/place", the video codec could issue a few hundred thousand PUT requests to store its output, and then we could fetch those outputs.

A similar kind of thing happens with, say, a drawing that gets rendered to a high resolution set of pixels — in the extreme case, the drawing is some view of the OpenStreetMap database. If we move one of the lines in the image, naïvely, that invalidates every tile of the cached image. To avoid invalidating the whole cached image, you need some kind of intermediate layer that efficiently partitions the drawing elements into those that affect different bounding boxes, then makes the individual pixel tiles depend only on the subset drawn within their bounding box. This seems feasible, but still tricky.

High-bandwidth computations

Copying a byte into or out of a user process may take only 171 ps of

one core, which suggests that you could do 23 gigabytes per second on this laptop, or 11.7 gigabytes per second out one process and into another. That's pretty decent bandwidth; 1920×1080×60Hz×4bytes is only 498 megabytes per second. But that bandwidth gets divided by the number of intermediary processes it ends up flowing through.

Still, this seems like it should be okay.

Low-latency computations

One of my examples above is "a millisecond or two of PCM audio". For media playback applications like watching a movie with bae, fairly high latency is acceptable as long as you have enough buffer to even out the jitter. But, if you're building a synthesizer out of a bunch of processes piping PCM audio to each other, even 10 milliseconds of audio latency is intolerable.

Buffering a millisecond of audio is not bad. Earlier today I wrote a softsynth in C++ to make the sound of the synthesizer in Eurythmics' *Sweet Dreams*. It has 29 very-low-level processing nodes in an 11-level-deep DAG, all of which are stateless and use continuous time. If they were being called upon to generate a millisecond of audio (192 bytes at DAT quality) we would expect sending that millisecond to require 300 ns + 192 * 171 ps = 330 ns at each node, plus another 330 ns on the node that received it; all 29 would take a total of 19 µs of system call overhead. The current C++ implementation spends about 20 µs generating that millisecond of audio, but 90+% of that is in method invocation overhead (including virtual method dispatch), because each node is generating a single sample instead of 48 samples.

(Some of the nodes are invoked more than once; there's a seven-node sawtooth C3 note subgraph that's used at two different times by the flanger, and a five-node triangle-wave C2 note subgraph that's both used to amplitude-modulate the flanged signal and added to the final mix, and the identity function is a single node that's used in six places. So this count of 29 is slightly low. Compiling the node graph into JS reveals 45 operations, but 15 of those are the identity function and could perhaps be skipped; another couple are redundant constant evaluations. So I think 29 is in the ballpark.)

So this approach, with a single millisecond of latency, would impose an order-of-magnitude overhead on soft-real-time audio-processing applications, but would still be almost two orders of magnitude more performant than necessary in this case.

In fact, it would not even be fatal if these messages were enqueued in a message broker between each node behind a 16-kibibyte buffer of other data. If each of the 9 levels of the tree cost 3.1 µs of bufferbloat, the total would be 27 µs of message-broker bufferbloat latency, less than the 39 µs of CPU usage to get all those tiny messages in and out of all those tiny processes; the total would be 66 µs, or 6.6% of the millisecond.

Topics

Everything is money?

Kragen Javier Sitaker, 2019-08-31 (4 minutes)

A remark overheard at the end of a math lecture, when the students were complaining about having to use different programming languages in different math classes (R, Octave, Python): "Everything is money." This was intended as an explanation for why essential free-software libraries for certain algorithms were not available in some of these programming languages, ruling those languages out for classes that needed to use those algorithms.

This struck me as profoundly short-sighted and unaware of the history of free-software development, particularly coming from a professor who shall not be named but who can trace their academic lineage back to Lagrange. When Euler and Lagrange were inventing the variational calculus in 1754 to 1756, Euler was indeed being paid — he was the director of mathematics at the Prussian Academy of Sciences. But spending some of his professor time on reading letters from Lagrange was entirely his own decision — not only did the providers of funding not know that Euler was doing this, but under the basic norms of Prussian academic freedom later codified by Humboldt, they did not have the right to know or to veto it. Moreover, Lagrange was not being paid for this; he did get a position as Sostituto del Maestro di Matematica for the Piedmontese army in 1755, but there he was being paid to teach calculus and ballistics to military artillery engineers.

The variational calculus wasn't published until 1762, at which point it was published by the Turin Society (*Societas Privata Taurinensis*), which Lagrange established with his students in 1757, and which today is the *Accademia delle Scienze di Torino*. It was initially established as a private club, but before the publication of the variational calculus, it had gained royal patronage from Victor-Amadeus in 1759, making it the Royal Turin Academy of Sciences (*Société Royale des Sciences de Turin*).

Much of the history of mathematics, academia in general, and free software is like this. If you want to know why a theory was developed or why a book or free-software library was written, it is only minimally informative to investigate who was paying for it and what they wanted to fund. Instead you should look for what interested the individual people who developed or wrote it, what other scholars they were in touch with, and what ideas they were influenced by.

Sometimes the *absence* of free software or research can be explained by funding. Snapshots in free-software filesystems don't exist because NetApp funded patents to stop them during many years. High-quality free-software mixed-integer linear programming solvers don't exist because researchers use CPLEX, Gurobi, Xpress, or SCIP — see Some notes on the landscape of linear optimization software and applications (p. 1285) for the whole sad story. There's no decent free-software spreadsheet for Android because people just use Google Sheets. Little public-domain research exists on isotopic enrichment because the US and Israel have funded wide-ranging efforts to prevent it, to the point of releasing industrial-sabotage

viruses into the wild and assassinating researchers.

Wikipedia, of course, is the first place any student of mathematics goes to learn about any mathematical concept. Wikipedia is written and edited by people who do not get paid, except in a minority of cases usually considered vandalism.

So, not everything is money, I think.

Topics

CCD oscilloscope

Kragen Javier Sitaker, 2017-06-20 (updated 2017-07-04) (7 minutes)

I've been trying to figure out how to build an oscilloscope from e-waste (see files TV oscilloscope (p. 1253), VCR oscilloscope (p. 213), Laser printer oscilloscope (p. 449), and Disk oscilloscope (p. 713)). The difficulty is that a basic oscilloscope has frequency response up to 20MHz, which means that if you want to digitize the signal you need a sampling rate of at least 40 megasamples per second, and such fast analog-to-digital converters (ADCs) are hard to come by. To display it without digitizing it, you would need some kind of display device with response up to 20MHz, like an analog oscilloscope tube, which is challenging to construct and very rare to find in garbage streams, especially intact.

I finally have an answer that I am confident will work: use a laser to store the analog signal on a CCD long enough to digitize it with an easily-available low-sample-rate ADC. Discarded flatbed scanners contain linear CCDs, which are already connected to moderate-speed ADCs; discarded laser printers contain lasers connected to scanning mirrors. Point the laser at the CCD and modulate the laser brightness with the signal you want to measure as it passes across. Then you can digitize the signal from the CCD at a lower speed in the usual way.

Let's consider a garden-variety 22ppm A4-size landscape-mode 600×600dpi printer laser assembly. A4 paper is 210 mm × 297 mm, so the paper moves at minimally 22x210 mm = 4.62 meters per minute, or 77 mm/s. 600 dpi is 23.6 "dots" per mm, so the printer must scan at a bit over 1800 scan lines per second; if it uses a 6-sided spinning mirror, that's 300 revolutions per second, or 18000 rpm, which is a pretty high speed.

(And it seems like a pretty average speed. The Samsung M2020W costs AR$2139 = US$134 and is 20ppm with fake 1200dpi in portrait mode; the HP Wifi Pro M12 costs AR$2900 = US$181 and is 18ppm A4 with 600dpi in portrait mode; a Xerox Phaser 3020 costs AR$1900 = US$119 and is 20ppm A4 with 600dpi, but I can't tell if that's landscape or portrait.)

For a laser printer, the laser doesn't need to be linear (it's either on or off for each pixel) but it does to be modulated at a pretty high speed. Each of those 1800 scan lines is, hypothetically, 297 mm long, with 7000 pixels per scan line. (In portrait mode you need more scan lines, but they are shorter, for the same number of pixels per page and per second.) This means the laser needs to be turned on and off at a bit over 12 MHz to meet the 600 dpi spec, but possibly 12MHz is already past the low-pass 3dB point of the electronics hooked up to it. However, laser diodes that can be modulated at many tens of MHz are easy to come by, and can be tested using a garden-variety photodiode and existing known-good oscilloscope.

A 6-sided mirror sweeps the laser across some 120 degrees in each scan, which are linearized by some weird one-dimensional optics in the laser assembly. They are linearized over a much longer region than the CCD from a scanner, which is typically as small as possible to cut down on silicon wafer costs, but which has its own set of optics to focus an image from typically the width of A4 or A3 paper onto it.

Although it limits bandwidth, these two complementary sets of optics could be left in place with a sheet of paper between them as a diffuser, but maybe you could get better performance (lower "dark current" due to stray light) if you could eliminate them and just scan the reflected laser directly across the CCD, maybe using a mirror and a second laser at a different angle to cut the sweep angle down to 30°, and compensate for the temporal nonlinearity across the CCD in software. This allows you to scan the laser across the CCD several times (such as four, in this example), which keeps the CCD's spatial resolution from being the limiting factor on the bandwidth.

The number of points you can sample from the captured analog signal on the CCD presumably depends on the resolution of the original scanner, but even a shitty scanner is 300 dpi, which across a 210 mm A4 page gives you 2480 pixels. A normal scanner is four times that; a high-end scanner is eight times that. 2480 samples is several times what you need for a decent DSO.

I think there may be some "dead time" during which it is impossible to capture the signal, either because the scan angle is too large for the CCD or because the laser is hitting the corner of the mirror or because the CCD is going through some kind of reset process between frames or something. There is definitely "dead time" on the laser end of the process, at least in the printers I've disassembled, where the laser is directed elsewhere than through the optics onto the page for part of the scan, thus getting lost.

Any other kind of camera would also work for converting from spatially modulated light (integrated over some frame time) back into a signal temporally modulated slowly enough to be easy to digitize. It's just that the CCD from a scanner is high-quality (typically more than 8 bits per pixel) and has a lot of pixels already in a line.

If no lasers are available, a possible alternative is to use an LED viewed through a spinning mirror and focusing optics, i.e. take a picture of the spinning mirror in which the LED is reflected with a camera. Many LEDs have relatively low junction capacitance and consequently can respond far more rapidly than they really need to for their intended use.

If no mirror is available either, it may be possible to do the job by spinning the camera (relative to the LED) instead, although this involves potentially tricky electrical connections to the rapidly spinning camera or LED. To avoid that problem too, you could nutate the camera or LED rather than rotating it, although that seems likely to be mechanically tricky and will also require trickier time base correction.

Topics

Writing math in Unicode with the Compose key

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

I can write quite a bit of math in a quasi-reasonable way with Unicode combining characters. For example:

This one uses COMBINING OVERLINE; xterm supports this, but Emacs, gnome-terminal, and Firefox 2 don't:

$z\bar{z} = x^2 + y^2 = |z|^2$

Using • for the compose key, I type this as `zz•^__ = x•^2 + y•^2 = |z|•^2`.)

This one totally abuses it, and it doesn't look that great in xterm either:

$\sqrt{a^2+b^2}$

That's `•sqa•^__•^2•^__+•^__b•^__•^2•^__`, which is kind of a pain, but still kind of cute.

This uses superscript and Greek characters:

$\theta = \tan^{-1}(b/a)$

That's just `•*u = tan•^-•^1(b/a)`. My Greek bindings, which are based on the standard Greek keyboard layout, are adequate for math, but not for typing Greek, because they don't include accents. And, as you can see, typing things like $x^{32^{07}2}$ is a bit of a pain.

There are logic characters:

$\forall x \in \mathbb{Z}: \exists y \in \mathbb{Z}: y > x$

`•AAx •in • Z: •EEy •in • Z: y > x`

$\forall x, y \in \mathbb{Z}: x < y \vee x = y \vee x > y$

`•AAx, y •in • Z: x < y •\/ x = y •\/ x > y`

Definition of XOR:

$x \oplus y = \bar{x}y + x\bar{y}$

`x •(+) y = x•^__y + xy•^__`

Or in different notation:

$x \veebar y = \neg x \wedge y \vee x \wedge \neg y$

`x •\_/ y = •-,x •/\ y •\/ x •/\ •-,y`

I'm not sure if this is any clearer than the C:

`x ^ y = ~x & y | x & ~y`

And I can write:

$f \circ g = \lambda x.f(g(x))$

`f•o*g = •*lx.f(g(x))`

Also I have a keybinding for the vector arrow, even though its rendering is pretty suboptimal in every piece of software I have handy:

$\vec{F} = m\vec{a}$

`F•^> = ma•^>`

An example from the Zeldovich HiStar paper:

$L_1 \sqsubseteq L_2$ iff $\forall c: L_1(c) \leq L_2(c)$.

`L•_1 •[= L•_2 iff •AAc: L•_1(c) •<= L•_2(c)`

Topics

Electric hammer

Kragen Javier Sitaker, 2018-07-02 (updated 2018-07-05) (14 minutes)

A hammer is a simple machine; you apply energy to it over a long period of time, and upon impact, all that energy is released in a short time. As an example, you might swing a 2kg hammer in a 3m-diameter circle at 2Hz, which works out to 9.4 m/s and 89 J, and it might take you 500ms over 2.4 m for the whole swing, thus an easily human-achievable 37 N and 178 W (though at least the power must vary, increasing as it does from zero); upon impact, this energy is released within, say, a millimeter, which is about 200 µs, 450 kW, and 89 kN. So in effect you have a mechanical advantage of about 2400 — your 37 N applied over 2.4 m is amplified to 89 kN applied over 1 mm.

Correspondingly, your 178 W is amplified by 2500 to get 450 kW, because it's released over 1/2500 the time.

Springs can be used in a somewhat similar way, and also in reverse, which is a thing we don't usually do with hammers. Springs have the feature that the input force and distance are always the same as the output force and distance, but the input and output power need not be. A bow, slingshot, or onager is the hammer-like approach — you apply a small power over a long period of time to charge up the spring, then release the energy over a short period of time. Applying the same principle in reverse, you can wind a watch, applying a large power over a short period of time, and then let it run down, applying a small power over a long period of time.

A traditional simple machine like a lever doesn't have the variability in time that hammers and springs do — the instantaneous power in is always equal to the instantaneous power out. A rough electrical equivalent of the lever, if we disregard dc, is the transformer — the input voltage can have whatever relationship to the output voltage, but the power in is equal to the power out.

Capacitors and inductors can be used like springs or hammers in the electrical world. You can add energy to either one slowly over a period of time, then release it over a short period of time; in the case of the inductor, the voltage is arbitrarily different from the original voltage, while the current changes continuously, while in the case of the capacitor, the current is arbitrarily different form the original current, but the voltage changes continuously. As one example, engine spark plugs are fired with ignition coils that work this way, and flashtubes are fired with capacitors that work this way. Some spot-welding machines also use either capacitors or inductors in either of these two ways, and of course ordinary switching power supplies use inductors precisely in order to change voltages. And it's easy to provoke accidental, and dangerous, energy releases from either kind of device. Heck, I just spewed sparks of vaporized copper into the air when I plugged in my power strip because it has a wall-wart plugged into it.

In this process, the non-ideal properties of the various devices, including even switching elements, come into play. The ESR and ESL of capacitors limits how quickly they can discharge (or charge), their leakage current wastes energy, and the ESR also causes energy

losses during both charge and discharge; parasitic capacitance of inductors limits how quickly they can store or release energy, and their ESR causes energy losses during the whole time they are storing energy. The limited energy density of either device also

For capacitors, the resistive losses are concentrated in the high-power part of the cycle, because they are quadratic in current. Consider charging a 1µF 1Ω capacitor with a constant-current source to 1 V over 1 second and then discharging it, also at constant current, over 1 millisecond. You charge it at 1µA, eventually depositing 1µC in it, dissipating 1pW and thus finally 1pJ in its ESR (and 1µJ in its dielectric). When you discharge it at 1 mA, it's dissipating 1µW, a million times as much, and finally 1nJ, a thousand times greater.

For inductors, instead, resistive losses are concentrated in the low-power part of the cycle. If you spin up a 1mJ 1Ω inductor with a constant-voltage source to 1 A over 1 second, then discharge it through a constant-voltage load in 1 ms, your current rises linearly from 0 A to 1 A, and the resistive loss rises quadratically from 0 W to 1 W over that second, dissipating a total of ⅓J. During the discharge cycle, it drops again from 1 W back to 0 W, playing the same curve backwards but a thousand times faster, and so dissipating only ⅓µJ.

So I was wondering about the limits of this kind of thing.

The Pulse Electronics PA4309.105NLT unshielded drum-core inductor is 1mH (±20%), 1.08 A, and 3.915Ω; it's 7mm × 12.8mm × 12.8mm. At its full current, it stores 0.58 mJ and dissipates 4.6 W, which I guess means it only stores about 127 µs worth of power, about 0.5 J/liter. I don't know what its self-resonant frequency or parasitic capacitance are, but Pulse measures the inductance at 100kHz, so I'm guessing it's at least 1MHz. So you can probably get a boost or dropdown of up to about a factor of 100 or 200 higher or lower voltage and power out of these inductors, with microsecond-scale spikes of a few hundred watts and volts.

Their PA4309.104NLT is 0.1mH (±20%), 3.5 A, and 0.405Ω, and is the same size. It can store 0.61 mJ, almost the same as the previous one (which makes sense, since all that's changed is the wire!) and dissipates even more, 5.0 W (which, again, makes sense).

The PA4309 series goes down to a 1µH 20A 19mΩ inductor of the same size, which is slightly higher power (7.6W) and lower energy (0.2 mJ).

So it seems like, for energy-storage purposes, the power, energy per unit size, and cutoff frequency don't really vary with inductance, at least within a given core material and size.

Capacitors can handle much longer energy storage (going beyond microseconds up to seconds, hours, even longer) as a result of us having much better approximations to perfect insulators than to perfect conductors, at least around our normal temperatures. They're also much denser.

A random capacitor I have lying around (in a broken CFL bulb) is a can 8mm in diameter, 17mm long, 3.3 µF, 400 V. This works out to 264 mJ in 0.854 mℓ, 309 J/ℓ, 600 times denser than the inductors I was looking at. I don't know how fast it can discharge and whether it's electrolytic (as it appears) or polyester (it says "PET" all over one side). But I think electrolytics are good up to about 10kHz or so, so it might not be able to do under 100 µs or so. In the CFL, it's being used to smooth the ripple in rectified 50Hz ac.

A random inkjet printer board I had sitting around has a Nichicon supercapacitor on it: 8mm diameter, 20mm long, 2.7 V, and an astounding 2.2F, which works out to an astounding 7.3 J in 1.0 mℓ, and thus 7.3 kJ/ℓ. However, supercapacitors have high ESR and much lower capacitance over short timescales.

Checking out datasheets and stock at Digi-Key, the Nichicon JUWT1105MCD costs 92¢ down to 30¢ (quantity 5000). It's a 1-farad 2.7V 6.3mm-diameter 10.5mm-long EDLC rated at 4Ω ESR at 1kHz. The datasheet lists a 4Ω "DCR", but I don't know what that is. The capacitance rating is based on discharging in, I guess, 270 seconds after a 30-minute (!) charge cycle. (See Capacitors: some notes on tradeoffs (p. 134).) This works out to 0.33 mℓ, 3.6 J, and 11.1 kJ/ℓ. The datasheet gives 4Ω ESR at 1kHz, so its maximum energy output at around that timescale would be into a 4Ω load, which would work out to 460 mW.

Another inkjet printer power supply has an electrolytic that is 16mm in diameter, 35mm long, 50 V, and 2200 µF, which works out to 2.75 J in 7 mℓ, 390 J/ℓ.

At the other end of some kind of scale, I have a microwave oven capacitor which is 25 µF and 450 V, 50 mm diameter and 70 mm long, 137 mℓ, 2.5 J, 18 J/ℓ. This is presumably designed for high currents and low ESR, although its extensive surface markings do not provide any indication of how high and how low.

Comparing to batteries, a lead-acid Yuasa GYZ16H holds 16 amp-hours at nominally 12 volts (690 kJ), is 150 mm × 87 mm × 145 mm (1.89 ℓ), weighs 12.4 lbs (5.6 kg), and can deliver 240 amps (nominally 2.9 kW, but I bet the voltage dips) to start your motorcycle. This works out to 365 kJ/ℓ or 120 kJ/kg, orders of magnitude better than even the supercapacitor. Li-ion and Li batteries are even better, and combustible fuels are orders of magnitude better again.

Murata has a line of "GR7" ceramic capacitors specifically optimized for camera flashes, the largest of which (GR731CW0BB473KW03#) is 3.2 mm × 1.6 mm × 1.8 mm. These are ceramic capacitors rated for 350VDC, 10nF to 47nF, with 50mA max discharge current, which works out to at least a megavolt per second, except that its capacitance droops below 50% at full rated voltage, so it's more like two megavolts per second. The power output of this device is an astounding 17.5 watts, but its energy content is only 2.2 mJ, which works out to 234 J/ℓ. I don't know what their ESR or ESL is; the datasheet lists their "dissipation factor" as 0.025 max at 1V and 1kHz.

Most promising are tantalum capacitors like the AVX TAJB226M010RNJ, which costs 69¢ in quantity 1, 23¢ in quantity 1000. It's 3.5 mm × 2.8 mm × 2.8 mm, 22µF, 10V, has an ESR of 2.4Ω at 100kHz, and has a leakage current of 2.2 µA. This works out to 2.2 mJ, 27 µℓ, and a rather poor 80 J/ℓ. The leakage current will steal 100 mV/s, so these are only good for storing energy for timescales of up to a minute or two. Discharging through a 2.4Ω load, it can theoretically deliver 2100 mA and thus 20 W.

So, suppose I want to dump 1000 J into a spark that lasts 1 ms — a rather high power of 1 MW. If electrolytic capacitors are my medium of choice, I need 3.2 ℓ of electrolytic capacitors! If I were to use tiny ones like the one from the CFL, I'd need 3800 electrolytic

capacitors. Using the bigger one from the inkjet, I'd need 360 capacitors, totaling 2.5 ℓ.

(Putting this power into context, arc welding on steel commonly uses a spark of 40 V at 150 A, melting a weld puddle a few millimeters across, which works out to 6 kW. This spark would generate heat 167 times as rapidly.)

Supercapacitors are apparently capable of discharges over a timescale of milliseconds, but their ESR is too high; using the one whose datasheet I examined above, you'd need 2.2 million capacitors, containing a total energy of 7.9 MJ, to get an instantaneous 1 MW out of them. Also it would cost you US$650 000 and they would occupy 720 liters.

Using Murata's camera-flash capacitor, on the other hand, you could get 1 MW out of only 57000 capacitors, which is actually a more reasonable quantity than it sounds like — the capacitors are tiny, so it's only 530 mℓ. But the total energy capacity would be only 126 J, which is an eighth of what I specified above, even if it is a bit larger than the literal 89J hammer blow I started the explanation with. To get to 1000 J you need 450 000 capacitors, but they "only" occupy 4.2 liters. I don't have the price handy but I imagine it would add up to around US$45000.

The AVX tantalums seem like perhaps a better balance: they hold the same 2.2 mJ as the ceramic capacitors, but can deliver a higher maximum discharge current despite the lower voltage, so the output power ends up being almost the same. Unfortunately, they occupy even more space and are probably more expensive.

Topics

• Electronics (p. 3430) (138 notes)
• Physics (p. 3632) (119 notes)
• Pricing (p. 3646) (89 notes)
• Hammers (p. 3491) (3 notes)

How can we do online pitch detection?

Kragen Javier Sitaker, 2018-04-27 (updated 2018-04-30) (6 minutes)

These are my notes on pitch detection algorithms for three DSP projects I want to do: the Magic Kazoo, the R2D2 Light Switch, and Ultranarrowband Speech.

The projects

The Magic Kazoo is a synthesizer the size of a regular kazoo that you play mostly by holding it in your mouth and humming into it. It does live looping, sampling, synthesis of a variety of instruments, rhythm accompaniment, harmony, and autotune.

The R2D2 Light Switch is a light switch that you control by whistling at it to turn lights in your house on or off, or possibly dim them, change their color, or activate disco dancing strobe mode. Instead of running regular speech recognition algorithms which require a lot of processing power and are easily confused by background noise, I'd like to use a small number of chirp-based whistle signals, maybe with analog components to control brightness.

For the Magic Kazoo, I want to be able to do pitch detection on a distorted human voice, vastly outweighing any noise, with very low latency, like, under 10 ms, with relatively low computational cost, like under 50MIPS. For an R2D2 Light Switch, I want to be able to detect whistled pitch patterns even in the presence of substantial background noise, but I can tolerate latency up to maybe 200 ms, and it would be reasonable to use gigaflops if necessary. For Ultranarrowband Speech, pitch estimation is a necessary element, and it operates under fairly strict latency constraints (traditionally up to some 40 ms).

Why does the Kazoo need so little latency? Brandt and Dannenberg around 1997 say: "There do not seem to be published studies of tolerable delay, but our personal experience and actual measurements of commercial synthesizer delays indicate that 5 or maybe 10 ms is acceptable. This is comparable to common acoustic-transmission delays; sound travels at about 1 foot/ms."

Such low latency is particularly challenging because human voice pitches are often as low as 100Hz or even lower, which means 10 ms is a single period. Worse, the precision needs to be pretty high; if we autotune, we can afford up to a quarter-step of pitch error, which is about 2.9%. At 44.1ksps, 100Hz is a period of 441 samples, so the period estimation can be off by only up to about 13 samples. This is definitely doable with either zero-crossing detection (with a sufficiently amplified waveform) or various kinds of autocorrelation-based measures (with low distortion).

Ultranarrowband Speech is an effort to develop a comprehensible speech codec below one kilobit per second. However, I have recently learned that such algorithms already exist, like David Rowe's Codec 2, which manages comprehensible speech at 700 bits per second.

The algorithms

Zero-crossing counting

Simple counting of zero-crossings, on the raw waveform or on variously filtered versions, is the simplest thing to try. But to achieve the kinds of latency we're talking about here, we need to use the lapse between the zero-crossings, not the number of zero-crossings during some predefined interval. Perhaps the median lapse during the last 10–20 ms or so would be the best measure to use for this.

The downside of counting zero crossings is that you discard almost the entire wave, and that wave has a lot of information about the current phase which it would be better not to discard.

Quadrature encoding with the derivative

Another approach that has occurred to me is to use the instantaneous amplitude and derivative to estimate a phase quadrant, which you can use like a quadrature encoder to count revolutions. It also gives you phase information twice as often as simple zero-crossing counting.

Phase-angle tracking with the derivative

Going further in that direction, you could imagine estimating a phase angle, and timing the first time the phasor reaches a given phase angle during each cycle. At each new angle, you can measure the time lapse since it reached that angle during the previous cycle.

Measuring the derivative can be done in a variety of ways, but it's important to keep in mind that the derivative is proportional to the frequency, among other things. So you have to multiply the derivative by something to get the phasor to follow a roughly circular pattern near your desired frequency (say, within an octave or two of it). As long as you're privileging a certain octave, you might as well run some degree of low-pass filtering over it as well — for example, differencing the latest sample from a simple moving average over the last N samples, or differencing a shorter moving average from a longer one. The width of these moving averages provides a crude low-pass filter, and the lag between their centers provides a high-pass filter.

Particle filters

Another approach is to maintain a variety of hypotheses in memory about what the waveform is doing, updating the probability of each one in a Bayesian fashion after each sample, then resampling the population of hypotheses according to the implicit distribution.

The hypotheses can be things like "repeating the wave N samples back with M amount of Gaussian noise" or "silence" or "a new wave with a period around P Hz".

Autocorrelation

This is sort of the gold standard. If you're working on a 10-ms window at 44.1ksps, you need 441*440/2 = 97'020 multiply operations, most of which are multiply-adds. You could maybe do this every millisecond, which ends up as 2200 multiply-adds per sample, 2.2 million per second. Then you need a bit more work to look at the autocorrelation spectrum and pick out the peaks. This is probably too slow to do in real-time on many processors, but it's an easy first thing to try, and provides a basis for comparison for other algorithms.

Topics

Cross current zone melting

Kragen Javier Sitaker, 2016-10-06 (1 minute)

In additional to the usual countercurrent configuration for minimal heat loss and the cocurrent configuration occasionally used, a recuperator-type heat exchanger may be arranged in a cross-current configuration for process intensification, where you have alternating layers of pipes in the X and Y directions. (A rete mirabile is a much better way to do high-density heat exchange, but that is a different topic.)

This cross-current configuration is optimal, however, for a different purpose: rapid zone melting. Although I suspect that the speed of zone melting is limited by crystal growth speeds, maybe you can still do it faster by doing it in thin pipes (made of a material that doesn't dissolve significantly in the material you're trying to purify, of course). You can run hot coolant through only one or two cross-current pipes above and below the layer of the material being purified to melt it in a narrow region, while perhaps simultaneously running cool coolant through other adjacent pipes to intensify the temperature gradient.

By switching between pipes, it should be possible to rapidly move the molten zone along the material being purified.

Topics

- Materials (p. 3560) (112 notes)
- Thermodynamics (p. 3747) (49 notes)
- Mechanical things (p. 3569) (45 notes)
- Zone melting

The Magic Kazoo: a synthesizer you stick in your mouth

Kragen Javier Sitaker, 2017-04-04 (updated 2019-05-12) (6 minutes)

I thought I must have written about this in detail elsewhere, but I can't find it.

I propose the Magic Kazoo, an electronic musical instrument. You use it by putting it in your mouth and singing into it; the pitch, volume, airflow, and tonal quality of your voice, together with buttons on it, control a synthesizer running on one or more microcontrollers to produce music emitted from a built-in speaker. With live looping, you are a one-man band.

Real-time audio synthesis is no longer power-hungry

Modern microcontrollers — even MSP430s and AVRs, but especially the ARM Cortex-M series — are powerful enough to do real-time audio synthesis easily, even running on tiny batteries. We should expect the vast majority of the power consumed by the Magic Kazoo to be the power dissipated in its speaker.

Pitch detection can be super simple; amplitude detection may be harder

Your voice recorded on a microphone that is actually inside your mouth is enough to saturate just about any normal microphone, and it's reasonably sinusoidal, so you can get the frequency just by counting zero-crossings. You can't get the amplitude in the usual way, because it's totally saturated and does a pretty good impression of a square wave. You may still have 200–500µs or so in between saturated positive and saturated negative, though, and the slope in that 500µs (or equivalently its length) kind of tells you what the amplitude of the whole sine wave would be if you could record it.

Given that, you can detect sound onsets with amplitude, measure the frequency with time between zero crossings, round to the nearest halfstep, and control a softsynth. Maybe you can use the inferred volume contour of the sound to give you further control over the sound.

Even ceramic capacitors might be adequate microphones in such extreme circumstances

(The power of the sound is such that you might be able to use things that you wouldn't normally use as microphones. Like high-barium-titanate ceramic capacitors, which are a lot cheaper, and which can sometimes generate over 100mV piezoelectrically, or even over two volts when Dave Jones bangs them on wood like drumsticks .)

Interfaces to the outside world

Aside from the speaker, it has a headphone jack.

Should air pass through it?

Since you can breathe through your nose, it doesn't need to have air pass through it, and if you take that option, you can make music for yourself or a headphone listener by humming quietly. This does remove the possibility of controlling a dimension of the music with airflow, though.

Sliders, knobs, and buttons

In addition to the continuously-variable frequency and amplitude inputs from your voice, the possible continuously-variable spirometry input, and the possibility of maybe detecting other aspects of your vocal-cavity resonances, the Magic Kazoo probably needs some further controls. The variety of things you *might* want to control are endless: adjusting the volume, selecting from thousands of existing instrument sounds, recording a pattern to loop later, enabling or disabling existing patterns, starting or stopping a canned beat-box rhythm, adjusting tempo, and any of the endless variety of effects pedals out there; and if you're using it to construct your own virtual instruments, you could start with adjusting attack, decay, sustain, and release, recording samples, autotuning them, and then selecting what part of the sample gets used, generating sounds from systems like Karplus-Strong and FM synthesis, and so on.

There is, however, a very limited amount of space available on a kazoo-sized thing for the full set of controls you might want on a digital audio workstation, and even less screen space for feedback. I have this little 8-color ballpoint pen here that's about the right size; as mentioned in A phase-change soldering iron (p. 2270), it's 17 mm in diameter and 130 mm long. The eight colors are selected by eight sliders somewhat like the digit inputs on a Curta calculator, which slide about 25 mm; things that are more closely spaced than that probably aren't too practical, and even those might be pushing it — I can't move two adjacent sliders at once because my fingers are too big. So controls need to be at least 13 mm apart to be simultaneously operable.

For such a device, there are strong incentives in favor of capacitive touch sensing instead of physical buttons: moving parts break, especially on instruments played by children; physical buttons tend to let water in, which is a problem for an instrument normally partly immersed in saliva; the higher parts count and extra assembly steps for physical buttons raise the cost; and physical buttons are generally limited to detecting contact or non-contact, with no analog levels, until you go to potentiometers (which really exacerbate the other three factors). But normal touch sensing is kind of a no-go for a musical instrument, because it generally requires looking at the thing and responding to what you see, which adds far too much cognitive latency. It's *really* important to be able to find controls with your fingers before you activate them.

A possible solution is to expose the touch surfaces through holes in a touchable template, which allows you to find the "button" or "slider" hole you want by feel alone, then use a bit more pressure to squeeze your finger skin through the hole so it touches the surface.

Topics

Mechanical buck converter

Kragen Javier Sitaker, 2016-06-20 (5 minutes)

Springs apply a force that increases as they are deformed from their equilibrium form. To compensate for this, old clocks and crossbows use the horological fusee to compensate.

But, in electronics, to compensate for the similarly variable voltage from batteries and especially from supercapacitors, we use buck converters instead. Buck converters periodically dump a pulse high voltage into an inductor, which ramps up current rapidly in response, and then ramps down (perhaps more slowly) when the high voltage is disconnected. The roughly constant but oscillating current through the inductor is used to charge a capacitor, whose voltage oscillates slightly as the current rises above and falls below the current drawn by the load in parallel with it. The voltage on the capacitor is used to control the pulse width.

The mechanical equivalents of inductors and capacitors are springs and masses, but there is a duality; interchanging current and voltage, and inductance and capacitance, you get a new circuit that functions similarly.

In this case the particular analogy I was thinking of was attempting to maintain a constant but smaller force (analogous to voltage). If force is voltage, then velocity must be current, right? Because the thing you multiply force by to get power is velocity. So the component analogous to a resistor would be a dashpot, as expected: velocity proportional to force. The component analogous to an inductor, with the force proportional to the derivative of velocity (acceleration), must be a mass. And so a capacitor must be a spring, with the velocity proportional to the derivative of force, which is to say, the position is proportional to the force. So far, it all checks out.

So periodically we allow velocity to flow briefly from the mainspring to a mass, such as a flywheel, for example through a freewheel clutch mechanism (analogous to a diode). Then we use this flywheel to energize a spring, such as a torsion bar, connected to a load. The torsion bar is maintained at a roughly constant but slightly oscillating torsion as the flywheel is accelerated by the mainspring, then left to slow down under the influence of that torsion bar or whatever. And then we use the total force in that spring to control the duty cycle with which energy is dumped into the flywheel, by applying and removing a brake (a clutch, really, connected to a fixed shaft that cannot rotate) from the driving side of the freewheel.

It's easy to imagine two rotating actuators, one attached to the flywheel and the other attached to the load, whose difference measures the spring force in the torsion bar, like the pointer of a torque wrench. And you could imagine these two things engaging and disengaging the clutch once per revolution. You probably want those revolutions to be pretty slow, like under 1 Hz, to get acceptable clutch life. You could achieve this either with a large flywheel or by gearing down the relative motions of these timing devices.

So this gives you an efficient, if probably unreliable, kind of continuously variable transmission, with a built-in governor that produce a constant output force (torque), regardless of how fast the

output is rotating, as long as the input force is equal or greater.

As an alternative feedback mechanism, you could control the duty cycle of the clutch to obtain a constant output velocity rather than a constant output torque, for example using a centrifugal fly-ball governor.

If we run this the other way around, we have a mechanical boost converter instead. In this configuration, the input shaft is allowed to freely spin up a flywheel, which is at times not connected to any load. Periodically, however, a clutch is engaged, connecting the flywheel to a freewheel which winds up a spring. A second freewheel keeps the spring from unwinding while the drive clutch is disengaged.

When the clutch is engaged, the flywheel can decelerate much faster than the input drive, providing much greater force. If the force required is much greater than the force provided by the input, the clutch will be disengaged most of the time, while if it is almost the same, the clutch will be engaged most of the time.

All of these contrivances seem to me perhaps foolish compared to the reality of continuously variable transmissions achieved by existing methods.

In the world of hydraulics, the boost converter is well known in the form of the hydraulic ram; I suspect that pursuing the analogy further is likely to be fruitful.

Topics

• Physics (p. 3632) (119 notes)
• Mechanical things (p. 3569) (45 notes)

Complementary goods in home economics

Kragen Javier Sitaker, 2017-07-19 (3 minutes)

Suppose you have a budget of US$1000 to spend on two complementary goods, A and B. The benefit you get from the combination is jointly proportional to the quantity of A and the quantity of B that you purchase. Then you should spend US$500 on A and US$500 on B, because if x is the amount you spend, your benefit is proportional to x(US$1000-x), whose maximum is at x=US$500. Note that this doesn't depend on the unit prices of A and B!

And it only depends a little bit on constant offsets, for example if spending x on B gets you 10 + x/$15 units of B; you can simply impute the cost of the constant offset to your budget and spending on B, in this case pretending that you are spending US$150 more on B than you really are, out of a total budget of US$1150. (So in that case you want to spend US$575 on A and US$425 on B.)

You'd think that if there are three analogously complementary goods involved, you should probably spend equally on each of the three. And yes, it turns out that although xy(1-x-y) blows up to positive infinity in several different directions, its only maximum for positive x and y is at x=y=(1-x-y).

I think this reasoning is applicable fairly generally in home economics, although you have to contend with diminishing returns as well.

For example, spending US$1000 a month in rent and maintenance costs on your apartment implies that you should probably also spend US$1000 a month to decorate it, or whatever else makes it enjoyable. If you're spending more than that on decoration, maybe you would enjoy it more if you moved into a bigger apartment and spent a bit less on decoration; and if you're spending less than that on decoration, maybe you should move into a smaller apartment so as to have more money to make it habitable.

Here I'm assuming that "maintenance" costs like cleaning and repairs, as well as your rent, scale proportional to the size of the apartment (adjusted by some quality factor), but there are cases where this is backwards — some kinds of maintenance, like repairing damages from a leaky roof, are a matter of wasting money as a result of renting a low-quality good. In those cases, it would be more intelligent to rent a higher-quality good instead of wasting money on maintenance.

Does this mean that spending US$500 a month on car payments (or the time-adjusted equivalent in up-front costs) implies that you should probably also spend US$500 a month on mileage-proportional costs such as fuel, oil changes, and maintenance?

Having a nicer, faster, or more efficient car means that time you spend in the car is more pleasant, and you can travel proportionally further in the same time. But if that expense means that you don't have enough money to pay US$500 a month in variable costs, maybe you would get more value out of a cheaper car that you could afford

to drive more.

But maybe you have a lot of money and not that much time to spend traveling in the car.

Topics

Sparse sinc

Kragen Javier Sitaker, 2019-09-15 (10 minutes)

Is there a reasonable way to approximate a low-pass time-domain sinc filter with a sparse filter cascade, in the sense described in Sparse filters (p. 834) and elaborated, for example, in Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547)?

A sinc consists of two sine waves of the same frequency, 180 degrees out of phase, joined with a kind of parabolic lump in the middle (the "main lobe"), and with an overall hyperbolic envelope: go twice as far out and the oscillations are half as high. A sort of important thing is that the zero-crossings are at perfectly regular intervals, except that there's one missing in the middle, in the center of the parabolic lump, which is instead the maximum of the whole function.

Most commonly, instead of convolving with the full sinc function, which has annoyingly long support, we convolve with a windowed sinc function, whose amplitude diminishes considerably more quickly.

A couple of different ideas occur to me for how to do this.

Beating frequencies with a Gaussian window

If we want a sine wave to go through a 180-degree phase shift, a simple way to get this is to generate a beating signal by adding two sine waves at frequencies above and below it. For example, if we want a 22,050-Hz sine wave to switch polarities 100 times a second, we can add equal-amplitude sine waves at 22,000 Hz and 22,100 Hz. If we put a tight window around a zero-crossing of the phase, we can attenuate the signal sufficiently that only a single crossover has a significant amount of energy.

Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) shows how to put a Gaussian window on a sine wave by composing a feedback comb and a feedforward comb, and by adding two different windowed sine waves you can get much of the tails of a windowed sinc. For better or worse, the amplitude will diminish in the tails as $\exp(-x^2)$.

This doesn't provide the lump in the center, but perhaps we can get a reasonable lump out of a simple (non-oscillating) Gaussian and add that in.

The above gives us about 15 additions and subtractions per sample.

However, I think the envelope of this approach will be too far from correct, because the sidelobes immediately next to the main lobe will have amplitudes that are too small.

This approach is particularly interesting for a slightly different application: as an ideal *bandpass* filter has a frequency response of a symmetrical low-pass boxcar convolved with an impulse at its center frequency, it has a time-domain response of the low-pass boxcar's sinc *multiplied pointwise* by a sinusoid at the center frequency. Where the

sinc crosses zero, the center-frequency sinusoid reverses phase. So the beating trick can be used to get some of those sidelobes.

Summing a bunch of in-phase Gaussian-windowed oscillations

Suppose that, instead of trying to generate a beating frequency and then window it, we start with a single Gabor kernel (a Gaussian-windowed sinusoid) of, say, an eighth of the overall window width. Then we build up the overall hyperbolic envelope by adding delayed and amplified copies of the thus-windowed signal, careful that the delays don't put the various copies out of phase (except in the middle). Perhaps six or eight copies should thus be adequate to get a good approximation to the overall hyperbolic envelope, and then, as before, you need to add in the central lump.

The copies on opposite sides of the central lobe are the same amplitude, so you can sum them first and then multiply them by the amplitude; with this approach, eight copies will require four additions and four multiply-accumulates, all starting from the same windowed sinusoid.

With this approach we should need about six adds and subtracts to get the initial windowed sinusoid, six adds and subtracts to get the central lump, and four additions and four multiply-accumulates to put the whole thing together: 20 operations in all.

Summing exponentially-decaying oscillations

The things above all work using Gaussian-windowed oscillations, but while a real sinc decays as $1/x$, the Gaussian decays as $\exp(-x^2)$, which is a lot faster. In between, we have exponentially-decaying oscillations, such as those proceeding from the feedback comb filter

$$y(t) = x(t) - ky(t-s)$$

where $0 < k < 1$. Maybe such a decay could produce one of the tails of the sinc filter efficiently, with a slower decay than the Gaussian and potentially a much sharper onset, although you might want to tone that down a bit in order to reduce discontinuities.

A feedforward-comb trick similar to the trick used to get Gaussian-windowed oscillation can cut this exponential off after some number n of oscillations by composing with a filter something like this:

$$y(t) = x(t) - k^{2n}x(t-2ns)$$

This will inevitably have some rounding error, but the rounding error will decay exponentially thereafter, so it shouldn't be a major concern in this application. This allows you to put together the tail out of a piecewise-exponential decaying oscillation, although you can't derive each piece from the same exponential the way you could with the Gaussians.

You might think to just use $1/k$ for the other tail, but in that case the inevitable rounding error from the cutoff would grow exponentially out of control. A better approach, when it's feasible, is bidirectional filtering --- not in the filtfilt way, where the two unidirectional filters are *composed*, but where the results of the two unidirectional filters applied to the same input signal are *added*.

This bidirectional approach might give adequate results with a single exponential tail in each direction, with the first impulse in the oscillation being a quarter-cycle away from the origin. This would be two subtractions and two multiplications per sample, plus whatever low-pass nonsense you need to do to get rid of the third and higher harmonics.

Summing flat oscillation plateaus

The procedure mentioned earlier for generating a Gaussian-windowed oscillation consists of convolving three or more rectangular-windowed oscillations. But another possibility is to *add* scaled rectangular-windowed oscillations; for example, 64 oscillations scaled by 1/64, 32 oscillations scaled by 1/64, 16 oscillations scaled by 1/32, 8 oscillations scaled by 1/16, 4 oscillations scaled by 1/8, 2 oscillations scaled by 1/4, and one oscillation scaled by 1/2. This gives you a sort of stepped-pyramid approximation of the sinc shape, which you can then perhaps smooth a bit by convolving it with one or more windowed oscillations at the same frequency.

This stepped pyramid requires a *single* oscillator (one subtraction), seven subtractions to get the steps, then six additions (with binary scale factors) to get each side of the sinc from them, for a total of 20 operations per sample. Several more operations might be needed to add the central lobe and smooth the steps.

(Maybe triangular-windowed oscillations would be a better basis function, at least for the final tail; or you could tag on an exponential decay.)

Not worrying much about high harmonics

Any departure from the perfect sinc shape in the resulting filter's impulse response represents some kind of departure in the frequency response from the desired boxcar --- either in passband flatness or in some nonzero response in the stopband. But some kinds of departure are less harmful than others; for example, the comb filters we're using here have not one passband but an infinite number, spaced at odd harmonics of the normal passband. In some sense this is an extreme departure from ideal behavior, but it's not a particularly bad one; unless the desired passband is very wide, the extra passbands are located far enough away that it's easy to filter them out adequately with fairly simple low-pass filtering.

By contrast, if you want a filter that passes 140 Hz to 150 Hz, and you instead get a filter that passes 140 Hz to 151 Hz, that extra hertz will be an extremely difficult error to remove. These sparse filters might actually be able to do a better job at this than conventional FIR and IIR filters, because they can afford to use much wider support, which means they can get much sharper frequency cutoffs.

Undertone filtering

Because comb filtering passes (either all or odd) harmonics just as strongly as the fundamental, you can compose combs at different frequencies to get a filter that passes the common multiples of those frequencies; for example, one third, one fifth, and one seventh of the desired frequency (35/105, 21/105, and 15/105), maybe with a feedforward comb to notch out any unwanted frequencies. This is

useful when the frequency you want is not close to a factor of your Nyquist frequency (or half the Nyquist frequency, in the negative-feedback case).

(An alternative in such cases is often to use spectral inversion.)

Topics

Handling Landsat 8 images in limited RAM with netpbm

Kragen Javier Sitaker, 2014-04-24 (4 minutes)

Landsat 8 images are now freely available, including the 15-meter panchromatic band that was new in the Landsat 7 ETM+ sensor. The thermal bands have lower resolution than in the L7 sensor, but whatevs.

An immediate problem is how to deal with the images in the Level 1 Data Product. They're 16-bit TIFFs with dozens to hundreds of megapixels; it turns out that, at least on my netbook, things like ImageMagick (`display` and `convert -sample` anyway) and the GIMP just do not manage to open them in a reasonable amount of RAM. netpbm seems to be able to come through here, successfully generating a thumbnail of a 225-megapixel image in only three minutes:

```
anytopnm LC82250842013110LGN01_B8.TIF | pnmscale -height=600 | \
    pnmtopng > small-pan.png
```

The results in a rather dim, but still 16-bit, image, with a maximum pixel value of 9007; a contrast autostretch with `pgmnorm` helps, but leaves the image proper rather low in contrast, because the black border pixels are so far from the image values. `pnmhisteq` results in a striking and highly legible grayscale.

In the thumbnail, Buenos Aires is 40x34+365+278 out of the 624x600 image (found with `display small-pan.png`, then Image Edit → Region of Interest). The original image is 15201x14621, according to `anytopnm LC82250842013110LGN01_B8.TIF | head -2`. That means I should be able to extract a 974x829 image at (8892, 6774) to find the city, using `pamcut`:

```
time anytopnm LC82250842013110LGN01_B8.TIF | \
    pamcut -left 8892 -top 6774 -width 974 -height 829 | \
    pnmtopng > buenos-aires.png && \
    < buenos-aires.png anytopnm | \
    pnmhisteq | \
    pnmtopng > buenos-aires-stretched.png
```

It would be a lot more efficient to break these images up into compressed tiles of near the latency-bandwidth product of the disk drive, using `pamdice`, and mipmap them, rather than spending over a minute rereading the entire 400-megabyte GeoTIFF for every operation, but at least it's workable. The above took a little under 90 seconds, producing a striking image of most of the city (my RoI was a little too conservative) in `buenos-aires-stretched.png`. Every street and every plaza is clearly visible, but not every building. Using `pgmnorm` instead of `pnmhisteq` gives much better results, with large-scale patterns becoming visible. The resulting city map is only about 1.5MB in PNG form or 470kB in JPEG form.

It would probably be a big improvement to add chroma data from the other bands, but that's a little more work.

Now that it's so easy to get unlimited Landsat 8 scenes, we can quite reasonably make movies of construction and the like, albeit on a block-by-block basis rather than a building-by-building basis. That is, you'll be able to see which block construction is happening on, but not what shape the buildings are, unless they're humongous.

(If we figure the latency-bandwidth product is 500kB and that we need about 1.5 bytes per pixel, the tile size would be around 600×600; this particular scene would then be 25×26 tiles, and it would probably make sense to produce 3:1, 9:1, and 27:1 reductions, adding another 1+9+27=37 reduced-size tiles to the original 650; the total would be some 350MB. Divide by roughly two to account for the fact that the zero pixels added around the edges compress to almost nothing.)

Extracting bands 1, 2, and 3 for red, green, and blue, we have

```
for band in 1 2 3; do
    time anytopnm "LC82250842013110LGN01_B$band.TIF" |
    pamcut -left 4446 -top 3387 -width 487 -height 415 |
    pnmtopng > buenos-aires-$band.png
done
rgb3toppm <(anytopnm buenos-aires-3.png | pgmnorm) \
          <(anytopnm buenos-aires-2.png | pgmnorm) \
          <(anytopnm buenos-aires-1.png | pgmnorm) |
    pnmtopng > buenos-aires.rgb.png
```

This produces a somewhat legible color image of the same region as previously, but it's fairly unsaturated.

Topics

• Graphics (p. 3483) (91 notes)
• Unix (p. 3765) (7 notes)
• Datasets (p. 3402) (5 notes)
• Satellites

Hardware multiplication with square tables

Kragen Javier Sitaker, 2019-02-08 (updated 2019-07-09) (4 minutes)

If you can build a high-speed routing network in a small amount of silicon, you can build an extremely high-throughput multiplier for high-latency multiplies by using square tables.

$xy = ¼((x+y)^2 - (x-y)^2)$; for example, 579×414 is $¼((579+414)^2 - (579-414)^2) = ¼(993^2 - 165^2) = ¼(986049 - 27225) = ¼\ 958824 = 239706$. (See Multiplication with squares (p. 1983) for more on this.) This seems potentially interesting in the context of modern high-throughput computing, in which multipliers occupy a significant amount of silicon area and power consumption. Squaring a number can be done with a table lookup, which is relatively light on power consumption. The remaining addition and two subtractions can be done quite a bit more easily.

The difficulty, of course, is that a large table occupies a very great deal of silicon area. But perhaps all is not lost — we can share this large table among many concurrent multiplications, if they are latency-tolerant.

Consider, for example, a 32-stream 10-bit×10-bit multiplier built using this scheme. Each pair of 10-bit numbers entering the multiplier at one of 32 multiplier ports gets its sum and difference taken; these are sent into a mesh routing network to be squared, each tagged with an output port. The 1024-entry lookup table is split into 64 independently operating lookup tables of 16 entries each; on every cycle, each such lookup table receives a 10-bit input from the network along with the routing tag (or, occasionally, an idle notification), and in a single cycle places the 20-bit or 18-bit result and the routing tag onto a second mesh routing network. So, every cycle, we're doing 64 10-bit squarings. These squares make their way through the second routing network to output buffers, where they are paired up with their partner from the input and subtracted.

The queuing-theory bit is somewhat tricky in that it's totally plausible to do a bunch of multiplications by the same number or by similar numbers. Similar numbers can be broken up a bit by some kind of simple hash ($x \wedge x >> 5$, for example) so they don't all hit the same lookup-table shard, but some amount of replication is probably unavoidable to handle many multiplications by the same number. Alternatively, perhaps the original numbers themselves could serve as the routing tags, thus permitting the coalescence of multiple concurrent requests for the same number; this would require somehow duplicating the results on their way to the subtractors. Regardless, some amount of nondeterminism in performance is unavoidable with this scheme.

The routing network from the inputs to the multiplication table can probably consist of three layers of 16 4×4 17-bit crossbar switches with output buffers to wait on contention; similarly the routing network from the table to the output can probably consist of three layers of 16 27-bit crossbar switches.

Such a device could carry out 32 10×10-bit multiplication results

every cycle, with a typical latency of 8 to 11 clock cycles. The bulk of the silicon area is the 1024-entry 20-bit (or 18-bit) square table; the crossbars, adders, and subtractors should be relatively small by comparison. (Actually, is that bullshit? Is it possible that those crossbars are fucking huge, even though they don't contain many transistors, just because of the wires?) Scaling it up to, say, 16-bit × 16-bit multiplications would involve scaling the table up to a 65536-entry 32-bit table, which is 100 times larger; but, if split among a correspondingly larger number of independent table lookup units, this would perform 4096 16×16-bit multiplications every cycle, with a variable latency of 15 to 20 clock cycles. At 400MHz, it would provide 1.6 trillion low-precision multiplies per second.

If an electronic crossbar switch turns out to be infeasible, another possibility is optical free-space routing.

Topics

The Stretch book is truly alien

Kragen Javier Sitaker, 2018-11-27 (6 minutes)

Reading the 1962 book about Stretch. There are a lot of strange ideas and terminology in here.

"Edit" seems to be used in a sense I don't understand.

IBM, especially at that time, had a habit of plagiarizing other people's ideas, so it's hard to tell what's really new here; but Stretch was the first IBM computer, at least, to be pipelined, though this was called "virtual memory," predating the use of that term to mean swapping to disk.

The idea of a "program" solving a "problem" for a "sponsor" is pretty far from our current understanding. So, too, the justification of "multiprogramming" (also "time-sharing"): "it becomes economically practical for a person seated at a console to observe his program during execution and interrupt it while considering the next step", and also to "avoid delaying the calculator [!] for input-output".

The disk storage (intended for 250,000 words per second, though only half of that was achieved: 8 megabits per second, seek time 150 ms; the terms "DASD" and "fixed disk" do not appear) was considered to avoid the need for "very-high-speed magnetic tapes."

"On line" and "off line" also appear, in quotes, to describe different ways of entering input data: on directly connected devices, or using physically separate devices that produced magnetic tapes, "the fastest possible medium".

"To an increasing extent, bits in even a scientific computer represent things other than numerical quantities: elements of a program metalanguage, alphabetic material, representations of graphs..."

A rather striking feature is that Stretch's memory, in 1961, had a 2.1-µs cycle time, which they considered rather slow (thus its 64-bit word size). Nowadays the main memory of my laptop (whose CPU uses a 64-bit word size) can produce bursts substantially faster than this, at close to 1 ns, but random accesses still cost it nearly 100 ns, which is only 21× faster.

Stretch could address 2^{18} words, but the first 32 words were its registers; I imagine this wasn't useful to user programs because of the memory-protection scheme explained below. Register 0 was the zero register, "a bottomless pit."

Memory protection was provided as follows: "The interpretation and execution of instructions is monitored to make sure that the effective addresses are within boundaries defined by two [address-]boundary registers."

Only floating-point math was bit-parallel; logical operations and fixed-point arithmetic were done by "the serial section" of "the arithmetic unit", which was not yet called the ALU (though "Instructions that combine bits by logical and, or, and exclusive or functions have been available in earlier computers."). This had the advantage that the operands didn't have to be word-aligned, or even byte-aligned, but could instead be at arbitrary bit offsets. "Byte-aligned" wouldn't have made any sense anyway — "The

number of bits used to encode individual characters may be varied. Thus a decimal digit may be compactly represented by a binary code of 4 bits, or it may be expanded to 6 or more bits when intermixed with alphabetic information."

It still had only a single accumulator (and 15 index registers) but the accumulator was two words, which I think means 128 bits! This despite extensive handwringing about how hard they worked to wring the most out of their slow memory (this was the reason for the pipelining, which prefetched).

The index registers were 64 bits wide even though memory addresses were only 24 bits ("A complete word-and-bit address forms a 24-bit number."). So they used the extra bits to count loop iterations and specify a "refill address" for reloading the register, and there's a "progressive indexing" mode "in which the index quantities may be advanced each time they are used".

Its indirect addressing mode was wild: "The term *indirect address* refers to an address that gives the location of another address. An indirect address may select an immediate address, a direct address, or yet another indirect address. Indirect addresses are obtained in the 7030 by the instruction LOAD VALUE EFFECTIVE, which places the effective address found at the specified memory location into an index register for indexing a subsequent instruction. Multiple-level indirect addressing is obtained when LOAD VALUE EFFECTIVE finds at the selected location another instruction LOAD VALUE EFFECTIVE which causes the indirect addressing process to be repeated."

The interrupt vector table was called "a table of fix-up instructions".

When talking about data formats for address arithmetic, the data formats shown are actually instruction word formats, suggesting that despite the ample index registers on the machine, self-modifying code was common.

One of the strange uses of "edit": "For example, a floating-point datum may be developed as a unit in a computation, its components then used in radix-conversion arithmetic, and the characters of the result finally used as units in editing for printing." I think maybe "edit" means "format" (although the Fortran statement is FORMAT, not EDIT.)

Aha, there's a definition: "A final class, *editing operations*, includes all operations in which data are transformed from one format to another, checked for consistency with a source format, or tested for controlling the course of the program." So "editing" includes comparisons for control flow!

Topics

The paradoxical complexity of computing the top N

Kragen Javier Sitaker, 2017-01-04 (7 minutes)

Comparison sorting is O(N log N) in the best case, using one of the standard sorts (mergesort, quicksort, heapsort, binary search trees, library sort, or a variant of one of these). In large datasets, the log N factor can reach 20 or 30, which represents a significant slowdown. But often people are comparison-sorting because they want the top M items: top 1, 10, 20, 30, or 100, say. Surprisingly, getting the top 100 items can be done in linear time (O(N)) using quickselect!

Quickselect is a version of quicksort that recurses only on one partition. If you have a million items, the first pass will examine all million of them; the second pass will examine, typically, half a million; the third pass typically a quarter million; and so on, until the 30th pass examines just one item. (I think there's a little bit of a fudge factor, as with quicksort, which uses 1.39 N log N comparisons on average; but I will ignore that for the moment.) This adds up to just under two million compares and half that many swaps.

When quickselect is done, each of the pivot elements it chose, including its final result, partitions the array into items smaller and larger than itself. So to get the top 100 elements, it is sufficient to invoke quickselect to select the 100th element, and then take the first 100 elements of the reordered array.

The paradoxical complexity of sorting the top N

This has a sort of paradoxical result. Suppose you have 256 elements and you want the top 64 of them. Quickselect can give you this top-64 result in about 512 comparisons and 256 swaps. But now if you want to comparison-sort *just those 64 elements*, so that you get the top 64 in order, you need 1.39 N log N comparisons, which is 534 comparisons, more than it took to find that top-quartile in the first place!

This gets worse as the problem size gets bigger, if we hold the quantile fixed. If you have four billion elements and you want the top quartile, you can get them in 8 billion compares, but fully sorting that top quartile will take you 42 billion compares, five times longer. At that scale, even the top eighth is faster to find than to sort fully once you find it.

Improving quickselect average-case time for extreme order statistics

Median-of-median is a way to improve quickselect's worst-case time (from quadratic to linear), but we can also improve the constant factor of the linear average-case time with one weird trick. Using a variant of the median-of-3 pivot approach, you can improve quickselect's linear-time average case by up to a factor of 2 by better pivot selection, especially when you're looking for a fairly extreme order statistic.

The improvement is limited to a factor of 2 because vanilla quickselect does less than 2 comparisons per element to start with, and computing an order statistic exactly will require examining each element at least once.

You do this by picking a first-step pivot that has a high probability of reducing the next step by as much as possible. For example, if you're looking for the 0.1% quantile of a billion items, you can sample 2000 candidate pivots and take, say, the 0.2% quantile from them (using about 4000 comparisons if you just use regular quickselect), which is very likely to partition the billion-item list into a very large list that does not contain the candidate item and a very small one (only about two million items) that does contain it. This reduces the problem size from the first step to the second step by a factor of 500 instead of a factor of 2, and so you will need on average only 1.004004 comparisons per element out of the billion rather than almost 2 comparisons per element.

This approach can, of course, be applied recursively, both in the pivot-selection step and on the reduced-size problem.

Also, since this approach depends on the position of the desired element within the reduced window, it may work to improve quickselect's performance significantly even for non-extreme order statistics. For example, if you're seeking the median, and your first-step pivot turns out to be very close to the median, then what was originally the median becomes an extreme order statistic in the first recursive call. Perhaps you're seeking the median of a million items, and your initial pivot turns out to be element #499000. Now the value you are seeking is element 1000 out of the 501000 you're selecting from in the next recursive call. With this approach, you can sample 500 or so candidate pivots, and probably manage to reduce your problem size down to about 2000 elements in the next step, ending up needing only about 1 506 000 comparisons (I think) rather than the nearly 2 million you would need with vanilla quickselect.

The expected improvement factor on any step is limited to about the square root of the number of elements. That's because finding a pivot to reduce the interval by some factor F is going to require selecting it from somewhere around F candidate pivots. (I've used 2F in the examples above; I don't know if 2 is the optimal factor.) But once you're examining more candidate pivots than the number of elements you expect to have remaining after partitioning with the pivot, you're not winning any more; you're losing.

I'm not sure what the optimal number of candidate pivots to consider is, and I'm also not sure about the optimal size of the range to try to consider. It seems like you'd want to have great assurance, generally, that your small range is big enough to include your target element in it, because otherwise you've just squandered an entire pass over the input data. As the desired order statistic wanders toward the median, though, the loss from such a miss diminishes, and the potential gain becomes smaller.

Topics

Coinductive keyboard

Kragen Javier Sitaker, 2016-07-30 (4 minutes)

The traditional way to scan a keyboard is with row and column lines which are connected through mechanical keyswitches. This runs into some trouble with bouncing and contact oxidation, as well as number of moving parts; a solution is to use variable capacitors instead, where pressing a key brings two pieces of foil close together, but not into contact, maybe squishing some plastic foam in between them or something. This prevents contact oxidation, since there is no contact to make sparks, and bouncing, since a small deviation in distance will produce only a small deviation in capacitance.

It still requires about $2\sqrt{N}$ wires for N keys, though. As an alternative, consider using selective inductive coupling instead of selective capacitive coupling: pushing a key pushes a ferromagnetic core through some coils of wire, greatly increasing the inductive coupling between those wires.

You could imagine, for example, using five "exciter" wires, each of which runs through 25 air-core coils, arranged in a 5×5 matrix. Each of these coils is physically near a coil on a "row" wire, which runs through the 5 coils on a row for this exciter wire and also the corresponding 5 coils on the corresponding row for each of the other 4 exciter wires, and a "column" coil, which has a corresponding column-wise arrangement. Normally, you have limited inductive coupling between the wires, but pressing a button inserts a ferrite core through all three coils, strongly coupling any AC signal on an exciter wire to a corresponding row and column wire. The signal is only weakly coupled to the other row and column wires. (A factor of 20 difference between coupling with and without the core should be easily achievable.)

In this cubic arrangement, 15 wires are adequate to scan a 125-key keyboard, while the traditional matrix arrangement would have needed 23.

But it's possible to go further with more coils: instead of coupling just three coils, let the inserted core couple, for example, five coils. An exciter current on one of three exciter wires can produce a current on four other wires, each chosen from a set of three; in this case, we have 15 wires scanning 243 keys.

It's also possible to go further with coil polarity: each coil can be wound in either of two different directions, resulting in an induced voltage with a phase difference from the exciter voltage. You could maybe also use different numbers of turns on different keys, but that seems like it is going to start to be flaky when the core is incompletely inserted. (That's the same reason to prefer using, say, five wires to scan five columns, rather than four.)

There's no reason, then, to leave some of the sense wires inactive on each keystroke. You could run each sense wire through each key with a coil in one or the other direction; for example, you could have three exciter wires which each excite 32 keys, plus five sense wires which run through all 96 keys, with a coil in one or the other direction, producing a unique 5-bit binary code for each of the 32 keys for a given exciter wire. This gives you 96 keys with only 8

wires. Since the codes are balanced, the stray inductive couplings through keys that are not being pressed should mostly cancel out.

That logic brings us inexorably to preferring only a single exciter wire, which brings us finally to 128 keys using a single exciter wire and 7 sense wires: 8 wires for 128 keys, a great improvement over 23.

If the sliding core slides through a hole in a high-permeability member that curves around to behind the coils, then it can close a magnetic circuit, reducing the total circuit reluctance by orders of magnitude, thus increasing the inductive coupling dramatically.

This keyboard design has the great advantage for klutzes like me that it contains no uninsulated electrical conductors and no tight clearances, so it should work perfectly well even if it's, for example, immersed in salt water or filled with Coca-Cola.

Topics

- Electronics (p. 3430) (138 notes)
- Physics (p. 3632) (119 notes)
- Keyboards (p. 3537) (5 notes)
- Input devices (p. 3525) (5 notes)

More thoughts on powerful primitives for simplified computer systems architecture

Kragen Javier Sitaker, 2015-08-18 (updated 2015-11-02) (165 minutes)

XXX transactions are useful for reactive programming because they prevent ordering problems?

In 2012 I wrote about "choosing powerful primitives for a simplified computing system", including as examples gzip and other things, and speculating about automated cache management. Well, since then, automated cache management has kind of gone mainstream in the form of "reactive programming" frameworks. What other things could drastically simplify our software systems?

My focus here is on what kinds of mechanism could support a wide variety of applications, eliminating complexity from the application code, without having to write and maintain separate, specialized versions of that mechanism in different parts of your computing system; and by so doing, make the system less complex, more approachable, and more adaptable to new situations. I'm looking for candidate components with a high "strength-to-weight ratio", in the sense that their internal complexity is low compared to the functionality they provide, and also a low "surface-to-volume ratio", in the sense that they impose very little extra complexity on applications that use them, compared to their own internal complexity, so the application is simplified greatly by not containing its own reimplementation of their functionality.

In particular, such mechanisms need to be acceptably efficient across their wide range of uses, rather than performance-optimized for one particular use. If this isn't the case, experience has shown that people will write customized versions of them for their particular application, rather than investing effort in further optimizing the general-purpose mechanism. The result is that specialized mechanisms proliferate, overall system complexity explodes without a corresponding increase in power, composability suffers, and none of the specialized mechanisms receives enough attention to make it efficient. This should sound familiar, since this is the tarpit we're in today.

I believe this matters for a few different reasons.

One is that, as Dan Ingalls says, to be a medium for personal expression, a system needs to be comprehensible by a single person. And this is the vision that has motivated Smalltalk, the VPRI STEPS project, and Chuck Moore's FORTH work. It unfortunately suffers a bit from mostly appealing to very independent-minded people.

A second reason is that this poor architecture results in decreasing returns to software work, which matters economically at a societal level. People's day-to-day lives are largely driven by what we might call economic factors: do you have to spend your time washing clothes by hand and sewing them up, or can you use machinery to make that easy? Is your house warm? Do you have running hot water and safe drinking water? What do you have to do to make a

living — does it compromise your health or moral integrity? XXX

A third reason is that software freedom is fundamental to individual human rights, to collective economic justice, and to individual economic justice in the 21st century, but the freedom to study and modify software you can't understand is only theoretical, and the power of that freedom is multiplied by the modifications and enhancements you actually can manage to make in your limited time.

Software freedom is fundamental to individual human rights because whoever controls the software controlling your cellphone, your pacemaker, your hearing aid. They can see photos of your genitals, if you ever take them, and from Snowden's revelations we know that some of them pass them around the office for laughs. They can trick you into thinking your friends hate you, that your husband is cheating on you, that the Jews faked the Holocaust. They can judge you based on what you read or who you're friends with, and secretly tell other people about those judgements, and those other people may be the police. Once you have the pacemaker, they can kill you, and nobody will ever know it wasn't natural.

Software freedom is fundamental to collective economic justice because it allows every country or group of people to prosper independently of the rest. Over time, we can expect the strong standardizing tendency of software to standardize on a small number of operating systems, a small number of web browsers, and so on. Without software freedom, that will result in enormous economic returns accruing to the vendors of the dominant software, which can then extend their monopoly power into new areas of software: perhaps Apple, Microsoft, Google, Facebook, and Amazon, but just as likely, a few new companies that you haven't heard of yet, probably in China.

Software freedom allows everyone to participate in production, while proprietary software moves us toward a world where, if you don't live in Seattle, Shenzhen, or California, you can only consume, or at best source "user-generated content", not produce or participate as an equal.

Many governments have responded to this harsh reality by trying to pretend that software is just one industry among many, and they can import software just as they import oil, while focusing their domestic industry on shipbuilding or steel production or something. But the truth is that your shipyards, your ships, and your steel mills will get most of their value from software — if not in 2015, then surely by 2035.

Software freedom is fundamental to individual economic justice because unless you were born to wealth, your other choices are joining the startup lottery, where almost all the participants end up worse off than if they hadn't played, and being employed, where almost all of the value you produce will be skimmed off by your employer. By contrast, when you write free software, nobody can ever take it away from you. In fact, even when *other people* write free software, nobody can ever take it away from you. (By contrast, when other people write free-in-theory-but-not-in-practice software, it can totally be taken away from you. Like if the maintainer decides to stop supporting your platform, or bundle some kind of adware you hate.)

Any economically productive activity that you can do with

software becomes more productive when there is more free software out there for you to draw on.

The World-Wide Web, for apps

The Web actually provides us with some tremendously powerful primitives that simplify a lot of applications, even as they make others more complicated. Out of the box, a webapp gets:

• a gentle learning path;
• a typographic layout engine with a relatively powerful box model and excellent Unicode support;
• image and sound decoding and caching;
• alpha compositing;
• a hierarchical UI event system;
• some degree of incremental relayout;
• a high-level reflective prototype-oriented OO language with closures and reasonably efficient implementations: JS;
• sufficient robustness in the event-driven programming environment that your app will probably still mostly work even if some features are throwing exceptions;
• a vector drawing library that's even easier to write in than PostScript, which is saying a lot: SVG;
• a compound document system where you can OLEishly embed one document in another, including mutually untrusting documents;
• a mobile-code system that downloads your webapp once and updates it on the client only when it's changed, and potentially only the parts that have changed (if you configure your server headers correctly or if you use a cache manifest);
• by far the best debugging tools I've ever used anywhere: Firebug and the Firebug clone that comes with WebKit;
• serialization of simple data for free with JSON;
• simple persistence (e.g. using localStorage);
• a SQL engine (although that's been deprecated);
• sending some text over the network to a server in a single line of code;
• network protocols that provide you fault-tolerance by default as long as you obey the REST constraints;
• transparent network protocol compression;
• a global namespace for remote objects that encompasses the actual physical globe, not just your program's "global" namespace.

I'm not sure that these primitives have a great "strength-to-weight ratio", in the sense of providing a lot of power for low implementation complexity, or for that matter even low interface complexity, but the difference between a JS webapp and, say, a Tk program to do the same thing is pretty big.

Consider this "microblogging" program in ksh, like a non-networked version of Twitter:

```
while read line; do
    ENTRY="$(date +'%d-%m %H:%M:%S'): $line" echo "$ENTRY" >> microb.log
    clear; cat microb.log; echo -n "sup-> "
done
```

Here's an equivalent in DHTML, with has better text layout,

scrolling, search, navigation, and editing, and automatically live-upgrades itself to the latest version of the code after each post, although it's a bit more code and I think it runs slower:

```
<script>var ls = localStorage, d = document;
function add(name, html) { ls[name] = (ls[name] || '') + html }
function esc(text) {return text.replace(/&/g,'&amp;').replace(/</g,'&lt;');}
d.write(ls['microblog'] || '');</script>
<form onsubmit="add('microblog',new Date+': '+esc(this.line.value)+'<br>')">
<input name=line autofocus>
```

If you take out the HTML-escaping code, the DHTML version also has rich text.

Going back to how we would do things in 1995, a Tcl/Tk version of the same thing would have to take care of a lot of these things by hand. I started writing it, in the same ridiculously compressed style as above:

```
#!/usr/bin/wish -f
text .blog; label .sup -text sup; entry .e -textvariable e
pack .blog -side top -fill both -expand true
pack .sup -side left -fill x; pack .e -side left -fill x -expand true
bind .e <Return> blog; proc blog {} {
    global e; .blog insert end "$e\n"; set f [open microb.log w]
    puts $f [.blog get 1.0 end]; close $f; set e ""
}
```

XXX include full version

This is still lacking a bunch of things, even compared to the shell version, including reading the microblog file at startup, not losing data in machine crashes, making the textarea itself non-editable (?), adding the dates, focusing the text entry, not being an ugly gray, and so on. And this isn't even hitting any of the web's strong points, like compound documents, networking, garbage collection, or SVG.

If you wanted to replace the web as an application platform --- which is certainly what both Apple and Google are trying to do with their respective smartphone operating systems, but also the intent of platforms like VPRI's STEPS --- you need to understand the strengths of the platform as well as its excessive complexity. Some of those strengths are in fact powerful primitives, like those mentioned above. Others are more subtle and in fact work against the VPRI vision of an overall system that is as simple as possible, such as the Principle of Least Power, the gentle learning path, and View Source.

Other candidates

What other powerful primitives might have a high "strength-to-weight ratio" in a computing system, in the sense of enabling us to build much more powerful systems with very small amounts of code?

Content-addressable blob stores for persistent, self-certifying trees

Traditional Unix-like and MS-DOS-like filesystems provide expandable virtual disks: "files," which are mutable sequences of

bytes, supporting random-access read and write, plus appending data at the end. Instead, a variety of more recent filesystems and related systems either do not permit modification of existing "files" at all, or only allow appending to them, and identify stored blobs by using a one-way hash of their contents; name-to-blob-hash mappings are typically stored in another blob. Systems using one or another variant of this design include the Xanadu Ent, WAFL, Venti, GoogleFS, FreeNet, Wax, IPFS, Tahoe-LAFS, Bup, Jumprope, Nix/Guix, git-annex, and most importantly, Git and BitTorrent.

This permits very efficient comparison of filesystem trees and storage of slightly-modified versions of existing trees, since deep equality can be performed by an inexpensive shallow equality test, just as with hash-consed in-memory persistent data structures.

IPFS's author calls this the "Merkle DAG" model: parent nodes link to their child nodes with secure hashes, just as in a Merkle tree, without the linearity requirement of the tree.

The one-way-hash "names" are "self-certifying" in the sense that the mapping from the name to the blob contents can be checked by the reader, so they do not need to rely on the storage provider. Another kind of "self-certifying name" consists of a public key or a one-way hash of one, although there are tricky issues related to this.

Avery Pennarun (the author of Bup) wrote a very compelling advocacy of the position that content-hash-addressable blob stores are such a powerful technology that they invert much of our conventional wisdom about data storage. XXX link it!

Could you replace most or all of these blob stores with a single blob store? Could you provide a single blob-store interface to write your application software on top of, allowing you to run it against any of these blob stores? (Git-annex already kind of contains such a thing internally.)

Distributed hash tables

These are worldwide key-value stores with some degree of resiliency against malicious participants, and much of the yearly [what is that thing where Boon Thau Loo published so many papers called? IPTPS?] workshops focused on versions of these. In concert with self-certifying names, they allow distributed blob stores to have new blobs added to them, and they allow participants with a common interest to exchange rendezvous information without depending on a central server.

The earliest DHT work was "consistent hashing" at Akamai, but later systems include Chord and Kademlia. BitTorrent uses a DHT to bootstrap "trackerless" torrents.

DHTs are mostly only used for decentralized systems (that is, systems to which it is infeasible for any party to deny access to any other party), but their failure properties in the face of malicious participants are not well understood.

Is it possible to use a single DHT implementation for many different applications?

Persistent data structures

Unfortunately, "persistent" has acquired two confusingly similar and conflicting meanings, both of which have "ephemeral" as their antonym, because most people are illiterate troglodytes who never

venture outside their cave to see what's happening in the larger world, so they invent confusingly conflicting terminology. In this case, I'm not talking about your program's data surviving restarting it or restarting your computer (because the data is on disk or something), but rather data structures which might be in volatile RAM, but which you don't modify. Instead, you make new, modified versions of them. If you use efficient persistent data structures, most of the structure will be shared. This is the kind of "persistence" that pertains to most of the standard data structures in, say, Clojure.

This pretty much requires a garbage collector or its moral equivalent, like reference counting or a linear type system. RAII is not going to help you out here. This also means you have no idea why your program is using so much RAM.

There are a few different advantages to persistent data structures over their "ephemeral" cousins, the kind where you modify them in place. One is that they're less bug-prone: no more aliasing bugs where you passed a dict to some function and then to another function and then the first function modified it. Another is that you get "undo" without any extra complexity in your code. This is pretty important if you're, say, trying to do software transactional memory (see below) or backtracking. A third is that if you actually are using a bunch of different versions of a data structure (like in the backtracking case) they'll probably use *less* memory. A fourth is that you can access them without locking, and replicate them without fear (whether across the CPU caches of your different cores or halfway around the world) so they can give you better efficiency. (They're a crucial enabling technology for CRDTs; see below.) A fifth somewhat subtle performance point is that, if you *are* using a garbage collector, they prevent you from wasting CPU time on its write barrier, while mutable data structures have you beating on the write barrier all the time.

I mentioned above that an immutable blob store is in a sense a store for persistent data structures. Hopefully that is clearer now.

"Hash consing" is a technique applicable only to persistent data structures: when asked to create a given data structure, you hash it to see if it already exists, and if so, return a pointer to the existing instance rather than making a new copy. This allows you to perform deep-equality testing in constant (and small) time. Content-hash-addressed blob stores provide this feature, but only if you are careful not to include any extraneous data.

Clojure's standard library already includes a wide variety of persistent in-RAM data structures that are simple to use and useful to many applications. Could these be abstracted so that the same code also supports distributed and blob-store cases?

Pub-sub

Publish-and-subscribe systems, also known as event-based integration systems or event buses, allow you to connect together very-loosely-coupled subsystems with real-time notification. These are universal in financial institutions, but are also used in systems at a variety of scales to reduce coupling between modules.

Typically, when a publisher publishes an event on a topic, all the subscribers to that topic receive a copy of the event, shortly afterward or possibly not at all. The publisher is not informed of the success or

failure, and does not know how many subscribers exist.

The use of this technique in user interface programming goes back at least to MVC in the 1970s, where the model would notify all of its views that it had changed and they might need to redraw.

As a current and local example, when I save this text file in my editor, the kernel sends an inotify notification to any processes that have registered to receive notifications for this file; this can trigger reindexing it in a full-text-search engine, updating a filesystem browser window, or output of extra lines from tail -F. And of course group chat systems like IRC, Slack, and WhatsApp are a very direct application of pub-sub; and D-BUS is the center of a modern Linux desktop, for better or worse, mostly worse.

Usually, these systems are capable of losing messages, because the desirable decoupling they provide between subscriber and publisher prevents them from applying backpressure to slow down a publisher who is publishing messages faster than some subscriber can process them. (It would, after all, degrade service to all the other subscribers as well.) Lampson's Hint that a system must shed load to keep functioning in the case of overload constrains the overall system to lose messages in this case; typically this is done by discarding extra messages at an event-bus-providing server and possibly notifying the subscriber that messages have been lost. In cases where the messages are being used for cache invalidation, recovery typically involves an expensive re-replication of state.

(Content-addressable blob stores can make re-replication much less expensive! So can rolling hashes, mentioned next.)

One particularly simple way to solve that particular problem is to make each subscription one-shot: to receive another message, you must renew your subscription.

Windowing systems almost universally support some kind of pub-sub notification, though sometimes synchronous and even with a reply capability built in.

Could we replace the wide variety of pub-sub mechanisms in use, often intertwined with a horrific amount of application-specific policy, with one or a few pub-sub mechanisms?

Rolling hashes, or “the rsync algorithm”

Rsync efficiently discovers duplicate file chunks between two replicas of a file, even in the face of insertions and deletions, by hashing one replica of the file by aligned blocks, then computing a "sliding" or "rolling" hash over all possible blocksize-sized blocks in the other replica, whatever their alignment or overlap with one another. This allows it, like magic, to transmit only the missing data over a (presumably slow) connection, despite not being able to directly compare the two versions of the file.

This same approach is used by rdiff-backup to efficiently find a small delta to apply to a potentially large file, as well as by bup (for a similar purpose), zsync (a replacement for rsync that precomputes the block hashes on the server side and therefore requires no run-time computation on the server side, permitting efficient synchronization from a dumb blob store), Zbackup (an alternative to bup), ssdeep for computing and recognizing malware signatures, Jigdo, and Jumprope.

Could we support all of these rolling-hash-based applications with a single distributed data structure?

Backtracking or nondeterminism

From the depths of the 1970s, Prolog and SNOBOL come at thee! Nondeterministic programming is a lot like normal programming, except that parts of your program can "fail", and then if you've provided an alternative path, they will take it. The most everyday version of this is the regular expression:

```
decode_pattern = re.compile("(.{11}) (\\d+)<([^>]*)>(\\d+)\n")
```

This example, taken from a project I'm working on called Gab, matches lines like `m0oTzNujJpx 6<I=EInw>6\n`.

If you're not familiar with regular expressions, the "\d+" means "look for as many digits as possible without making the pattern fail". If the pattern fails afterwards, the "\d+" will try matching fewer and fewer digits, until it can't match any, at which point it will itself fail.

In one sense, this particular regexp is not a particularly great example because it's carefully written to be deterministic: it can never backtrack and then succeed. If we were to replace the "[^>]" *with* ".", so that it could match any string at all and not just a string that doesn't contain the ">" symbol, then it would start by matching to the end of the line, then fail since it didn't find a ">" there, and backtrack until it did find one. In another sense, this is a fantastic example of how, in practice, we work very hard to keep the poor performance of constant backtracking under control when we use nondeterministic constructs.

Nondeterminism implemented by backtracking is fantastic for parsing: you write down a program that could *generate* the language you're trying to parse, and use nondeterminism to "execute it backwards": try all the possible paths of execution until you find one, or the one, that generated the string you're parsing. PEGs, which are the most interesting advance in parsing algorithms in a decade or two, have a very simple semantics in terms of backtracking. The "packrat" PEG-parsing algorithm guarantees linear-time parsing by memoizing partial parsing results.

Database queries are easily conceptualized as nondeterministic programs. `SELECT FOO.A, BAR.B FROM FOO, BAR WHERE FOO.C = BAR.C AND BAR.D > 10` has a very simple interpretation as a nondeterministic program.

The everyday "diff" is also easily conceptualized as a nondeterministic program, but in this case it isn't sufficient to find *some possible* set of edits to get from the old version to the new version; we want *the smallest* or *nearly the smallest* set of edits. And the standard quadratic dynamic-programming LCS algorithm to solve it can be understood as a way to tame the exponential blowup of choices with memoization (see below).

There are a large number of different ways to rein in the exponential complexity blowup that seems inherent to nondeterministic programming, mostly specific to one or another domain. Packrat parsing uses memoization, limited context, and implicit limits on backtracking; practical Prolog programs use cut; Datalog uses stratified resolution, which is a lot like memoization; truth-maintenance systems note the proximate causes of failures; SQL systems use indices and hash and sort-merge joins; parsing systems in general use lookahead; regular-expression engines sometimes use DFA compilation, Boyer-Moore search for constant strings, and tables of the locations of improbable bytes in the string,

and can also use suffix-table indices (see below); and so on. It would be very useful to have a unified framework that avoids all this duplication of mechanism, and moreover can be applied to make nondeterministic execution in new domains reasonably efficient without needing to invent another special-case hack.

Might such a general efficient nondeterministic-computing algorithm exist?

Memoization

Several of the wild exponential nondeterministic domains are tamed by memoization; in the Packrat case, all the way down to linear time. (You could also view the lazily-constructed DFA approach to regexps as being memoization to get linear time.) Memoization is storing the arguments and return value to a function in a "memo table" so that they can be returned next time without recomputing the function. At the best of times, it magically makes your program go faster, sometimes astronomically faster in the presence of recursion. It is a kind of computation caching, and it's also central to the Self-Adjusting Computation paradigm of incremental computation that I'll mention later.

Memoization in practice can be tricky to win at, due to both false misses and false hits in the memo table, and also for efficiency reasons.

False misses occur because the function is being invoked with data that it does not care about, which is not always obvious; perhaps it depends not on the exact value of some numeric argument, for example, but only on the number of digits in it. This can be fixed by dividing the function into two functions, one of which reduces the input data to the necessary, but this modification seems artificial without the context of memoization.

False hits occur because some data not considered for the memo-table lookup affects the return value of the function, or because the function has some other effect. If the function has arguments that refer to some mutable data structure, mutating that data structure between calls to it may affect its results such that you wouldn't want to reuse the memoized result.

Efficiency is tricky for a variety of reasons. If the function is a thin wrapper around some other function, memoizing one or the other of them will probably give you the benefit of memoizing both of them, at half the cost. Looking up the arguments in the memo table can be expensive if the arguments are large data structures and you aren't using hash consing. If the arguments are mutable, you may have to copy them. The memo table can use up a lot of memory. If the function is rarely or never called twice with the same arguments, memoizing it will only make it slower.

Under some circumstances it may make sense to share a memo table across runs of a program or even across a whole distributed system. Indeed, this is a major function of build systems like `make` and `ccache`, but it could potentially be useful for smaller computations as well. I read a paper about using a precomputed distributed memo table of optimizations to enable absurdly aggressive compiler optimizations, for example.

Could we provide generally useful memoization with one or a few memoization mechanisms, orthogonal to rather than mixed into the

code being memoized?

Memoization is closely related to deterministic rebuilding (see below).

Monads

There are already too many tutorials on what monads are, so I would not try to explain, even if I knew. I just want to point out that sometimes you can write a function with one monad in mind (lists, say) and then run it in another one (for example, backtracking, although in a lazy language there may not be so much difference).

How often do we write code that is unnecessarily coupled to a single monad when it could instead be reusable across different monads?

Constraint solvers and logic programming (like SQL, but more so)

I think I mentioned this in my original post (XXX did I?): SAT and SMT solvers are now powerful enough to replace a fairly wide variety of custom code in things like compiler optimization. Maybe they could be the unifying approach to nondeterminism that I was saying was needed! I don't know enough about them, though.

They are also a crucial enabling technology for formal methods: you can use them to find test cases that will break your program, or prove that there are no such test cases.

Unfortunately, I have no experience with SAT or SMT solvers, so this is very speculative!

How widely can we apply SMT solvers to simplify our software? What's the simplest SMT solver that's efficient enough to be useful in common cases?

Truth maintenance systems, due to Stallman and Sussman as refined by Doyle, are a kind of nondeterministic constraint solver that notes what sets of assumptions have led to inevitable constraint violations (through a kind of relevancy logic), improving the performance of the search by large exponential factors by cutting off nonviable search branches early.

This is a key aspect of modern SAT and SMT solvers.

Prolog was the first logic programming language, introducing nondeterministic programming, and can be viewed as a kind of constraint solver. It led to a lot of excitement in the 1970s and early 1980s as a general declarative system, but practical Prolog programs have to intersperse evaluation-strategy information rather intimately with declarative information in order to achieve usable levels of efficiency. Consequently, Prolog and similar systems such as KL1 failed to fulfill the high hopes many had had for it, and in particular caused the failure of the Japanese Fifth Generation Computing Systems project, resulting in part in Japan's continued failure of development in the software field.

Will Byrd and his colleagues have been working on a new family of logic programming languages named Kanren, whose smallest member, μKanren, from 2013, is 39 lines of Scheme, but even miniKanren is only 265 lines. A version of miniKanren has been incorporated into Clojure's standard library under the name "core.logic". miniKanren's constraint solving is powerful enough to automatically enumerate, for example, quines or programs that

produce a particular output, and there is a theorem prover written in a variant called αKanren which can not only search for a proof for a given theorem, but also the theorems that can be proved from a given set of postulates, including theorems matching a particular pattern.

As the old miniKanren page explains:

> KANREN is a declarative logic programming system with first-class relations, embedded in a pure functional subset of Scheme. The system has a set-theoretical semantics, true unions, fair scheduling, first-class relations, lexically-scoped logical variables, depth-first and iterative deepening strategies. The system achieves high performance and expressivity without cuts.

PrecoSAT, Armin Biere's SAT solver, apparently was a competitive-performance SAT solver in 2010; it is only about 5300 lines of C++.

Array languages like APL

In APL, I can write the following:

```
D ← A + B × C
```

and it can correspond to any of the following in C:

```
D = A + B * C;
for (i = 0; i < n; i++) D[i] = A + B * C[i];
for (i = 0; i < n; i++) D[i] = A + B[i] * C[i];
for (i = 0; i < n; i++) D[i] = A + B[i] * C;
for (i = 0; i < n; i++) D[i] = A[i] + B[i] * C;
for (i = 0; i < n; i++) D[i] = A[i] + B[i] * C[i];
for (i = 0; i < n; i++) D[i] = A[i] + B * C[i];
for (i = 0; i < n; i++) D[i] = A[i] + B * C;
for (i = 0; i < n; i++) for (j = 0; j < m; j++) D[i][j] = A + B * C[i][j];
for (i = 0; i < n; i++) for (j = 0; j < m; j++) D[i][j] = A + B[i] * C[i][j];
```

and so on. That is, not only does the APL code avoid writing the loop out explicitly; it *abstracts over whether there's a loop at all*, as well as how many levels of loop there are, allowing you to use the same code in the loop case and in the loopless case.

In many cases in C, we would actually write something like this instead:

```
int D(int A, int C) {
    return A + B * C;
}
```

perhaps using a global constant B, in which case the APL has saved us the overhead of a function definition --- not only syntactic, but also mental. Note that if B stops being a constant, we need to modify the argument list.

Given how often we find that something we thought was a constant is in fact a variable, this seems like a potentially very useful decoupling.

Of course, vector languages like APL, Octave, and R are in some sense very much stuck in the 1960s: you have integer array indices all over the place, with no safeguard to keep you from accidentally indexing a nonsensical array with them; no useful garbage collection of indices is possible; and accidental performance bugs are ubiquitous.

Note also that APL is not capable of interpreting `D ← A + B × C` as the following:

```
for (i = 0; i < n; i++) for (j = 0; j < m; j++) D[i][j] = A + B[i] * C[j];
```

--- a limitation which is, to me, a direct consequence of the unprincipled and undisciplined hot integer-index action ubiquitous in array languages. If the earlier-mentioned varying interpretations are in fact valuable, this one seems certain to be valuable as well. But APL requires us to explicitly call it out as `D ← A + B ∘.× C`, since otherwise it has no way of knowing that the indices of B and C are semantically orthogonal, unless they happen to be of different cardinalities, in which case it barfs.

Despite this, it is common for a function written with Python scalars in mind to work correctly elementwise on parallel Numpy vectors, or one written for vectors to work correctly on matrices.

I feel that there is a very close connection between these vector-valued variables and the table-valued variables of SQL, or the variables in backtracking languages like Prolog which may have some arbitrary collection of values during a single repeatedly-backtracking invocation of a predicate (or function, if you're not in Prolog). Each of these unusual semantics allows a single statement to be polysemically interpreted as an operation over an arbitrary-sized manifold, or just a single point on that manifold, according to context.

Array languages explicitly expose parallelism that is implicit in other languages, and they directly provide operations such as reduction and scan which have nonobviously parallel algorithms available. This has driven efforts to distribute array-language expression evaluation across clusters and to vectorize it on GPUs and using SIMD instructions.

Despite all this, array languages have serious obstacles in their way: their effective lack of type-safety makes it difficult for them to provide useful error messages, or often any error message rather than a numerically incorrect answer; their unprincipled nature often converts programming in them into puzzle-solving, and limits the dimensional decoupling that is achieved in practice; and their potential for brevity is a seductive but fatal temptation. To use them effectively, you more or less have to represent your data with parallel arrays, and those are unfashionable nowadays in part because of the lack of type-safety mentioned above, but also due to ever-present consistency bugs under update and clumsy support for dynamic and local allocation.

(I have an unfinished essay coming up where I compare and contrast the dominant memory models of programming languages, which conveniently all can be identified with one or another programming language from the 1950s: FORTRAN's parallel arrays, COBOL's nested records, and Lisp's object graphs. Array languages are firmly in the FORTRAN camp, although Numpy has been adding record types.)

Is there a formulation of array languages that would be broadly useful to many different applications, including exposing hardware parallelism in an easy-to-use form, without compromising the

comprehensibility of the application code?

Probabilistic programming systems

We can treat parsing by nondeterministic backtracking as attempting to simulate the execution of a program that could have generated the string we are trying to parse, conditional on the actual contents of the string --- we backtrack when the output conflicts with an observation. Probabilistic programming systems are a generalization of this paradigm, or from the other side, a generalization of hidden Markov models: we begin with a prior probability distribution in the variables that we are trying to estimate, and we update it according to observations of the actual facts.

A recent paper in the area by Kulkarni improves the state of the art in certain difficult computer vision problems, and matches it in others, by estimating the probability distribution in a simple probabilistic image-generation program: https://mrkulk.github.io/www_cvpr15/1999.pdf

There are a number of probabilistic programming languages now available, but I've never tried any of them. Could you use some kind of probabilistic computing system to replace backtracking nondeterminism in general? Does it have a hope of being efficient? (This seems to be what Oleg Kiselyov and Yukiyoshi Kameyama did by implementing a logic-programming system using Oleg's probabilistic-programming system Hansei, in their paper "Rethinking Prolog".)

Could you avoid coding the Viterbi algorithm for HMMs?

miniKanren (see above, also written by Oleg, among others) has been extended to probabilistic logic programming as probKanren, using Markov Chain Monte Carlo evaluation.

Incremental or self-adjusting computation

Umut Acar's self-adjusting computation algorithm is a mechanical transformation you can apply to a batch algorithm to get an incremental algorithm (one which, given a set of changes in its inputs, propagates them to changes in its outputs) that is in many cases optimally asymptotically efficient. He uses ubiquitous memoization of a CPS-transformed version of the program, I think with hash consing, to do this, and uses a "virtual clock" to ensure that side effects do not invalidate the memoization when re-executing functions whose inputs have changed. (I think. I haven't managed to finish his dissertation yet.)

That is, the idea is that you run the transformed algorithm once to get its output, and then to update the output for a changed input, you re-execute only the parts of the program that are affected by the changed input. He's demonstrated speedups over the raw batch algorithm of several orders of magnitude, in exchange for a slowdown of a factor of 5 or so while running the trace of the initial execution.

Hammer has been extending this work, and Jane Street recently published their implementation of SAC for OCaml: https://blogs.janestreet.com/introducing-incremental/

This is a very interesting kind of decoupling: your code is decoupled from whether it is being used to compute the entire output or only a change to it. This is potentially very useful not only for efficiency but also for debugging ("What parts of the program would

change their results if this input changed?", leading to "What part of the input affected this part of the output?") and for dynamic program dataflow analysis in general. You could imagine using such an execution trace to replace a scalar input with a vector, as in the array-languages item above, and to rapidly provide feedback to optimization algorithms which want to know which input changes will improve their utility function.

Can we apply this kind of incrementalization to programs that were written without it in mind, as a compiler for nonstandard semantics? (I think that's been done!) Does it subsume backtracking, if the decisions taken at nondeterministic backtrack points are treated as input? Can we do all of this efficiently without using tens of gigabytes of RAM, perhaps by making more judicious choices of what to memoize?

Fully homomorphic encryption

A lot of effort has gone into developing very clever cryptographic protocols to protect parties from one another in ways that would not be needed if they were willing to rely on the integrity of a common "trusted entity", traditionally called "Trent", because any person or machine you nominate as Trent is in fact corruptible. As Szabo explains [XXX], FHE allows you to collectively create a "virtual Trent" that carries out any agreed-upon computation whose correctness and confidentiality all participating parties can verify.

That is, if you have a practical FHE protocol, that protocol would subsume essentially all cryptographic protocols, in theory.

The problem now is how to make FHE sufficiently efficient to be practical for anything.

Convergent replicated data types

Also known as "CRDTs".

Eventually-consistent databases have been a topic of some interest ever since some PARC work in the 1990s argued that ACID was wrong for many applications, although of course Lotus Notes was an eventually-consistent database since its inception in the early 1980s. Interest in them exploded after Eric Brewer's CAP conjecture (later theorem), which showed precisely how high the price of ACID was, and about half of the NoSQL fad, including the influential Amazon Dynamo paper, which I STILL haven't read, was an exploration of the eventually-consistent design space.

The problem is that with Notes, or Riak, or the git plumbing, the database doesn't reach consistency on its own. It just tells you that you have a conflict and you need to resolve it. It's up to your application code (in the case of git, the porcelain) to resolve the conflict, and probably in some cases to throw itself upon the user's mercy so they can resolve the conflict. Even if the application contains ad-hoc application-specific code to resolve the conflict, that code usually gets tested pretty rarely, and it's not at all unusual to have subtle bugs in it that will sometimes resolve a given conflict incorrectly.

CRDTs are a way of resolving update conflicts between different replicas that come with proofs of convergence: if your application (or database!) uses a CRDT algorithm to update a particular replicated data store, the merging process is guaranteed to always converge,

always to the same state regardless of the order of merge operations, and generally without losing any data (although that depends on which CRDT!) That is to say, "eventually consistent" becomes an automatic guarantee, not a statement of hope.

CRDTs offer an automatic way of replicating data to keep it available in the face of node or network failure while compromising consistency guarantees to a lesser degree.

A very simple CRDT which is foundational to most of the others is a set or bag to which things can only be added, not removed. Conflicting updates are simply resolved by taking the union.

AI optimization algorithms

SAT is, in a sense, the problem of inverting a Boolean function. You have a big propositional expression, which you know evaluates to true; you want to know what the values of the variables in it are, or at least whether there is any possible set of values for those variables. Similarly, nondeterministic and probabilistic computation can be viewed as other function-inverting problems.

Optimization algorithms solve a related problem: they are trying, more or less, to *approximately* invert a *black-box* function, a function whose behavior we hope is in some sense orderly. This is harder than SAT because we don't get to look at the code for the function, but also easier because the algorithm gets some indication of how close it is.

(Actually they're usually trying to maximize or minimize the function.)

There's a whole huge family of numerical optimization algorithms from AI: random sampling is the simplest, of course, but then you have hill-climbing (gradient descent), the simplex method for linear parts of the problem, simulated annealing, genetic algorithms, and so on. If you relax the black-box constraint and make your function differentiable (see automatic differentiation below), gradient descent becomes enormously easier, and the possibility of using something like Newton's Method to try to find zeroes opens up. And, since optimization typically involves re-executing the same code repeatedly with slightly different inputs, self-adjusting computation may be able to speed it up by orders of magnitude.

It is very common for it to be easy to evaluate the goodness of a solution to a problem, but algorithmically tricky to efficiently find a good solution. Consider the particle-system text layout algorithm Alan Kay was so excited about in VPRI STEPS --- wouldn't it be better to just specify what a good paragraph layout looks like, then search for one, as TeX does? Photogrammetry, structure-from-motion, point cloud coregistration, G-code planning for a 3-D printer, image dithering, lossy image or audio compression, lossless compression, edit-sequence computation (text diffing), place and route --- all of these are problems that can be easily cast into the numerical optimization mold.

Michał Zalewski's afl-fuzz combines genetic algorithms with compile-time instrumentation (see below) of object code programs to search through their execution paths, with the primary objective of finding bugs that lead to crashes, but has demonstrated the capability to generate valid SQL statements and JPEG files by probing SQLite and libjpeg.

How much can we abstract out these optimization algorithms to decouple them from the details of particular problems we are solving with them? How much code could we avoid writing if we can simply provide a computable utility function to optimize and apply a generic optimization algorithm to it? Can we make that more efficient by automatically incrementalizing or differentiating it? What's the relationship between SMT solvers and optimization algorithms?

Graph search algorithms

A* search is a provably-optimal graph-shortest-path algorithm, which takes an "admissible" heuristic to possibly improve its choices, usually allowing it to beat the heck out of Dijkstra's algorithm in performance. If the heuristic is admissible but not useful (for example, always returning 0), it decays to Dijkstra's algorithm in the worst case. It's commonly used for pathfinding in video games, and commonly with an inadmissible heuristic, which can cause it to find suboptimal paths. Amit Patel has a fantastic presentation of it.

Many search problems can be conceptualized as graph search (or tree search, since a tree is just a graph that is sparsely connected in a certain way). It's necessary that the choices be discrete, or discretizable.

What's the relationship between A *and the optimization algorithms discussed in the previous section? Can you use A* search efficiently for problems like dithering an image or superoptimizing a basic block?

Particle filters

Particle filters attempt to approximate an evolving continuous probability distribution by sampling it with hypotheses or "particles" that concentrate in the more probable parts of the space under the information you have so far, allowing multimodal probability distributions. Probably one of the various demos of this dynamical process on the web is better than some text at explaining it.

Particle filters are currently being successfully applied to a variety of hard perceptual or inference problems, including beat-tracking in music, motion-tracking in video, robot localization from noisy sensor data in confusing environments, 3-D object tracking, and automatic electrical fault diagnosis.

The same "importance sampling" mechanism used by particle filters also underlies some probabilistic programming systems (see above); for random variables drawn from a continuous distribution, even the random continuous drift that allows particle filters to explore points nearby in a continuous space might be useful.

XXX I should probably see if I can find people relating these two things to each other and make sure my understanding of the terms is right.

What else can we use particle filters for? Software-defined radio filtering, clearly: the frequency, phase, and direction both of the desired signal and of the loudest sources of nearby-frequency interference should be quite amenable to particle-filter estimation.

Transparent persistence, or single-level store

This is the other kind of persistence: storing your data on disk. If you've programmed an AS/400 or Squeak or run things inside VirtualBox (see virtual machines, below), you already know what this

is like: you need to explicitly separate your long-term persistent data from your ephemeral data in order to support code upgrade, but you don't need to make that separation just in order to not lose data when you turn the machine off.

Transparent persistence allows you to build a computing system without building a filesystem, and it supports fault recovery and process migration, and in particular it simplifies reasoning about capability systems, but it puts some extra demands on other parts of your system. Random number generators (like /dev/urandom) are not secure after restoring the system state from a snapshot (see below about snapshots). Your garbage collector needs to be comfortable with the idea that you have terabytes of data.

Without snapshots (see below), transparent persistence on spinning-rust disks poses certain problems of consistency: disk operations just before a power outage may have been reordered and some arbitrary subset of them may be lost. There's no guarantee that, when the machine reboots, the on-disk state will be consistent.

This particular powerful primitive is in no way new, forming as it does the bedrock of Lisp, Smalltalk, and AS/400 systems since the 1970s. In those systems, it does simplify some aspects of software, but it may be that it is a feature that doesn't pay its own complexity cost, except when used to implement features you can get no other way. Part of the problem is that most of the difficulty of persistence is not, in fact, the need to write serialization and deserialization code, but rather the need to distinguish irreplaceable data from dispensable data and to carry forward the irreplaceable data to new versions of the software. Rails migrations and Spark RDDs (see "deterministic rebuilding", below) may turn out to be bigger advances in persistence than single-level stores were.

Ropes

Lots of programs manipulate lots of strings, some spend most of their time manipulating strings, and network packets and the current states of disk files, memory regions, entire disks, and all of memory are also strings. With the usual terminated or counted array string representations, every string operation implicitly takes a potentially unbounded amount of CPU time, and many common operations, like replacing characters in the middle of a large string and appending large strings, do in practice take significant amounts of CPU time. And the standard string representation is not persistent (in the functional-programming sense; see above).

There's a heavier-weight representation of strings called "ropes" which *is* persistent, and in which most common operations are logarithmic-time or constant-time; in particular, concatenation is usually constant-time, although this varies a bit. A rope is either an immutable array of bytes (or characters), a concatenation of two or more ropes, implemented with pointers to them, or sometimes a pure function that can be invoked to retrieve characters within that range. You need to memoize (see above) subtree lengths to ensure that indexing is logarithmic-time. With hash consing, ropes enable string comparison to be done in constant time, too, which is particularly beneficial for memoization (see above). (I think rolling hashes are necessary and sufficient to make hash consing of ropes efficient.)

This is useful for two reasons: first, because string manipulation is

so expensive that we add a lot of complexity to our program designs in order to avoid it, and ropes may offer the opportunity to avoid that; and, second, because memoization is a very-broadly-applicable optimization.

Jumprope and bup are two rope-based storage systems that store their nodes in content-addressable block stores (see above) using rolling hashes (see above). Rope-based disk storage is particularly helpful for machine state snapshots (see below) which are often large, with a great deal in common with previous versions of the snapshot.

Erlang uses a very minimal kind of rope for I/O, called "IO lists": the various byte output routines accept "IO lists", defined as either a "binary" (Erlang blob), a byte, a Unicode binary, a Unicode character, or a cons of IO lists. This gives it constant-time concatenation among strings destined for output, which is enough to make it very fast at generating network packets.

I've started working on a non-persistent (in the functional sense: not immutable) variant of rope I call "cuerda", which I think has the potential to replace not just native string types but also filesystems, editor buffers, and most parse trees, including things like the accursed DOM. (I would prefer it to be persistent, but I haven't found a reasonably efficient way to provide certain other functionality I deem essential on persistent ropes.)

In a sense, WAFL and other modern filesystems like it are ropes-for-storage, although they usually embody tree structures, not simple linear strings.

XXX mention stratified B-trees?

What if some kind of rope were the normal kind of string, and also your filesystem? Could that simplify the whole system? (With a single-level store (see above), they could be the very same tree structure, although that might be undesirable for performance reasons on spinning rust, due to how it demands much more locality.)

Snapshots of state, ephemeral and durable

Unix creates new processes with the `fork()` system call, which results in two processes that are identical in nearly every way, each with its own private virtual memory space. This used to be a fairly slow operation, especially before 3BSD, when it involved physically copying the process's entire memory space. However, modern Linux machines can carry it out thousands or even tens of thousands of times per second, because the virtual memory mappings are merely marked copy-on-write.

afl-fuzz takes advantage of this efficiency to test thousands of random test inputs per second. OCaml's debugger uses the same facility in its debugger to "freeze" process execution at various points in the past, allowing you to step time *backwards* as well as forwards.

Similarly, QEMU, VirtualBox, and Xen support freezing the entire state of a virtual machine (see below), including its display, memory, and virtual devices, allowing you to back up to that checkpoint and resume execution from it later. Emacs does the same thing to speed up startup; Squeak and other image-based Smalltalk systems do the same, as mentioned above under "transparent persistence". Xen's "Remus" feature maintains a continuously-updated snapshot of the entire virtual-machine state on a remote machine, so that in the case of a hardware failure, you can

continue executing on the remote machine without rebooting.

WAFL avoided (avoids?) filesystem corruption on power loss by only committing internally-consistent snapshots of the filesystem to spinning rust, once per second; as a bonus, the snapshots share storage, and so old snapshots are accessible. This permits safely backing up a consistent snapshot of an active RDBMS transactional store. The same functionality is provided by ZFS, ext3fs jbd, and various LVM systems, which can be used as layers underneath other filesystems and transactional stores to ensure that recovery to a consistent state is possible, both after a power failure and after a failed system upgrade.

Git, of course, snapshots the source files it controls in your repository as a commit, storing them in its Merkle DAG in its content-addressed blob store (see above), which in turn is a CRDT (see above). Nix is a package manager and Linux distribution (with a GNU/Linux variant called Guix) that uses the same approach for configurations of installed software.

Some caution is needed: although snapshotting automatically converts ephemeral data structures (in the functional-programming sense) to persistent ones, they may not be *efficient* persistent ones, either in time or in space. If you're appending to an amortized-constant- time growable buffer, for example, and you snapshot its state just before it reallocates to twice the size and then repeatedly replay forward from that snapshot, it will rapidly become inefficient. And if you're snapshotting an object that references mutable objects, you need some way to convert them to immutable objects (and back to mutable objects when you revivify them), which generally involves copying, spending both space and time.

What if you could efficiently snapshot any part of your running system, ephemeral or persistent, at any time, and freeze it and possibly duplicate it for later? Could you implement this efficiently for mutable object graphs with help from the garbage collector's write barrier? How broadly could you apply this principle? (Maybe you could use this feature to implement the clone() operation for a prototype-based object system.) If you apply it to real-time mutable data structures, do you get guaranteed-efficient persistent data structures? What if you had Git for in-memory objects? What if all your software and data could be frozen and easily migrated between low-power handheld devices and more powerful laptops, or incrementally between data centers?

Virtual machines

Unix processes are virtual machines, although their I/O architecture is not borrowed from any physical machines. Unfortunately, they have access to so much system-wide state that it is often inconvenient to do what you want to do inside the confines of a single process; and there are programs you may want to run that are written using some other I/O architecture.

Whole-machine or near-whole-machine emulation, as with QEMU, VirtualBox, or Xen, is now a common system administration technique again, as it was in the 1960s; it allows you to snapshot the machine state (see above), to pause the machine's execution, to migrate it to a different host, to restart it from a snapshot, and to indirect its I/O through some virtual device. It's still very computationally expensive. In the other direction, chroot and

similar process- isolation mechanisms in Linux and FreeBSD are being beefed up, with things like Docker and the Nix/Guix isolated build environment (see below about deterministic rebuilding) to allow them to do more of what whole-machine emulation does.

Perhaps more interestingly, MirageOS is a set of libraries for building "operating systems" that are really just application processes, running bare on top of a Xen or similar hypervisor, thus reducing the security attack surface of your system by eliminating the operating system's TCP/IP stack, and allowing a transition to an overall simpler operating environment to proceed incrementally rather than all at once. You could imagine running a self-contained simple system like Squeak or Oberon under a hypervisor like Xen, communicating with a Linux guest to run legacy applications such as Chrome. Qubes is an OS project using an approach like this one to improve Linux security. (Unfortunately, Xen still relies on Linux drivers to handle the hardware.)

Virtual-machine emulation also provides crucial debugging capabilities that dramatically accelerate the first steps in the development of new operating system kernels, and potentially the debugging or analysis of existing ones. Back in the 1990s, the "spa" SPARC Performance Analyzer was a very slow virtual-machine emulator that was used to deeply examine the execution of processes on a simulated SPARC, in some cases using techniques similar to compile-time instrumentation.

(Virtual machines also have potential uses in information archival: a well-defined virtual-machine specification with emulators written in it capable of running other software could enable the recovery of lost software and the file formats implemented by that software. Such a fictional scenario, centered around the "UM-32 Universal Machine", was the premise for the ICFP 2006 Programming Contest.)

Historically, we have often introduced virtual machine instruction sets not modeled on any physical CPU for a variety of reasons, including improving debuggability, allowing compilers to run their code on new machines without retargeting (the original excuse for the JVM, ridiculous now that HotSpot is much larger than javac), efficiently supporting operations not supported by the underlying machine (in the case of the WAM, for example), and stuffing more program code into limited memory (the reason for the Excel bytecode interpreter, the BASIC Stamp bytecode, the Microsoft BASIC-80, and to a significant extent, Smalltalk-80). You could imagine in particular that a snapshot of a system could be more useful if it were portable between CPU architectures, allowing you to migrate running applications between your cellphone and your server, but I'm not aware of anyone having done this.

Probably introducing new virtual CPU architectures will not simplify a computing system. At this point, instead of emitting bytecode, you should probably emit machine code for a common CPU, like i386, x86-64, ARM, MIPS, PowerPC, or maybe MMIX or RISC-V, and run it under an emulator for that CPU if necessary. You can emit really dumb machine code, and it will still run faster and be simpler than emitting and interpreting bytecode. This also has the advantage that someone else has probably already written an efficient implementation of your "bytecode machine" for most

popular architectures out there, and you can use object-code instrumentation and analysis to add new features either to code built with your own system or existing code.

But maybe I'm wrong and there is some possible system simplification, even today, from designing an abstract virtual machine.

Automatic dependency tracking and deterministic rebuilding

`make` is the standard Unix utility for avoiding unnecessary recompilation by memoization (see above), but it requires you to explicitly and statically specify the dependencies of each build target, and it almost invariably omits dependencies like the compiler executable itself and the Makefile rule.

Meteor is a JavaScript framework in which you run your presumably deterministic "computations" in a sort of transaction (see below about transactions) that tracks what they read and write, intermediating their access to a remote database. Then, if there is a change in any of the things they read (probably because some other computation wrote to it), they can re-execute your computation. This automatically detects the things that could have changed the result of your computation, using exactly the same mechanism as Composable Memory Transactions. In this way, you can construct a reactive dataflow computation with dynamic topology without ever explicitly specifying dependencies, and as long as your computation accesses no data that Meteor cannot audit access to.

Debian, Nix, Guix, and the Tor Project are using a similar approach to get reproducible compilation products from possibly unreliable compilation machines: by knowing all of the inputs to the compilation process, including the compiler executable, they can detect and recover from a compilation process on one machine producing incorrect executables, either due to bugs or due to attackers attempting to inject malicious code. Generally these reproducible-build systems identify the compilation inputs with a secure hash and store them in a content-addressed blob store.

This reproducibility also makes it possible to memoize the compilation process (see above about memoization), avoiding actual runs of the compiler that are not motivated either to verify reproducibility or because the inputs have changed. Vesta and ccache are two systems that work this way.

(Acar's Self-Adjusting Computation also, in some sense, works the same way, at a much finer granularity.)

ccache even automatically tracks compiler inputs the way Meteor does, as does Bill McCloskey's memoize (now on GitHub), which uses strace and is therefore slow. redo, implemented by Avery Pennarun, tracks inputs comprehensively and dynamically (like Meteor) but not automatically.

Apache Spark is a system for scalable cluster computation which deterministically computes "RDDs" either from stored input files or other RDDs, and which records the "lineage" of each RDD, which is to say, all the inputs that went into creating it; this enables RDDs, or parts of RDDs, to be recomputed in the case of machine failure.

RDBMSes do automatic dependency tracking to automatically update indexes and, if supported, materialized views. However, I

think RDBMSes typically use a different technique, which I'm calling algebraic incremental updates (see below), rather than rebuilding parts of an index or view from scratch when they have been invalidated. If simple rebuilding could be made adequately efficient, which is kind of what Acar's self-adjusting computation claims, maybe RDBMSes wouldn't need to support algebraic incremental updates.

This kind of automatic dependency tracking could dramatically improve the efficiency of snapshots (see above): anything that could be automatically rebuilt does not need to be included in the snapshot, and indeed that is how we manage our source code repositories.

If previous build outputs are stored in a distributed cache, they typically are stored in their entirety, perhaps compressed. However, they are often very similar to one another. Using a rolling-hash-based system like bup or Jumprope could conceivably allow the storage of orders of magnitude more cached build outputs, which could be a useful alternative to finer-grained dependency tracking like SAC.

A common performance problem in C and especially C++ compilation is header-file parsing; due to recursive `#include` lines, it is common for a single source file to include tens or hundreds of megabytes of header files, which must all be tokenized. A common approach to this problem in the past has been to add "precompiled header" support to the compiler, which serializes the state of the compiler to the filesystem at some predetermined point, and then attempts to use the serialized state only when it is safe to do so. This tends to be bug-prone. A possible alternative would be to snapshot the compiler's runtime state several times per second during the compilation, annotating the snapshot with the set of inputs so far consumed; then, when a rebuild is called for, the old inputs can be compared to the new inputs, and the previous compiler run can be resumed from the last snapshot which had not yet read any changed data. This will not speed up the compilation of one C source file using data from the compilation of another (except perhaps in the sense that the snapshots could share space due to after-the-fact substring deduplication in storage), since the compiler receives the differing command lines immediately on startup, but it will speed up the recompilation of modified files, potentially by a much larger factor than mere precompiled headers.

A totally different alternative way to avoid precompiled headers would be to parallelize the tokenization process using a parallel prefix-sum algorithm; a third approach would be to make the token stream itself memoizable, so that you only need to retokenize any given header file when the header file itself or something it `#includes` changes.

Prefix sum, cumsum, or scan

The prefix-sum problem is to compute, say, 1 3 6 10 15 21 from 1 2 3 4 5 6, or 32 5 1 32 1 from 32 -27 -4 31 -31, or using max instead of addition, 1 2 2 4 4 4 4 8 from 1 2 1 4 1 2 1 8; each output item depends on all the input items at its position or earlier, but the associativity of the operation used to combine them allows the use of a variety of efficient parallel algorithms to compute it in logarithmic time. This means that if you can cast your problem into the format of a prefix-sum with some associative operation, you can more than likely get reasonable parallelism out of it.

Now, note that function composition is associative! It might seem that this would allow you to simulate any sequential process in logarithmic time on a parallel machine, but in fact that is only the case if you have a bounded-space representation of the transition function over an arbitrarily long period. For example, if you're concatenating millions of 3x3 matrices or simulating a 20-state Levenshtein automaton (or a lexer) over a gigabyte of text, this approach does indeed work! But for, say, parallelizing the execution of a virtual machine, it probably won't.

Prefix sums are included in APL and some other array languages as a fundamental operation, called "scan", "\", or "cumsum", but in some of these languages, it is not applicable to arbitrary associative binary operations.

One of the standard parallel prefix-sum algorithms recursively builds a segment tree over the input array; if memoized, this provides an efficient logarithmic-time lazy incremental algorithm, even when the operation in question does not admit inverses (the common examples being max, min, or associative bitwise operations.) The same segment tree can then be used for the range-minimum-query problem and its variants, even though RMQ is more general than the problem of returning an item from min-scan problem (since the range being queried need not start at the beginning of the vector).

(If the operation does admit inverses, you can instead incrementalize it using algebraic incremental updates; see below.)

This is a generally applicable strategy for incrementalizing algorithms expressible as prefix sums. For example, if you use a finite state machine to syntax-highlight an editor buffer or conservatively approximate a CFG parse, you can build a segment tree of state mappings over substrings of the input, which allows you to update the entire output in logarithmic time after localized changes to the input, such as edits to an editor buffer.

One of the standard applications for the plain-vanilla addition version of prefix sum is to rapidly compute box filters, also known as simple moving averages, sometimes as an alternative to mipmapping of images and in other DSP applications. They can also be used to calculate arbitrary linear IIR filters.

PID control

PID control is a fairly general linear control algorithm to push a system in the direction of the state you want. You exert a push that is a weighted sum of three components: the difference between the desired and actual state (the Proportional component), one that is the Integral of the difference in order to correct offset errors, and one that is the Derivative (or differential) of the current state in order to damp oscillations which would otherwise result from the other two components in the presence of hysteresis or phase delay in the system.

PID control is very widely used in the control of physical machinery in industry by microcontrollers or PLCs and even pneumatic controllers and analog circuits.

(Hmm. Maybe this is not actually that powerful?)

String similarity

useful for compression, virus scanning

Suffix arrays

A suffix array of a string is a permutation of the indices into that string, sorted so that the suffixes of the string starting at those indices are lexicographically sorted. Any particular chunk of a suffix array lists the places in the string that a given substring occurs, lexicograhically sorted by what's after it. (Well, maybe a range of substrings.) It's a lot like a KWIC index, except that it is fully comprehensive.

Suffix arrays are a simplification of suffix tries, such as Patricia trees (I've never been entirely clear on whether Patricia trees are a particular kind of suffix trie, or just another name for suffix tries). Once you have computed a suffix array for a corpus of text, there are a variety of queries you can execute efficiently on it: all occurrences of a given substring, the probability distribution of characters that *follow* a given substring, all matches of a particular regular expression, all approximate matches of a given string (using a Levenshtein automaton), and so on. As you can imagine, this is useful for data compression, and, indeed, the Burrows-Wheeler Transform used in bzip2 is suffix-array-based. Suffix arrays themselves are large compared to the text they describe (N log N bits for an N-character text, so commonly 5 or 6 bytes per byte of text) but there are suffix-array-based things I don't understand called "compressed indexes", such as the FM-index, which represent a text in a compressed form and support both decompression and efficient search operations on it efficiently.

Historically, suffix-array construction was very computationally expensive, which is one of he main reasons bzip2 is slow. But now there are new suffix-array construction algorithms (SA-IS, the skew algorithm, and a third one which I think is called SA-IR) that are linear in time and use a reasonably small amount of memory, like, less than twice the space of the final result.

This linear-time ("O(N)") bound is particularly intriguing since comparison-based sorts have a proven (?) complexity bound of O(N log N) time. It turns out that there is no contradiction: to have N distinct keys, each key needs to occupy at least $\log_2 N$ bits, so O(N log N) comparison sorting can actually be O(N) in the total size of the database.

Often, though, it isn't! So suffix-array construction could conceivably be an absolute performance win as well as a simplifying tool.

Most database sorting, indexing, and search problems are in some sense a subset of the substring-search problem.

Field search with suffix arrays: If a record has the string "CA2777697A1" in the field "patentnumber", most representations of it will also contain "CA2777697A1" as a substring, and probably not very many other records will. Indeed, with appropriate choices of encoding, you can ensure that, for example, the string "&patentnumber=CA2777697A1&" will occur exactly and only in the representation of records with that string as the value of that field. This extends in an obvious way to prefix (`patentnumber like 'CA27%'`) queries and range (`year between '1972' and '1977'`) queries, although multi-column indices can only be emulated in this way if the column sequence is a substring of a column sequence strictly observed in the representation of every record. On the other hand, this gives you N(N-1)/2 multi-column indices on N columns out of the box for

free, if your table is in 1NF and you observe this ordering constraint, as well as suffix searches: `patentnumber like '%7A1'`.

It's also possible to do full-text fielded search in this way by using regular expression search (`/&body=[^&]*Argentina/`) but I don't expect that to be very efficient. It's probably better to do full-text fielded search in some other way, such as by doing a full-text unfielded search and then examining each result to see if it's in the right field, or if that's inefficient, by concatenating the desired field into a separate corpus.

(This approach, without the suffix arrays, was the basis of the cult PC database program askSam in the 1980s, 1990s, and into the 2000s: its "records" were free-form text; a "field" was just a field name followed by square brackets with text within, like `patentnumber[CA2777697A1 ]`; and it used a full-text index to support fielded search. askSam is still around, though somewhat less popular than before.)

askSam was more like what we call a "document database" nowadays, like MongoDB; its records were not in 1NF.

Record sorting with suffix arrays: clearly if you want to sort the "patentnumber"-containing records in the above table by patentnumber, you can simply look at the chunk of the suffix array for strings beginning with "&patentnumber=", and find the beginning of the record each such substring falls within. If you insist on ordering strings like "X", "X%", and "Xa" in that order, you can change the delimiter from "&" to ASCII NUL. I suspect that using magical "out-of-band" "bytes" as delimiters is the easiest way to get this kind of thing to work correctly, although it's no doubt possible to devise an escaping scheme that preserves collating order and allows this sort of reliable substring testing to find field boundaries. (All three of the practical linear-time suffix-array algorithms cope well with odd-sized alphabets, and in fact use them internally. Probably the best way to represent the out-of-band values in memory is by storing a list of their indices.)

In the Canon Cat, Jef Raskin suggested that perhaps instead of seeing our data as a plurality of different files addressed by name, we should see it as one document, perhaps a very long one, divided into sections, which can be addressed by content, simply by doing substring searches. (I'm not sure if his later work on The Humane Environment continued to attempt this.) Suffix arrays make it possible to scale instant substring search up to at least hundreds of gigabytes of text, if not more.

Aside from full-text indexing, sorting, and compression using suffix-array-based schemes like the BWT, suffix arrays have also been used for LZ77 and LZSS data compression, and compact infinite-order Markov-chain models of text.

What else could you simplify in a computing system by using suffix arrays?

Levenshtein automata

Levenshtein automata are finite-state machines that recognize strings with up to a given Levenshtein distance from a desired search string; in particular, you can simulate a Levenshtein automaton in parallel on a sorted or trie index (including suffix-array indices) to find all the fuzzy matches to a desired search string in a body of text. The automata grow in complexity as the maximum Levenshtein

distance grows, but for distances of 1, 2, and even 3, the nondeterministic Levenshtein automaton has relatively reasonable size.

This is useful not only for words that someone might have misspelled, but also ssdeep forensic hashes of malicious code, nucleotide or peptide sequences,

XXX

Formal methods

"Formal methods" here means theorem-proving, specifically machine-checked proofs of the correctness of programs. Although this line of research goes back to the 1960s, it has started producing major results in the last few years, as people have finally started getting it to scale to real-world programs, including the seL4 secure kernel, the CompCert C compiler, a security kernel for an experimental browser called Quark, and forthcoming work that reports having verified the crash recovery of a filesystem. Essentially all of this has been done in a system called Coq so far, although other systems like Isabelle/HOL are intended as possible alternatives.

Of course, proving your source code correct doesn't help you much if the compiler introduces bugs in the process of compiling, which is why CompCert has been so central to this work. But much of the machinery of languages like C, and especially C++, is intended to do the same kind of thing that things like Coq are good at — statically find bugs, and automatically select possibly-correct behavior (such as applying a floating-point multiplication operation to two floating-point numbers) instead of clearly-incorrect behavior (such as applying an integer multiplication operation to them).

A recent paper from Kennedy, Benton, Jensen, and Dagand, "Coq: The World's Best Macro Assembler?", investigates the consequence — obvious in retrospect — that maybe we should just use write Coq code to generate the assembly or machine code, along with a proof of correctness.

This approach also offers the possible promise of being able to verify properties that aren't verifiable at C level, like a worst-case bound on the number of bytes needed for the stack, or execution timing for a timing loop; and it avoids the whole issue of proving the correctness of an inherently complex program like a C compiler.

Myreen and Curello have also formally verified a bignum implementation in AMD64 machine code using the HOL4 theorem prover.

Coq proof tactics are not Turing-complete — they are guaranteed to terminate — but they are capable of proving substantially more interesting safety properties of programs than the type systems of languages like C, Java, and even OCaml. (I suspect that C++'s template system is Turing-complete, just very clumsy, in the same league as languages like BF.)

This suggests a possible way to rehabilitate Forth, which is little more than an assembly language with expressions, compile-time metaprogramming, and reflection. The problem with Forth, in my limited experience and in hearsay, is that it's too bug-prone, and so you development starts to get slow pretty soon due to the bugs. But it invariably takes less total code to do anything in Forth than in any other language, if you count the code implementing the primitives.

Perhaps a Forth could constitute a useful penultimate level of intermediate representation, or even a better target language than assembly, for such a program of formal verification at the machine-code level.

Could you extend this macro-assembler approach into a full convenient programming language embeded in Coq or a similar proof assistant? What else could you simplify with a good proof assistant?

Hash tables, expandable arrays, and sorting

This, more than anything else, is the basis of languages like Python, Perl, Lua, Tcl, JavaScript, and arguably the Unix shell: two generic container types (or, in Lua's case, just one) allow you to represent things conveniently, and make the language easy to learn. Typically these use the Lisp object-graph memory model, in which all values and storage location are a single machine word in size, usually populated by a pointer.

Original Lisp in itself can be seen as the first such language: even McCarthy's 1959 paper had association lists in it, implemented FORTRANishly using parallel lists of keys and values, rather than in the now-standard alist representation of a list of pairs. But Lisp lists are optimized for structure-sharing ("persistence") and recursion at the expense of rapid indexing, mutability, and memory-efficiency, and Lisp's traditional representation of finite maps makes similar tradeoffs. This disfavored imperative code.

So Perl's 1987 minimal combination of arrays, associative arrays, and a convenient imperative scripting language turned out to be an extremely useful combination, particularly combined with regexps for input processing and string interpolation for output. Perl didn't have pointers before Perl 5, more or less obligating you to use parallel arrays or parallel associative arrays, but it was still eminently practical for a huge collection of tasks.

(REXX has "compound variables" which were associative arrays, which, like Perl4 associative arrays, couldn't be passed as arguments. REXX dates from 1979. But it doesn't have lists, and unlike Lua, there's no way to find out how many items are in a numerically-indexed compound variable.)

This kind of decomposition works best if the underlying implementations of the data structures are acceptably efficient across a broad range of uses; Lisp lists' O(N) append-item

Add sorting to arrays and associative arrays, as Perl did, and a whole host of traditional algorithms become simple. Parnas's famous 1972 paper is about how a group of programmers could coordinate through formal specifications to implement a KWIC index, essentially the following Perl program, which he says "could be produced by a good programmer within a week or two". It took me about 20 minutes, although it wasn't my first time. It needs about ten or twenty times as much RAM as the size of its input.

```
while (<>) {
    @w = split;
    push @lines, "@w[$_+1..$#w] | @w[0..$_] $.\n" for (0..$#w-1);
}
print sort @lines;
```

(I think this program works in Perl 4, but I don't have a copy handy to test.)

I took another hour or so to write a much more efficient, if nearly as obfuscated, version in Python, which can produce a KWIC index of the Bible in a few minutes on my netbook:

```
import sys

pos, aw, a = {}, [], []         # Each word is represented by index in aw
for r in sys.stdin:             # Build list of lists of word ids in a
    b = r.split()
    for w in b:
        if w not in pos:        # Assign an id to the new word
            pos[w] = len(aw)
            aw.append(w)
    a.append([pos[w] for w in b]) # Encode the line with word ids

g = sorted(range(len(aw)), key=aw.__getitem__) # Compute word sort permutation
v = [None] * len(aw)                           # Invert the permutation...
for i, j in enumerate(g): v[j] = i             # Now v[g[i]] == i.
a = [[v[w] for w in b] for b in a]             # Rewrite lines with permuted ids
cs = [(ln, wn) for ln, b in enumerate(a) for wn, _ in enumerate(b)]
cs.sort(lambda (lnx, wnx), (lny, wny): cmp(a[lnx][wnx:] + a[lnx][:wnx],
                                           a[lny][wny:] + a[lny][:wny]))
p = lambda ws: ' '.join(aw[g[w]] for w in ws) # map back to original words
sys.stdout.writelines("%5d: %s | %s\n" % (ln, p(a[ln][wn:]), p(a[ln][:wn]))
                      for ln, wn in cs)
```

(This task, of course, is trivial and perhaps even unnecessary if you have a suffix array already computed (see above). But it's an excellent example of how powerful primitives reduce the complexity of previously complex tasks.)

Smalltalk, perhaps the original exponent of "powerful primitives for a simplified computing system", has OrderedCollection and Mapping (XXX?) types, but culturally it doesn't emphasize them; instead, it has a wide variety of specialized collection types available, and it emphasizes defining new application-specific means of aggregating and finding things, perhaps wrapping an OrderedCollection or something.

Python has in some sense decayed a bit from this ideal: added to its basic tuple, list, and dict generic containers, we now have sets, generators, an entire `collections` module with containers like defaultdict and deque, some of which are dangerously bug-prone. While it's very useful to be able to define application-specific collection types and use them via ad-hoc polymorphism, and often these specialized collections are significantly faster in particular uses, I worry this makes Python harder to learn than it used to be, and perhaps also reduces the power available by combining components — although at least these new collection types all support common sequence and iterable protocols.

And of course bencode and JSON, and the systems based on them like BitTorrent and MongoDB, are entirely based on these two container types, plus primitive numbers, strings, and null. Bencode goes an additional step and ensures that the mapping between data

structures and serializations is one-to-one, like ASN.1 DER, allowing cryptographic signatures and the use of content-hash-addressable storage.

Could we simplify our systems and improve their composability by designing them to handle more data in JSON-restricted forms?

Finite binary relations

One alternative that has occurred to me to the hash/array/sort combination described in the previous item — or Lua/JS/REXX's version where you use a hash of int keys instead of an array, or PHP's version where the key-value pairs are ordered — is finite binary *relations*, as opposed to the finite binary *maps* provided by hashes. Relations allow potentially more than one value for the same key; you can implement them as, for example, hashes of sets.

I've written about a database query language based on binary relations at http://canonical.org/~kragen/binary-relations, but I'm still not sure if it's a good idea. But this is not about writing databases with query optimizers and queries and whatnot; this is about using binary relations as the standard container type in a very-high-level programming language.

At the level of objects in a programming language, relations have some possible advantages over maps. Relations are closed under inverse, composition, and transitive closure, as well as the set-theoretic operations union, intersection, difference, and symmetric difference, which operate on the set of their key-value pairs; furthermore, the inverse of the inverse of a relation is the original relation, relational composition is associative, the set-theoretic operations are commutative and associative, and there are various distributive theorems you can prove. You can try to define analogous operations on maps, but I can't think of definitions that support these properties.

If your relations are implemented with some kind of search tree over the keys, rather than a hash table, then the keys can be efficiently traversed in sorted order at any time, even when keys are being added and deleted dynamically. This obviates not only explicit sorting by keys but also explicit heaps; a search tree has the same asymptotic complexity for the usual heap operations.

Considering the KWIC program again, let's see what it would look like if Python had implicitly-ordered relations instead of dicts and "lists".

```
import sys
```

Now we don't need two separate collections for the words, but we still do need to be able to rapidly map from a word to its id as we are building up the word-id-sequence representation of the input text. So let's store the words as the keys of a relation, their ids as the values, and use a counter. (Actually, maybe I should have done this before.)

```
n, aw, a = 0, {}, {}

for r in sys.stdin:
```

Do we want iteration in this Relational Python to be, in some sense, iteration over a relation? That is, do we want sys.stdin to

somehow present its lines as a relation, or do we keep the same iteration protocol, which iterates over a sequence of things? I think we should keep the same protocol, because the nature of iteration isn't changing, just the containers. Iteration is still the execution of a block of code a sequence of times.

```
b = r.split()
```

Now what does split() return? Presumably a relation, but where do the words go — keys, values, or both? And, if not both, what's the other side? I think the answer is that, since the field sequence in `split` is usually important, and we usually want to be able to do field lookups by number, it should be a relation from field numbers to words. Here, we do care about the sequence of words eventually, since we need to do cyclic shifts of them eventually.

But now we need to iterate over the words that don't have ids, and we don't really care what order that happens in. This sounds like set subtraction, which we've already said is a fundamental operation on relations, but the set we're subtracting isn't exactly `aw`, the relation from words to their ids; it's just `aw`'s keys. But that's potentially a very large set, and we wouldn't want to do anything proportional to its size after every word.

So we're declaring that getting the set of keys of a relation is a constant-time operation, which seems plausible, since it doesn't require building a new key tree; it just requires interpreting the traversal results slightly differently. We're also declaring that a set is represented as a relation that stores the single value `True` for each key.

```
for w in b.valueset - aw.keyset:
```

(Since the keyset and valueset operations are so fundamental, it might be a good idea to provide syntactic sugar for them, like @b or aw@ or something.)

We probably don't want to have an `x[y] = z` operation for relations, since `x[y]` is an unordered set of things, and the fundamental operation is not replacing that set with a single item, but rather adding an item to that set. (Although, in this case, we've just verified that that set is empty.) So for now I'm going to use a method, although some kind of syntactic sugar might be better.

```
    aw.add(w, n)
    n += 1
```

Now we are faced with the need to translate the line (a mapping from field indexes to strings) through the id table `aw` (a mapping from strings to ids) before adding it to `a` (a mapping from line numbers to sequences of word ids). This is just relational composition, which I will represent with the infix operator @:

```
a.append(aw @ b)
```

The order of the arguments to `@` is the traditional one for relational or functional composition: the range values of `b` are the domain values (keys) of `aw`, the range values of `aw` are the range values of the result, and the domain values (keys) of `b` are the keys of the result.

You could read this as "Values from `aw` at the keys from `b`. It's kind of like `aw[b]` except that `b` has a whole sequence of indices for `aw` (in its values), not just one.

One potential problem here is that relational composition is well-defined when the value on either side of the composition is missing: `{4: 5, 6: 7} @ {1: 2, 3: 4}` is just `{3: 5}`, silently ignoring the absence of any key 2 in the left relation and any value 6 in the rightmost relation, while the Python expression `[pos[w] for w in b]` will instead raise an exception if `w` is not in fact found in `pos`, which is probably a programming bug and the kind of thing you would want it to catch. We could define `@` to do that, but I'm not sure we should.

This .append operation generates a new index by adding 1 to the previous greatest index in the relation, which is a thing that can be fetched in either constant time or logarithmic time.

Next, we need to permute the word ids according to the words, in order to derive new word ids that will induce the correct lexicographical ordering of cyclically-shifted lines. But, since relations are implicitly traversed in the order of their keys, this sorting is achieved by simple relational inversion, after which we get the new word ids by counting — using the `list()` function borrowed from Python, which in Relational Python turns an iterable into a relation with counting numbers as its keys:

```
g = list(aw.inverse())
```

Then, we need to calculate the inverse mapping, from our original word ids to our new word ids, and use it to rewrite our representation of the text.

```
v = g.inverse()
a = list(v @ b for b in a)
```

So far the program is slightly simpler than before. Now, though, it gets slightly more complicated, because the original code used a comparator function instead of a key generation function to specify the sort order of the cyclic shifts, specifically in order to avoid the materialization of the entire universe of cyclic shifts in memory at once. Using a key function instead of the comparator makes it run about three times faster on small datasets, but also use about 70% more memory on my example dataset. (My Python code uses about 12 bytes of RAM per input byte with the comparator optimization, and about 30 bytes of RAM per input byte using the key function.)

However, it's fairly simple to create a key class that lazily generates the key it wants to be sorted by, if we declare that relations use this **key** method to compare and sort their keys, but don't store their return values or leave an arbitrarily large number of them out there hanging in space unfinished.

```
class K(namedtuple('K', ['ln', 'wn'])): # line number, word number
    def __key__(self):
        return list(a[self.ln][self.wn:]) ++ a[self.ln][:self.wn]

cs = {K(ln, wn) for ln, b in a.items for wn in b.keys}
```

Here I'm declaring that ++ is the list concatenation operator,

which returns a new relation that is a superset of the first relation, with the values from the second relation assigned to new indices as per `.append`. The list() call is needed because slicing a relation returns you the key-value pairs from that slice unchanged.

The {} comprehension there is a set-comprehension, implicitly mapping each key to `True`; if it had a `:` after `K(ln, wn)`, it would be a relation comprehension.

There's a problem here, though, which is that `a[self.ln]` is actually an entire set of lines that happens to be, presumably, just one line. I need to figure out how to handle that. XXX

The remainder of the code is simplified slightly from the Python version, although it omits the `|`. It's assuming a `key()` top-level function that invokes the `__key__` magic method.

```
sys.stdout.writelines("%5d: %s\n" % (k.ln, ' '.join(aw @ g @ key(k)))
                      for k in cs.keys)
```

I'm not sure exactly what kind of thing the tuple there in the string-formatting operation should be; is it a relation? Should it not exist? Should I be writing it with {} or []? XXX

Transactions and transactional stores for concurrency

Transactions are an architectural style for building systems, whether distributed or otherwise, that contains the effects of each computation inside a "transaction", whose reads and writes are monitored, and whose writes are not visible to other transactions until it "commits". If it fails (according to arbitrary logic inside the transaction), it doesn't commit, and therefore has no effect. The system prevents transactions that had read data that was been written before they committed, either by preventing it from being written (perhaps by delaying or aborting the transactions that try to write it) or by automatically aborting and retrying the transaction that depended on the outdated data. In this way, the transaction system achieves an illusion of pure serial execution, while in fact permitting great concurrency and even parallelism.

Ideally, in fact, the transaction should never be able to read any data that were written after it started, in order to ensure that no implicit consistency constraints among different data that it might read are violated. For example, a transaction calculating the average packet size on a network interface might read the number of packets transmitted and the number of bytes transmitted, then divide them. If this result is to be correct, the number of bytes it is dividing must correspond to the number of packets it is dividing by.

(This is obviously easier to achieve if you have persistent data structures, in the functional-programming sense.)

This kind of transaction is known as ACID: it's "atomic", in that either it has all of its effects or none of them; "consistent", in that only transactions whose internal failure logic does not abort them will be committed; "isolated", in the sense that concurrently executing transactions can have no effect other than perhaps aborting them; and, kind of unrelated to all of the above, "durable", in the sense that none of these properties are lost if your computer loses power in the middle of the operation, and in particular no committed transaction will be lost.

Practical transactional systems often relax some of these properties to some extent. For example, it's very common for database systems that can provide fully-ACID transactions to support lower "isolation levels", perhaps permitting a transaction to see an inconsistent state of the database as it reads a sequence of different things.

The "Composable Memory Transactions" paper from 2005 or so proposes using transactional *memory*, implemented in software, as a general concurrency control mechanism for threads. The CMT transactions support nesting, and an outer transaction can react to the failure of an inner transaction by trying an alternate code path; this permits a more modular equivalent of the Unix select() call or the Win32 WaitForMultipleEvents, simply by trying such an alternation of nested transactions that each fail when the event they await has not yet happened. If any one of them succeeds, then the overall transaction continues; otherwise, it fails, and is later automatically retried when any of the data it read before failing have been modified. This permits the automatic interconversion of blocking and non-blocking APIs, something not possible by any traditional means other than first-class continuations.

There's an interesting correspondence between this kind of transaction failure and failure in nondeterminism implemented by backtracking. When you hit a failure in a backtracking system, you undo all the effects that led to that failure, back to the latest backtracking point, and then try the next alternative from that point. This is exactly the same thing that happens in CMT when an inner transaction fails: all of its effects are undone, and the outer transaction can either fail in response, or try a different alternative. In both systems, only when the outermost transaction succeeds do you get to see the set of assignments that it made. One interesting difference is that, in backtracking systems, the outer transaction can fail the inner transaction after it has already "committed", possibly causing it to succeed with a different set of effects.

This kind of transactional-memory discipline probably isn't reasonably efficient and effective without either hardware transactional memory support, or a sufficiently smart compiler or prover, to ensure that no effects escape a transaction before it is committed.

Obviously you can implement this kind of transactional-memory interface with a networked server instead of stuff in the same process as the transactions; although you can't prove that the stuff on the other end of the socket is respecting the requirement to have no effect if it gets aborted, having it in a separate process makes other kinds of isolation more plausible to implement, and the possible high concurrency may be more valuable when you can spread it across a whole network.

Also, there's a clear connection to automatic dependency tracking and deterministic rebuilding, as well as to reactive programming: the read and write tracking the transactional memory does is the same read and write tracking a reactive programming system like Meteor does, and indeed the re-execute-when-an-input-changes logic used by the CTM paper for transactions that haven't committed is the same thing Meteor does for "computations" that are still live. You could even imagine a nondeterministic search algorithm like a truth maintenance system implemented in these terms: if transaction A makes a write that results in a failure in a later transaction B, then transaction A is rolled back retroactively, along with all the things resulting from it, and re-executed with that write rigged to blow

up — at which point A can either fail, or handle the error by trying a different write.

All of this could be implemented at a number of different levels. One interesting level, for interacting with current systems, would be running a shell script inside an isolated environment monitored by a transaction system; it can read files and directories, write files, and spawn subtransactions, and if it completes successfully, then its writes are committed and become visible to other scripts. By itself, this would suffice to simplify certain kinds of system management tasks: you could abort long-running scripts halfway through without fear that they would leave things in an inconsistent state, and the transactional discipline would ensure that concurrent execution didn't cause subtle bugs. (And no more tempfile race holes!) But the prospect that you could, later on, automatically re-execute the script if any of its inputs changed, perhaps with a frequency limit or extra wait time, and perhaps automatically deleting its previous outputs in the interim — that seems like it could simplify a lot of things.

All of this shell-script stuff seems like it would work best with a blob store, in order to make it practical to distribute this work across machines, and also so that the operation of "committing" and making visible the output files can be done quickly.

We normally think of ACID transactions as being incompatible with eventually-consistent and partition-tolerant databases, because of Brewer's CAP conjecture (later somebody's XXX theorem), which is that you can't be consistent (in the sense of serialization), available (in the sense of always handling all requests), and partition-tolerant (maintaining these in the face of network partitions). But the above suggests a way to achieve at least ACI transactions in an eventually-consistent partition-tolerant system: if you remember the entire transition history and dependency graph, then you can commit a transaction and, when you find that it was in conflict, later roll it back, along with everything that transitively depended on it, then re-execute them in a proper order. This sounds pretty wild, but it is more or less the way that the US credit card and banking system work: any transaction can be disputed and rolled back for a month or three, which may result in rolling back other dependent transactions. It kind of means that your system can never make any promises to the outside world, though, which is becoming a friction point for interactions between US financial institutions and things like Bitcoin and physical goods delivery, which do not support rollback.

Re-executing the whole dependent sub-DAG of transactions at the timestamp when you discovered the conflict should be safe, at least if only one node is doing it. Re-executing them at their original timestamps with the new data could result not only in them writing different contents to the same locations, which is harmless since that only affects other transactions in the dependent sub-DAG that you were going to re-execute anyway, but also writing things to locations they didn't write to before. This could result in the need to re-execute other transactions that weren't originally part of the dependent sub-DAG. I don't know what the best choice is here. (However, I'm deeply struck by the correspondence with Umut Acar's self-adjusting computation algorithms, in which function calls are required to pass values around only by reading and writing memory, to read values from memory only at the beginning of a

function, and you re-execute function calls if their inputs changed — and each function invocation is timestamped with a virtual clock in order, if I understand correctly, to allow safely memoizing functions that access memory.)

This whole re-executing thing requires that the code that was executed in the transaction be preserved so that it can be re-executed later if the transaction needs to be rolled back and redone due to a conflict. In a distributed database context, you probably want to reference that code with a hash stored in a content-hash-addressable storage system rather than by saving a copy of your entire source base in every replicated transaction record.

(Also note that if you're saving and replicating the entire past history of transaction inputs, timestamps, and code, plus an initial database state, you have in effect turned your database into a CRDT; you replicate this history, perhaps using gossip (see below), and can then deterministically replay all the transactions to deterministically rebuild the current state of the database, and any node doing this will get the same results. This may be simpler to do if you in fact execute the transactions serially, eliminating the need to track inputs and outputs at fine granularity, though the more usual transaction approaches may give you better performance.)

This may be a foundation that simplifies operational-transform-based collaborative editing systems like Etherpad and Gobby.

Automatic differentiation

> Automatic differentiation … computes … any derivative or gradient, of any function you can program, or of any program that computes a function, with machine accuracy and ideal asymptotic efficiency.

It's an abstract-execution technique that executes programs on data that is annotated with its partial derivatives with regard to input data values, which is to say, the numerical Jacobian matrix of the program (and the Hessian, ad infinitum), evaluated at a point. Alexey Radul wrote a comprehensive introduction. It's easy to implement; Conal Elliott wrote a 2009 paper on this, where the first example shows how to do it for first derivatives of scalar functions only in 30 lines of Haskell.

The idea is that when you call a function and it produces some data structure full of values, you get not only the values, but also a lazy list of which inputs each of those values (locally) depends on, and what the coefficient of variation is between them at that point, and similarly for what that variation depends on, with minimal overhead.

This could be particularly useful with AI optimization algorithms that are trying to find input values for a piece of code that will produce a given output value, such as zero, or a given image. Gradient descent needs to know the gradient, after all. The alternative is hill-climbing by groping around in various directions looking for a local improvement, like a caterpillar at the end of a twig, which can be very slow in a high-dimensional space.

A lot of these are only feasible with what's called "reverse-mode" automatic differentiation.

There's clearly also a relationship here with incremental computation, which exactly computes the new value of a function after an incremental change to the input by tracking which parts of

the computation depend on that input; truth-maintenance-system constraint solvers, which trace back constraint violations they find to the minimal combination of nondeterministic guesses they made that will cause that constraint violation, in order to avoid wasting time with that in the future; probabilistic programming, which in some sense traces back observations of outputs to probability distributions over nondeterministic variables that would be needed to produce those outputs; neural networks (see below), whose "training" works by adjusting edge weights to reduce the difference between the output and the desired output, and indeed AD applied to neural-network evaluator does produce the backpropagation outputs; and interval arithmetic, which can produce conservative approximations that are much less conservative if it knows which uncertain input variables or intermediate computations an uncertainty bound depends on (consider $b \cdot b \cdot b \cdot b$ vs. b^4).

In particular, both incremental computation and automatic differentiation have to deal with the problem of preserving overwritten state when applied to imperative programs that use mutation internally; the body of knowledge that has developed around automatic differentiation is likely to be useful for implementing incremental computation.

Automatic differentiation only determines the derivatives in the local neighborhood of the computation that was actually carried out, and so in particular conditionals are not reflected; automatic differentiation will compute that `x ? -y-z : w/2` has a nonzero derivative relative to either `y` and `z` or to `w`, but not relative to all three, and not relative to `x`. For this reason, I expect it to be complementary to logical/symbolic search and deduction systems like miniKanren or SAT solvers or other constraint solvers, thus saving those systems from having to consider individual bits of a numeric result. (An SMT solver, if I understand correctly, is just a SAT solver extended with such an external "theory checker".)

(Similarly, the original truth maintenance system in Stallman and Sussman 1977 was a nonlinear circuit simulation package, kind of like an early SPICE; it was used to search among combinations of cases for piecewise-linear circuit component models, each case considered then being solved by a linear constraint solver.)

In the mathematically-oriented programming language Julia, there are several automatic differentiation libraries, and one named ReverseDiffSource is used by the Markov Chain Monte Carlo engine Lora to compute gradients of statistical models for, as far as I can tell, probabilistic programming systems (see above).

Interestingly, it turns out that the Tapenade automatic differentiation system uses program-state snapshotting (see above) and deterministic rebuilding (see above) to allow its reverse-mode automatic differentiation to work on imperative programs that overwrite part of their state during execution.

"The largest application to date [to have been automatically differentiated] is a 1.6 million line FEM code written in Fortran 77."

How widely can automatic differentiation be applied? Does it indeed synergize with efficient constraint solvers like miniKanren or CVC4, as I speculate above?

(I don't really understand automatic differentiation, so I hope none of the above contains any grievous errors.)

Bitcoin

Some years ago, after a conversation with Aaron Swartz, Zooko wrote a paper with an unmemorable title, now generally known as the "Zooko's Triangle paper". He proposed that although we would like globally-unique names in computer systems to be human-readable, securely dereference to the correct referent, and not be vulnerable to a centralized authority, in practice we can only achieve any two of these three properties.

Specifically, secure hashes, as used in content-hash-addressable blob stores, are not human-readable (the best we've been able to do is reduce them to about 80 bits and use representations like "able merit floor gower jerry jews sd", "ADD RAKE BABY LUCK MADE GOLD FEET SEEN", or "企珓鑣毒昕传"), and other kinds of global names are not "self-certifying", in Mazières's sense, so it would seems that we have to trust some external authority to certify them. Of course, fully homomorphic encryption (see above) could in theory enable this problem to be solved, but it's still not feasible, and when Zooko wrote his paper, it wasn't even on the radar.

Bitcoin, a variation on Nick Szabo's property title clubs, came out some years later. As Aaron pointed out, it demonstrated an alternative: a replicated global data store subject to nearly arbitrary consistency constraints that was not vulnerable to any particular participant, without having to figure out how to do FHE.

Bitcoin and pseudonymous cryptocurrency systems like it may be the most powerful primitive in this list, to the point that I suspect they may be a bad idea — the reason I haven't participated so far. Frictionless cross-border capital flows like those enabled by Bitcoin were expected to result from Chaum's centralized anonymous "digital cash" systems, which instead failed in the market. As Intel physicist Timothy C. May famously observed at the Hackers conference (perhaps Hackers 1989?), such flows seem likely to make taxation essentially voluntary, resulting in the collapse of governments. As pointed out later on the cypherpunks list, they could also enable pseudonymous betting to be used to crowdfund political assassinations, something that John Poindexter tried in 2003 in the Total Information Awareness program, resulting in a Congressional inquiry; this also seems likely to result in the collapse of governments.

I expect this to be a traumatic event, possibly resulting in the destruction of civilization.

However, Bitcoin also can be applied to solve a variety of difficult distributed-systems problems, and civilization might survive it and therefore be able to apply it to them. Naming is one; remitting money to your family overseas is another; email postage to stop spam is another; secure P2P rendezvous advertisement is another;

XXX

Deep neural networks

Conservative approximation

of what? parsing, say, or interval arithmetic.

Closures and Continuations

This is the Scheme 1975 vision of powerful primitives: closures give you one single way to do ad-hoc polymorphism, by giving you

in a sense first-class templated functions which you can press into service both as objects, as lazily evaluated values, and as basic blocks for control-flow constructs; and call-with-current-continuation gives you one single control-flow construct that subsumes threads, exceptions, Common-Lisp-style resumable errors, and backtracking, and allows you to invert control flow and turn blocking functions into nonblocking ones, and vice versa, which is kind of like the transactional memory concurrency constructs I mentioned above. Raph Levien's Io language provides a clean syntax that uses closure invocation even for statement sequencing.

Unfortunately, in a sense these constructs are *too* powerful.

If a continuation can be saved from anywhere, it's unsafe to irreversibly clean up resources on exit from the block where they are used; a continuation invocation could transfer control back inside, for example to implement thread context switching. At last I think `dynamic-wind` has been added to the Scheme standard, but it is substantially harder to use than `unwind-protect` or `try/finally`. Unrestricted multi-use first-class continuations can be implemented simply by allocating all your activation records on the heap, but implementing them *efficiently* more or less requires implementing segmented stacks, which makes every function call more expensive, although stack segmentation also makes threads a heck of a lot cheaper.

And even very clever compilers are unlikely to match the zero-cost exception handling that's standard in modern C++.

In a sense, using one single construct for all of these things requires both the human reader and either the compiler or the runtime to reconstruct a conservative approximation of some knowledge that was in the head of the original author (is this a thrown exception, a backtrack, or a thread context switch?), but who didn't write it down.

The same problem attends the use of closures and tail-calls for all control flow. The set of variables captured by a closure is purely implicit, so both the human reader and the compiler must reconstruct them; consequently, the lifetime of variables in a language with closures is purely implicit. But the compiler needs this lifetime information to produce efficient code, the author needs it to write code that works, and a human reader needs it to understand the code.

So I feel that, while continuations and closures are extremely powerful *semantic* primitives, and they have brilliantly elucidated many crucial aspects of compilation, their strength-to-weight ratio as *system-building* primitives is in doubt. They do result in shorter source code, but they often don't succeed in decoupling the code built on top of them from the aspects of the implementation they elide.

You could make many of the same accusations about dynamic typing, of course. I think the major difference is that while dynamic typing makes debugging easier (because the program mostly runs, and then when it crashes with a type error, you see an example of why), closures and first-class continuations make debugging harder.

You could make a similar accusation about implicit imperative control-flow sequencing: your program implicitly specifies a total ordering of computational steps to perform, and then the compiler works hard to recover the partial order you actually had in mind, in order to be able to execute your code efficiently. Nowadays, so does

your out-of-order-executing CPU.

Collaboration facilities

Succinct data structures

"Succinct data structures" are an extension of the compressed indexing work I mentioned under "suffix arrays". Unfortunately, I don't understand them. XXX

Interval arithmetic

Interval arithmetic is a particular form of conservative approximation (through abstract interpretation), with two principal uses: preventing numerical errors and avoiding computation.

Numerical errors are ubiquitous where we use floating-point math, because floating-point has finite precision, so its results are commonly approximate. The IEEE-754 floating-point standard offers control of the rounding mode — you can request that calculation results be rounded, for example, up or down. This allows you to, for example, calculate the sum of a list of numbers twice, once rounding up and once rounding down, and be sure that the true result was bounded by the two calculated results.

In interval arithmetic, instead of calculating with individual numbers, we calculate with (min, max) pairs of numbers known to bound the true quantity. Calculating (a, b) + (c, d) is quite simple: it's just (a + c (rounded down), b + d (rounded up)). Some other operations, like multiplication, are more complicated: (a, b) × (c, d) might be (a × c, b × d), but, for example, if a and b are positive and c and d are negative, it will be (b × c, a × d), with the appropriate rounding. Division is a bit more complicated still: (a, b) ÷ (c, d) might be a neighborhood of infinity, if c and d have opposite signs.

Interval-arithmetic libraries are available for bulletproofing your numerical computations by, as above, doing each operation two or more times, using different rounding modes. This can save you a lot of numerical analysis. These libraries are powerful and well-developed, but they are not really what interests me at the moment.

The more interesting case, to me, is avoiding computation.

SICP gives the example of approximating the zero of an arbitrary continuous, not necessarily differentiable, function, by recursively subdividing an interval in which it is seen to cross zero. You can extend this to find the zero locus of an arbitrary continuous multidimensional function by using interval arithmetic to rule out intervals where the function cannot be zero. But the technique has fairly broad applicability.

Consider backface removal. In rendering a solid bounded by polygons viewed from the outside, you can avoid half the occlusion computation if you eliminate the polygons that face away from the camera, since they will always be occluded by polygons on the near side of the object. So you can compute the normal for each polygon and consider whether it's facing toward or away from the camera.

This still involves computing something for each polygon on each frame, though. Interval arithmetic can save us. Suppose that the surface mesh is recursively divided into regions, and we store intervals for the x, y, and z components of the normals of the polygons in each region. Now, we can traverse this tree recursively and not even

descend into parts of the surface all of whose polygons face away from the camera, or for that matter all of whose polygons face toward the camera. It's only the mixed regions that require examination.

This saves us nearly all the computation for the backfaces, but we are still traversing the tree on each frame. We can do better still. Suppose we are rotating the object at a fixed, slow speed. Then, the rotation matrix for a given span of time can *also* be represented with interval coefficients, and we can multiply the normal intervals for mesh regions through this rotation matrix, discovering which parts of the surface don't face the camera at all during entire time intervals.

Somewhat similarly, when you're rendering, if you can compute the color at a point on the screen, and then bounds on the *gradient* of the color in pixel coordinates for a region around that point, you can determine whether that area contains any detail that needs to be rendered. If it's just a smooth gradient (the d[color]/dx and d[color]/dy are tightly bounded), you can render just a smooth gradient. And if you're computing an animation, you can also compute bounds over time.

(This is in some sense related to memoization and incremental computation: it allows you to change an input to a function slightly and retain its previous value, as long as you don't move outside an interval for which you've computed the result.)

I'm trying to remember the name of the guy who did his dissertation on precise interval-arithmetic rendering of implicit surfaces.

This is a kind of wild generalization of the widely-known bounding-box culling technique in computer graphics, and it allows us to achieve enormous economies of computation by computing that entire regions of a space of possibilities are of no interest. It's common to get three, four, or five orders of magnitude speedup with this approach, which you can then apply to focusing in more detail on the areas where more detail is called for.

It seems clear that this kind of possibility-bounding can work hand-in-hand with AI search algorithms (or simple nondeterministic search) and optimization algorithms, although I've never seen it applied in that way. If your SMT solver is looking for a contradiction in a region, you may be able to rule out the entire region at once. If you can place bounds on the "goodness" of points in an entire region of the parameter space, you may be able to quickly focus in on parts of the parameter space that are possibly of interest.

This only works usefully for operations that are in some useful sense mostly continuous, where reducing the size of the domain you're considering for an input will provoke a usefully reduced size of the range over which the operation's outputs are to be found. The bit-reversal function, for example, will tend to wreak havoc on interval-arithmetic approaches; you can only put useful (min, max) bounds on its output once its input is inside a very small range indeed. An interesting question is whether you can extend the kind of tractability that interval arithmetic provides to other kinds of functions, perhaps by using different representations than simply (min, max) pairs.

For example, earlier today I was optimizing a proportional-font word-wrap algorithm, which is one of the more expensive parts of a text-rendering system, simply because it has to examine every

character of the text it's wrapping to discover its glyph width. You could imagine a precomputed general-purpose interval tree covering the text, consisting of statements like "characters from 1032 to 1040 are all in the alphabetical interval 'a' to 'l'". From such an interval and a similar tree built over the font metrics, you could calculate a glyph-width interval: all the characters in that range then have a glyph width between 2 and 4, perhaps. And that would allow you to bound the sum of those glyph widths to be in the range (8×2, 8×4), which means that it's much less than the line width.

You could imagine a word-wrap algorithm expressed in a higher-level form, seeking the last candidate linebreak character before the prefix sum of the glyph widths of the characters exceeded the column width, being executed efficiently by progressively approximating the width-exceeding crossover and the last candidate linebreak, avoiding examining most of the characters.

But it seems obvious that such an algorithm, if it were useful (most likely it would only be faster than visiting every character once the lines became unreasonably long) would work *much better* if the interval tree over the text bounded the text's *glyph widths*, not its *codepoint values*, because the mapping from codepoint to glyph width is so irregular. As soon as your interval includes both 'l' and 'm', it spans the full range of possible glyph widths — even if the text itself is "the"!

Lempel-Ziv compression: an update

Gossip

IPv4 provides you with a universal address space and a low-latency unreliable datagram sending facility. Put the seven bytes of a destination IP address, protocol number, and port number into the head of an IP datagram, calculate the checksum, and hand it to your nearest router, and it'll likely get delivered to its destination. This is a seriously powerful primitive, enabling you to knit together all kinds of disparate unreliable transport networks into a single giant network: the inter-net.

Unfortunately, the only application that this directly provides is voice-over-IP, or maybe unreliable instant messaging. TCP is a little higher-level in some sense: it gives you a virtual serial-port connection or bidirectional pipe. This is still very low-level for many applications.

A gossip protocol is a way to share data among a group of participants despite unreliable and intermittent connectivity among them; whenever two succeed in rendezvousing, they interchange information. Typically gossip is used to converge on a gradually-growing set, but you can substitute any CRDT (see above) for the graduallly-growing set.

Gossip is no higher-level than TCP, and indeed many routing protocols are gossip-based. You can run it on top of other protocols, of course, and Bitcoin (see above) distributes both transactions and mined blocks using gossip.

Generally gossip protocols are resilient against information loss, but not against information flooding, and flooding can be the effective equivalent of loss. This typically means that either you must be able to identify and disconnect flooders, for example in a gossip-net

including only your laptop, your phone, and your server; or you must be able to drop messages when a flood is in progress, resulting in information loss. Bitcoin ameliorates this problem using hashcash (the proof of work being the hash of the newly mined block) but this approach doesn't ensure that legitimate messages will be propagated; it can raise the price of propagating them too high.

Algebraic incremental updates

Several of the earlier items in this document talk about incrementally updating the result of a computation by means of memoizing small pieces of it and only re-executing the parts affected by an input change. However, in some cases, a more efficient approach is available; Manuel Simoni wrote about it a few years back in the case of incremental MapReduce.

Doing a MapReduce incrementally with the memoizing approach is possible, of course.

The map function runs on an input chunk and produces a sorted mapped-input chunk in the filesystem; this file is in a sense a memoized result of the map function, or rather a whole pile of memoized results, and a small change to the input file generally should produce a small change to this output file, and it can do so efficiently if you can figure out what part of the output file corresponds to unchanged input.

The reduce function is a bit trickier, since it's running on an arbitrarily large set of mapped-input chunks with the same key. We can take some advantage of the fact that those chunks are in an arbitrary order, so the reduction is more or less required to be associative and commutative, so we can apply the reduction operator in a tree topology, which is an optimal configuration for memoization to be able to minimize the work for an input change, to O(log N).

However, in many cases, the reduce function is not merely commutative and associative; it also admits an inverse. (Min, max, and other quantile functions are the usual exceptions.) In these cases, a much more efficient approach is available, which processes input changes in O(1): when a mapped-input record goes away, you update the reduction function's output with its inverse; when a new mapped-input record is added ("inserted" into reduce's input), you simply update the reduction function's output with it; and mutations can be handled as a removal composed with an insertion.

This is also the way that relational databases normally handle index updates. They do not recompute the index for a table when a value in an indexed column changes, not even reusing memoized mergeable index blocks built from unchanged blocks of records; they delete the old value from the index and insert the new one. In this case, the "reduction" operation could be considered the updating of a sorted master file with a sorted update file containing insertions and deletions, and sometimes it is actually implemented this way, accumulating updates in a "side file" until they are large enough to be efficiently applied.

I'm calling this approach to incremental updates "algebraic" because it depends on algebraic properties of the reduction function: that it is commutative, associative, and admits an inverse, making it an Abelian group operation, I think. XXX

The old trick of compositing on-screen objects using XOR is

another example of this: to move a sprite, you would first quickly XOR it with the pixels at its old position, and then at its new position, rather than calculating a bounding box for the update that needed to be slowly redrawn from some kind of scenegraph. METAFONT's representation of filledness (each pixel contains a count of how many times it's been painted; 0 is white, anything else is black) would also be amenable to such an approach to animation, perhaps without the overlap artifacts that attended the XOR approach.

Are there other cases where such an algebraic property of a reduction or combination function could be exploited to get rapid incremental updates of cached computation results? Is there an automatic way to discover them, perhaps through abstract interpretation, thus avoiding the need to explicitly program the algebraic incremental update in each case?

Compile-time instrumentation and object-code instrumentation

A lot of the techniques discussed here involve augmenting some kind of program with extra new functionality: maybe running it forwards and backwards, or changing its regular number arithmetic into arithmetic on intervals or Jacobians. In an interpreted object-oriented language, that's fairly easy: instead of passing in a Number object, you pass in an Interval object or whatever. But what if you're dealing with code that isn't interpreted, maybe because you want it to run fast?

You can abstractly interpret machine code, of course, amounting to a sort of augmented emulator. Valgrind and AFL show two approaches to this problem: AFL inserts extra tracking code into the program at compile-time, and Valgrind inserts it into the program's machine code. A framework for this kind of instrumentation allows you to perform such analyses on arbitrary programs, without having to build the programs around the particular kind of abstract execution that you want.

I am not yet sure how useful this is, but I suspect that the answer is "very useful indeed". With this kind of technique, you can in theory build programs or subroutines with any existing toolchain that produces machine code, then treat them as any of a variety of abstract models.

This might seem like a very difficult thing to do, and indeed Valgrind is a monument to great software engineering, but I think you can probably get a significant fraction of the way there with relatively little effort. The /bin/ls installed on this netbook is about nineteen thousand instructions; half of those are mov, movl, or jmp, and getting to 18000 involves also call, lea, je, cmp, test, add, nop, jne, pop, sub, push, xor, ret, cmpb, movzbl, movb, sete, and or. If you handled common instructions like those and the other control-flow instructions, and you were running on a 386-family CPU, you could probably have a slow general-purpose path for the instructions you didn't write special handling for, simply by snapshotting the registers, jumping to a piece of code that has the instruction, and then snapshotting the registers again to see what changed.

This is a fair bit of code, but a lot less than a decent C compiler.

Partial evaluation

A lot of software optimization techniques have to do with specializing a generic algorithm for a particular set of circumstances. A polymorphic multiplication is slow; a 32-bit by 32-bit integer multiply is faster; a 32-bit integer multiply by the number 10 is potentially faster still. Automatically deriving these simpler versions, given the general case and an argument to hold constant, is "partial evaluation" or sometimes "specialization". A lot of the C++ STL could be thought of as partial evaluation of ad-hoc polymorphic operations: you take, say, a generic sorting algorithm and you specialize it for a particular container type, which itself is a generic container type specialized for a particular element type. In a language like Smalltalk, all of these algorithms and containers might be equally polymorphic, but with the type decisions made at run-time rather than compile-time.

An interesting aspect of this class of optimizations is that they are often applicable at the source-language level, rather than requiring a separate lower-level language to express them in, as register allocation often does.

The three Futamura projections are a particularly recursive application of this concept.

In the first Futamura projection, an interpreter specialized for a particular input program becomes an executable version of that program, an approach which may be useful even in its crudest form if your objective is merely simplifying installation; but given the mythical Sufficiently Intelligent Optimizer, is a rigorous way to automatically derive an executable from a source program and an interpreter.

Of course, automatically deriving an executable from a source program is compiling it. So if you partially evaluate the partial evaluator (!) with respect to the interpreter argument, you have generated a compiler from the interpreter, again automatically. This is the second Futamura projection.

Thus, if you partially evaluate the partial evaluator, whose two arguments are a program and an input to hold constant, with respect to its program argument, you have generated a compiler-compiler, which will convert an interpreter for any language into a compiler for that same language.

In a sense, this is cheating a little bit: your "compiler" can only run on the same platform the interpreter ran on (and which the partial evaluator was equipped to understand and optimize code for), so this doesn't work for cross-compilers, and somebody had to write the executable code to perform each of the operations in the interpreted language — either using another compiler, or in machine code, or the code for whatever platform the interpreter was written for. Typically people write partial evaluators for "nice" platforms like Scheme or C instead of hairy platforms like AMD64 or Forth, so this doesn't really help you with compiler bootstrapping, by itself.

It could help enormously with compiler optimizations, though, and compiler optimizations expand the scope of high-level constructs that you can write without an unacceptable loss of performance.

Implementing partial evaluation in a useful way involves program slicing — tracing the flow of values through the program from the fixed argument — and also abstract interpretation, since often the

useful statements that we can deduce about intermediate results in the program are not their exact values but merely particular predicates that are true of them at particular points in the program.

From another point of view, mechanical partial evaluation allows us to move computation freely between compile-time and run-time, letting us metaprogram in exactly the same language our run-time program is written in, and indeed mixing code at the two times freely. (Under some circumstances such an intimate commingling might be undesirable — for example, if you want to be able to predict the run time or memory usage of the output program. But perhaps you could make queries or assertions about the compiler output to fill this gap.)

A very similar kind of partial-evaluation-at-run-time is what's behind Acar's algorithms for self-adjusting computation: you have a hoard of precomputed results flowing from the input arguments that didn't change, and you need only compute the results that flow from the arguments that did. This suggests that cross-fertilization between the approaches may be fruitful: any techniques that can accelerate or simplify one of them are likely to be applicable to the other. Self-adjusting computation is in some sense more powerful, in that its hoard of memoized results allows an efficient response to a change in *any subset* of the inputs, rather than just *one particular subset*; but partial-evaluation systems can use specialized machine operations for the interactions between the fixed and variable subsets of values.

In particular, you may be able to generate a partially evaluated program by taking an execution trace of a self-adjusting program with respect to a particular set of changed input values. You may have to use either abstract execution (changing the input values to "unknown 1", "unknown 2", etc.) or systematically explore the space of possible sequences of taken branches (like American Fuzzy Lop, perhaps, or perhaps using a more systematic approach of backtracking to before each branch and taking it the other direction, with some kind of conservative trace merging in order to keep trace proliferation finite. You can either reason backwards from a branch condition to find inputs that will lead to it, or you can conservatively assume that if conditional X was computed transitively from input Y, then it's possible for input Y to make it go either way, and force it to go the wrong way, thus generating a conservative approximation of the possible Y-driven control flows.)

How wide a spectrum of optimizations can partial evaluation take over? What kind of language would be best suited for use with it?

Linear algebra

Matrix multiplication, that kind of thing. Maybe this is too obvious to mention, since it's fundamental to 3-D rendering, to statistical computing, to scientific computing in general, and so on. Still, it's not applied to as many things as it could be, in part because it's often considered purely numerical in nature.

As one example, the table of defined symbols in an executable or library could be thought of as a vector along a dimension of symbols, with the signal's definition (usually a memory address) at each point. The table of *undefined* symbols (unresolved relocations) in an object file (including an executable or library) is a (sparse) matrix whose rows are symbols and whose columns are memory addresses that refer to those symbols. Multiplying the symbol table through the

relocations matrix gives you a (sparse) vector of values to be added to the object file. In this way, a general-purpose parallel, distributed, or incremental sparse-matrix-multiplication algorithm provides you with a parallel, distributed, or incremental linker.

(In practice, you might have different relocation types, and on the 386, the addresses might not be aligned, but you nevertheless need to compute carries, so it's not quite as simple as regarding the object file as a large dense vector of words some of which contain memory addresses.)

Convolution

Convolution is a well-established powerful primitive for digital signal processing and the mathematics of linear time-invariant systems, in particular because of two very useful properties for improving computational efficiency:

• **Closure under composition.** Any linear time-invariant transform of a signal can be represented by convolution with some kernel, and convolution with any kernel is a linear time-invariant transform. This means that convolutions are *closed under composition*. In particular, linear time-invariant transforms of *discrete* signals can be represented by convolution with *discrete* kernels, and the kernel representing the composition of some discrete kernels has a length that is the *sum* of their lengths. This means you can compose together an arbitrary sequence of convolutions into a single convolution, then apply that convolution in a single operation. This is analogous to the closure properties of matrix multiplication, which is so useful to 3-D rendering; of arithmetic operations on the generator representation of continued fractions, as explained in HAKMEM, which is useful for exact arithmetic on all computable real numbers; and of interval arithmetic.

• **Pointwise product implementation.** Convolution is homomorphic to pointwise product under the Fourier transform, which is to say that the convolution of two functions is the inverse Fourier transform of the pointwise products of their Fourier transforms, which means that convolution can perform frequency-selective filtering. (This is the "convolution theorem".). Also, the Fourier transform of the pointwise product of two functions is the convolution of their Fourier transforms, which you can intuitively derive from the angle-sum trigonometric identities, and which gives rise to useful properties like superheterodyning and the limited bandwidth of the sidebands of amplitude modulation.

• **Commutativity.** When the underlying pointwise multiplication operation is commutative, convolution is commutative.

Because of the product-multiplication property, the convolution `f*g` of kernels `f` and `g` will not contain any frequencies that are not present in both `f` and `g`. (This may be useful for optimizing the implementation of the convolution operation by computing with downsampled versions of the kernel.)

Entirely in the domain of time-domain signal processing, you could imagine an evaluator for a complex expression DAG of convolution operations, discrete samples, repetition operations, weighted sums, delay operations, and domain restriction that used strategies similar to those of a SQL query optimizer to find an efficient evaluation

strategy for the expression within specified precision and performance constraints, using any number of the AI search techniques mentioned above. I don't think anyone has done this yet; typically the evaluation strategy is programmed manually by some dude in Matlab.

Convolution is used in particular for signal filtering (including image blurring and sharpening) and for simulating the effect of some physical system, whether optical, electrical, or audio, in particular including audio reverberation.

But there are domains a bit outside of what we usually think of as signal processing that could also benefit from convolution.

You can synthesize a xylophone tune by convolving the score, a time and frequency representation composed of impulses whose dependent variable is note volume, with a instrument patch, and extracting the frequency=0 slice. If frequency is represented using equal-temperament semitones, other parallel slices are transpositions.

The scores for some simple kinds of canons are the convolution of the score for the dux with a repetition and transposition function.

If, instead of using impulses, you use white noise all along the duration of a note, you can synthesize some kinds of sustained instruments, like violin and pipe organ, from windowed samples.

Instead of compositing the "glyphs" of a "sound font" into a temporal representation, you can composite the glyphs of a font into a window by convolving the font with a signal whose (x, y, glyph-id) impulses place and color individual glyphs. If you composite them into a 3-D opacity field instead, then you can get font resizing, drop shadows, pen width and shape, crude bolding, and glyph composition from features such as serifs, stems, and stroke thicknesses into the bargain, if you add an additional dimension of feature type or stroke thickness.

Similarly, an animation could convolve cels with (x, y, z, t) paths, perhaps followed by the same kind of 3-D or 2½D projection to provide opacity. It might be desirable to do the convolution in a space with five to seven dimensions, perhaps including rotation or even stretching, in order to express more of the desirable operations of animation.

A potential great advantage of this kind of unified convolutional architecture is that it is relatively practical to allow the user to ask "Why?" about a thing they see and get a reasonably comprehensible answer, without incorporating a lot of special-purpose mechanisms for answering "Why?" in each case.

Much of the above might not be practical with existing convolution code optimized for discrete samples, high density, and low dimensionality, but techniques based on interval arithmetic, query optimizers, and/or automatic differentiation (see above) could make them practical.

Discrete convolution is an operation built from two fundamental operations, multiplication of corresponding elements and summing the products. The associativity property that give rise to its closure under composition would seem to require some kind of algebraic ring to operate over, although I don't know that it depends on *all* of the ring postulates. Schönhage–Strassen multiplication is one application of convolution over a finite ring $\mathbb{Z}/n\mathbb{Z}$ (i.e. the Galois field $GF(p^n)$), but we could also imagine, for example, computing a convolution

over quaternions or 4×4 matrices representing three-dimensional transformations. (Such a convolution is not commutative. Can you compute it with the Fourier-transform trick anyway?)

But what about other data structures commonly used in programming? Can convolution be applied productively to things like polynomials, bits, strings, lists, binary relations, and finite maps (dicts), with some other operations standing in for multiplication and addition?

The most common example is Elias and Viterbi's convolutional coding for error correction, which XORs together certain plaintext bits from a sliding window to produce coded bits. A 1:1 convolutional code is exactly a convolution of a bitvector kernel with the plaintext, using AND and XOR (which are multiplication and addition in $\mathbb{Z}/2\mathbb{Z}$), but by itself is useless; the normal procedure is to interleave several such 1:1 codes, so that one bit from each code forms a "codeword". (And then, typically, you throw away some of the bits.)

You could, again, consider the different kernels to be displaced along an axis perpendicular to the text, and then two-dimensional convolution in the bit ring produces the bits of the convolutional code spread out on a two-dimensional plane.

http://www.math.nthu.edu.tw/~amen/2011/101219-4.pdf is about "polynomial division by convolution" XXX

http://stackoverflow.com/questions/22683195/boolean-convolution-algorithm XXX points out that you can convolve bitvectors in O(N log N) time by using the number-theoretical transform (FFT in $\mathbb{Z}/n\mathbb{Z}$) for a ring size larger than the shorter vector, and then saturate the results a the end.

http://mathoverflow.net/questions/10237/does-the-convolution-theorem-apply-to-weaker-algebraic-structures says, "it is a major open question in discrete algorithms as to which algebraic structures admit fast convolution algorithms and which do not." … "A substantially subquadratic algorithm for (min,+) convolution would (to my knowledge) imply a subcubic algorithm all-pairs shortest paths in general graphs, a longstanding open problem." Also mentions infimal convolution.

Optimizing compilers to machine code

Optimizing compilers are of course a standard tool in the programmer's toolkit since FORTRAN I, but usually we use them, but our programs don't. But there are a lot of cases where our programs *could* use them.

This is all mostly about constant-factor performance improvements, so you might want to skip it if you don't believe in those.

Suppose you've parsed a SQL query and derived the best query plan you can for it, but you estimate that it's still going to have to examine 5 million rows. What should you do?

Well, one possibility is to compile your query plan into C, then compile it into machine code with a C compiler, then run it. TCC can compile hundreds of lines of code per millisecond, and even GCC can compile a few lines of code per millisecond. The result typically

is only faster than interpreted code executed by an optimized interpreter by constant factors, but the constant factors can be substantial, like 5 to 30. It can make the difference between executing the SQL query in 15 seconds and executing it in 500 milliseconds.

There are a couple of obstacles to using this technique ubiquitously. One is that the result is potentially harder to debug, because of the extra level of indirection. Another is that if you have to write the compiler yourself, it's considerably harder than writing the interpreter, especially when you have to port to a new architecture. A third is that typically there are some fiddly bits relating to mprotect() and runtime dependency on the compiler and getting memory that's both writable and executable.

The main one, though, is the compilation speed. If an ad-hoc SQL query takes 100ms to run interpreted, you can't justify spending 200ms in a C compiler to get it to run in some shorter time.

A simple technique that improves compilation speed substantially, especially at low optimization levels, is to emit an abstract syntax tree rather than a sequence of bytes. This is the approach taken by Lisp environments that provide `eval`; its argument is an S-expression, which is essentially an AST. Unfortunately, compiled Lisps are usually kind of slow. Racket and Clojure may be shaping up to be exceptions.

There are several different practical ways to do this today in different programming environments:

- ObjectWeb2 ASM is a very widely used system for rapidly emitting JVM bytecode, which is then "interpreted" by compiling it to machine code and then, if it becomes a bottleneck, by running the bytecode through an optimizing compiler.
- LLVM, although its compilation is fairly slow, was originally intended for this purpose, and it can still be used for it, even though it's been mostly retargeted for ahead-of-time compilation.
- LuaJit compiles arbitrary code in a superset of Lua with many low-level constructs added to machine code, for amd64, i386, or ARM.
- TCC is very fast and supports most GCC extensions to C; it has a compile-into-memory mode, or you can generate a .so dynamic library and use `ldopen` to read it in. The emitted code is on the order of 5 times slower than code emitted by GCC with optimization, and only marginally worth than GCC without optimization. Even GCC is capable of compiling a few hundred lines of code in well under a second, as long as you don't `#include` big header files.
- Even compiling machine code into memory yourself isn't that hard, though it's nonportable; you can implement a simple system in C in a few hours and a few hundred lines of code. It's handy to use `gcc -fverbose-asm -Wa,-adhln=foo.lst foo.c` in order to see what constructs GCC generates.
- `eval` in PyPy also compiles Python code into optimized machine code, and MyHDL uses this to get enormous speedups for simulating digital hardware.

Of course, compilers are related to nearly everything else I've been talking about.

Memoization is such a helpful technique for compilers that we've been caching compiler output for decades (using automatic

dependency tracking and deterministic rebuilding), and `ccache` is a content-addressable blob store for memoizing C compiler output. A lot of the tradeoffs with compilation speed become less of a problem if you can memoize the compilation result, like a cached SQL query plan.

There has been experimental work on building distributed compilers that save the optimizations they've discovered in a common optimization store so that they can be reused later without having to search for them. (I forget who wrote that paper.) Users sometimes switch browsers because of the performance of their JS compilers. Compilers tend to be heavy users of pointers, and they tend to not have to run under strict deadlines or tight resource constraints, so FP-persistent data structures are a good fit for them. Incremental compilers, which would only recompile the part of your code that you'd edited, used to be popular; of course, it can be difficult to efficiently identify which part that was, and perhaps rolling hashes can help. (Incremental linkers are still popular, and the boundary between compilers and linkers is somewhat fluid.) Compilers frequently use backtracking search and constraint solving to find applicable optimizations, and memoized backtracking gives us PEG parsing. Nondeterministic search over compiler executions (or, rather, interpreter optimizations) is what miniKanren uses to automatically generate programs with certain properties. Not only can you use partial evaluation to implement a compiler; you can also use a compiler to implement partial evaluation. Indeed, there is a very fine line between compilers from a language into itself and partial evaluators. Some virtual machines, like QEMU (by the author of TCC), are implemented largely as compilers from the machine code of the guest machine into the machine code of the host machine.

Topics

A cute algorithm for card-image templates

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

There's a trick I think I saw originally in REXX, and which I think originally comes from the IBM mainframe world.

Suppose you have a record with some fixed format and you want to reformat it. For example, you have this:

```
199712100036325SITTLER   KRAGEN
```

And you want to reformat it to this:

```
KRAGEN    SITTLER    $00363.25  10/12/1997
```

The thing that would make this easy would be if you could write a couple of "picture" lines showing the desired input and output, and have software apply the transformation automatically:

```
199712100036325SITTLER   KRAGEN
19YyMmDd23456780PQRSTUVWXopqrstuvwx
opqrstuvwxOPQRSTUVWX $23456.78  Dd/Mm/19Yy
KRAGEN    SITTLER    $00363.25  10/12/1997
```

So far that's nothing terribly special. You use the correspondence of the characters in the before-and-after picture to show where to move the input characters around to in the output.

The special part is that it turns out you can implement this with a simple character substitution, the same kind of thing you would use to transform uppercase to lowercase or vice versa, or remove accents from ISO-8859-1 text for accent-insensitive comparison, or translate between EBCDIC and ASCII. Here's what it looks like in Python.

```
>>> import string
>>> the_input = '199712100036325SITTLER   KRAGEN    '
>>> beforepic = '19YyMmDd23456780PQRSTUVWXopqrstuvwx'
>>> afterpic  = 'opqrstuvwxOPQRSTUVWX $23456.78  Dd/Mm/19Yy'
>>> cipher = string.maketrans(beforepic, the_input)
>>> string.translate(afterpic, cipher)
'KRAGEN    SITTLER    $00363.25  10/12/1997'
```

So first we compute a character substitution that would convert `beforepic` into `the_input`. Then we apply that substitution to `afterpic`, and we get the desired output.

It's not a very versatile trick --- all the characters in `beforepic` have to be distinct, so it can't work in this form for anything over 256 bytes, it only handles fixed-width fields, and you can see I had a hard time coming up with reasonable-looking characters to use in the templates even in this small example. But the clever thing about it is that, given the existing ability to translate a string of characters according to such a table of correspondences, and the ability to construct such a table from a before and after string, it only takes a

couple of lines of code.

Topics

Recursive curves

Kragen Javier Sitaker, 2019-06-10 (5 minutes)

Thinking about the issues in Dercuano drawings (p. 64), I came up with some ideas that I think will be pretty interesting for interactive drawing. As usual, they are (sort of) algebraic and recursive.

The fundamental *spatial* element of the system is a curve, what you might call a path or a polyline. A curve is a continuous function from some interval of t points in time to (x, y) points in space, so it has direction and potentially variable velocity and duration as well as positions. (Maybe it also has a continuously varying pressure.) All curves start at time $t=0$, but they may have different end points.

The easiest way to create a new curve is to draw one with the mouse (which, in Some musings on applying Fitts's Law to user interface design and data compression (p. 1164), I found had a bandwidth of about 6 bits per second) or a multitouch touchscreen (which I haven't really tested yet but am hoping to find will have better bandwidth; it does have pressure). But you can also create curves that are straight lines or circles.

Each curve also has some kind of visual attributes, like width, color, transparency, blurriness, and some kind of noise texture, for each of stroke and fill. For right now, though, I'm less interested in those than about the purely positional ones.

Given a curve C, there are several unary operations that give new curves: C.reversed, which is the same positions in reverse order; C.normalized, which is the same positions with their time interval compressed or stretched to $t \in [0, 1]$; C.closed, which adds an instant straight-line jump back to the start; and C.constant, which makes the velocity along the curve constant, but doesn't change its duration. If curves also carry pressure information, there's also C.monoline, which sets the pressure to unity; C.brush, which sets the pressure to the reciprocal of the velocity; and C.invbrush, which sets the velocity to the reciprocal of the pressure. I want on-screen buttons of some kind to select these operations.

There are also operations to combine a curve, or set of curves, with a point: C + P translates the curve by the (x, y) coordinates in P, while P·C rotates and scales the curve (around its start point?) by the parameters in the point. I want interactive operations for invoking these with a point I specify interactively using the mouse or using pinch-zoom with two fingers. These two operations $P_0 \cdot C + P_1$ form a "frame of reference".

Each point along a curve can be associated with a frame of reference in different ways, and another curve can be transformed by that frame of reference. First, there's a translated frame of reference, where the rotate-and-scale part of the transformation is the identity. Second, there's a translated-and-rotated frame of reference, where the rotate-and-scale part of the transformation scales by unity but rotates so that the tangent forward along the curve is always in the same direction, or in some arbitrary direction when the tangent doesn't exist. Third, there are translated-and-rotated frames of reference where the scale is taken from the velocity or pressure of the curve.

The operation C_0.interpolate(C_1, C_2, N) produces a set of N

curves. The first curve is C_1 translated and rotated to the beginning of C_0, and the last curve is C_2 translated and rotated to the end of C_0. The curves in between interpolate smoothly between them. Another similar operation does the same thing without the rotation. These operations are invocable in a direct-manipulation kind of way; initially C_1 and C_2 are the same, and N can be interactively adjusted up and down.

(Hmm, maybe this is the wrong formulation? Maybe I actually want to add C_1 and C_2 to the definition of C_0? In that case the objects become drawings rather than curves? Maybe I want to be able to interactively see the interpolated curves as I'm drawing C_0? Maybe drawing C_0 is, at least normally, a context-menu operation on C_1? What about adding noise to the interpolated curves?)

The idea is that a variety of visual repetition with variability, including things like grids, hatching on one side of a line, wood textures, starbursts, and tree branches, becomes straightforward and easy to express and to adapt, while capturing the real Kolmogorov complexity of the drawing in the data structure that is built up.

Topics

Transmitting low-power TV signals around your house via RF modulation with an SDR

Kragen Javier Sitaker, 2019-12-01 (6 minutes)

Suppose you want a multi-monitor setup. As Elda King points out your monitors are more convenient to use if they're wireless, and perhaps more trustworthy (and certainly lower latency) if they have no RAM. Fortunately, there's a device that already exists with these attributes: the analog TV. They are currently being discarded in enormous numbers.

What would it take to use many analog TVs as prosthetic wireless monitors for your computer system? Merely an RF modulator, ideally on unused TV channels, and an antenna that's just strong enough to reach across the room. If it uses near-field transmission it might be able to be even more local and more efficient than the inverse-square law would suggest. RF modulators for VHF channels 3 and 4 were common home-computer accessories around 1980, but using cables rather than antennas to connect to the TV.

You don't get a whole lot of resolution with the analog TV standards: NTSC is 525 lines, while PAL and SECAM are 625 lines, with 4:3 aspect ratios --- 700x525 or 833x625 in theory if you use square pels, but some of that is lost to overscan, the HBI and VBI. The 625-line resolution has a 49-line VBI, so you only get 576 lines on the screen, or 768x576 if you use square pels; NTSC has only 486 visible lines out of its 525, or 648x486 with square pels. And because this is *interlaced*, one-pixel-thick letters can be hard to read as they flicker at 25 or 30 Hz. Analog HDTV never hit the mass market anywhere but Japan, so if you want higher resolution, you need to transmit DVB.

Still, if you're transmitting many frequencies at once, you can run many TVs at once.

NTSC is amplitude-modulated, which is why snow is such a problem, and the whole signal is 6 MHz wide. The VHF channels are in the 30 to 300 MHz "VHF" band, while the UHF channels are in the 300 MHz to 3 GHz "UHF" band. In North America, channels 2 to 6 occupy 54 to 88 MHz, 6 MHz each, while channels 7 to 13 occupy 174 to 216 MHz, again 6 MHz each, while UHF channels 52 to 69 occupied 698 MHz to 806 MHz until 2009, channels 14 to 20 occupied 470 to 512 MHz, channels 21 to 36 occupied 512 to 608 MHz, channel 37 was reserved for mostly radio astronomy, and channels 38 to 51 occupy 614 to 698 MHz until 2020; unlicensed devices are officially permitted in the 652-663 MHz range, including part of channels 44 and 46, and all of channel 45. Until 1983, 806 MHz to 890 MHz was UHF channels 70 to 83, so older TVs might be able to receive those, too.

Aside from channel 45, probably any transmission on these frequencies is illegal in the US unless you do it inside a Faraday cage.

The bandplan here in Argentina is presumably different, but harder to get information on.

A quarter-wave monopole antenna at 806 MHz would be 93 mm, so efficient antennas are compact and easy to build, and the "far field" might be the other side of the room. UHF frequencies transmit almost entirely by line of sight and are even substantially attenuated by building walls. The VHF frequencies, below 300 MHz, have much better wall penetration and can take advantage of ground bounce; at the lower extreme of 54 MHz, a quarter-wave monopole is 1.39 m long, which is inconveniently large, but by the same token, near-field communication can extend out several meters --- the Fraunhofer distance $2D^2/\lambda$ for a 2-m antenna would be 1.44 m, and a 4-meter loop antenna might be practical, as discussed in Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509), which would give you a near-field range of almost 6 meters.

You only get about a third of a megapixel per TV channel, so you need at least three channels, 18 MHz of bandwidth, to get a megapixel, and thus at least 36 Msps, better 50 Msps. The USRP N200 and N210, like the USRP2 before them, have dual DACs running at 400 Msps. So they should easily be able to handle that DAC demand if you can compute the waveform; indeed, at the DAC level at least, they should be able to transmit on 66 contiguous NTSC TV channels at once, covering the entire UHF TV band!

Of course it would be foolish to attempt to transmit a signal in this way that you wanted to maintain confidential, since your neighbors can probably tune in to it.

Vladislav Fomitchev KM4VTH demonstrated this more or less working in 2018 using GNU Radio, a HackRF One (US$600 in Argentina), and a signal flow graph he constructed, and marble has demonstrated doing something similar with GNU Radio, a HackRF One and PAL. Argilo has published a flow graph for transmitting ATSC on 438 MHz with GNU Radio and a BladeRF; the BladeRF has 61.44 MHz sampling and 2x2 MIMO and costs US$420 from SparkFun but isn't available in Argentina.

Topics

- Electronics (p. 3430) (138 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Communication (p. 3382) (19 notes)
- Displays (p. 3414) (13 notes)
- Radio (p. 3676) (8 notes)
- Sdr (p. 3698) (2 notes)
- Tv

Harvesting energy with a clamp-on transformer

Kragen Javier Sitaker, 2013-05-17 (7 minutes)

Clamp-on ammeters have a ferromagnetic "clamp" that encloses a wire in a closed magnetic circuit; the magnetic field induced in the ferromagnetic material is proportional to the total net current flux enclosed by the clamp, and you can measure this field precisely with a Hall-effect sensor, thus distinguishing wires carrying a load from wires that don't.

You could use the same approach to build a low-power electronic device that powers itself by leeching energy from the magnetic field around a current-carrying wire, without needing a direct electrical connection. This could enhance safety and reliability and ease installation, since you can "plug in" such a device without making a direct electrical connection, and if the device shorts out, it won't cause an electrical fire. Not only could it harvest power without ever coming in contact with the wire, it could harvest power without even coming near the wire; it need only enclose the wire with a loop of ferrite without also enclosing its return path.

Effectively this is a clamp-on transformer, with the "primary winding" of the transformer having only one turn of wire. If that wire is normally carrying, say, 100 amps, then the secondary winding of the transformer with, say, 1000 turns, could draw up to 100 milliamps; but if the total available voltage to be dropped through that one turn is 240 volts, the secondary winding would need to handle 240kV, which is difficult. If we limit the secondary winding voltage to 10V, then the voltage drop on the primary will be an insignificantly small 10mV, and the total power being transmitted can be up to 10mV * 100 A = 1 watt; the secondary winding then can draw up to 1 watt / 10 volts = 100 mA still. That's enough power to allow the use of a very simple power supply (say, a diode, a small capacitor, and a 7805) and a relatively inefficient electronic device.

If you don't enclose the wire in ferrite, but simply put a ferrite rod near the wire and perpendicular to it, you'll be able to harvest orders of magnitude less energy, but installation would be even easier.

Inductive coupling is only one possible way to harvest energy from power-line fields. Capacitive coupling is also feasible. However, the 60Hz of typical line power presents an extremely difficult problem for capacitive-coupling energy harvesting. If you manage 10pF of capacitive coupling to a power line field, which is probably about all you can hope for, your $1/\omega C$ capacitive reactance is 270 megohms at 60Hz. The other side of your power-supply input is then going to be your capacitive coupling to ground at around 100pF, I think. (Would it be better or worse if you had a wire to ground? Better, I assume.) You're going to need a power supply with an input impedance in the hundreds of megohms in order to be able to take advantage of that. Megohms or tens of megohms is probably feasible with CMOS; hundreds of megohms is probably not.

However, all is not lost! Fluorescent lights and high-intensity discharge lights powered by AC produce much-higher-frequency

harmonics, some one to two orders of magnitude higher in frequency. This brings the capacitive reactance down to the megohms to tens of megohms range you need.

So you could parasitically capture a significant fraction of the line voltage, but only in the high-frequency harmonics produced by the discharge.

Another approach, which would be more practical in areas without electrical lines, is to harvest atmospheric electricity, either from actual lightning strikes or directly from the atmospheric voltage gradient. During a lightning storm, atmospheric voltage gradients can reach 100kV/m, only an order of magnitude below air's ionization strength. In clear weather (the "fair weather condition"), it falls three orders of magnitude to some 100V/m.

A lightning rod struck by lightning has some 30kA available, but only for tens to hundreds of microseconds. If you erect a ten-meter lightning rod, you could in theory harvest up to a megavolt of the lightning's voltage --- a few megajoules per lightning strike. If you put your lightning rod on a mountain top, you could perhaps get several lightning strikes per month, for a total average power on the order of a watt.

Harvesting the energy of a lightning strike, however, seems like a really difficult problem. A 1000:1 step-up transformer as discussed above could reduce your 30kA to some 30A, at the cost of boosting your megavolt to a gigavolt. If you put a series of these transformers along the lightning rod's path to ground, you could drop only a tiny fraction of the voltage through each one, making the situation more manageable. If you can dump this massive amount of power through a low-resistance path into some kind of resonating circuit, you could then store it for milliseconds up to seconds in order to harvest it at a more reasonable pace.

Harvesting the atmospheric voltage gradient directly seems much more feasible, and I've heard you can do it as simply as holding up a spent fluorescent light tube in one hand in a thunderstorm. In the absence of a separate source of ions, you need a corona discharge to couple your wire to the atmospheric charge, which means that you need points sharp enough that the electric field intensity at the point is above air's ionization strength. If you're working with, say, only 1000 volts, then you need micron-scale conductive sharp points to generate ionization, and preferably enough of them to support a substantial ionic current. By contrast, if you have 400kV --- say, a two-meter fluorescent tube plus a two-meter-tall person, in a thunderstorm with 100kV/m --- then any conductive point radius below around 40cm will produce a sufficient field, if I remember my electrodynamics correctly. The points of the prongs on the end of the fluorescent tube are on the order of 0.4 mm in radius, so they should work down to about 400 volts.

The question, then, is how much current and thus power you can expect to draw at these voltages. It's observed that the fluorescent lamp in the thunderstorm experiment will simply flash periodically as the lamp's parasitic capacitance to ground charges sufficiently to ionize its contents and discharge the capacitance. If I SWAG this, we have a 1ms flash per 5 seconds at 40 watts with a 200V breakdown voltage, giving 40 microamps average charging current; or maybe charging a 1pF parasitic tube capacitance to 200V in 5 seconds, giving

about 40 picoamps. This difference of six orders of magnitude suggests that I don't know enough about the problem even to guess.

Topics

Analemma sundial

Kragen Javier Sitaker, 2019-07-05 (11 minutes)

There are a variety of sundial designs that incorporate the solar analemma in one or another form, so that they can provide the precise time according to current civil time standards — which hold that each day and each second should have the same length despite the eccentricity and consequent inconstant speed of the Earth's orbit and, thus, the solar day.

A spot on the wall

I was thinking in particular of using the sunbeam reflected from a small round mirror to illuminate a spot on a wall; the point illuminated on the wall will vary according to both the time of day and the angle of elevation of the sun.

```
    |       |
    | light |w
    \-------|a
mirror      |l
            |l
```

Each day the sunbeam will sweep a (noncircular) arc across this wall, but from day to day the arc will vary as the elevation angle of the sun does. The total variation in elevation from solstice to solstice is about 47°; it is fastest near the equinoxes and slowest near the solstices. I roughly guess that that means that, at its fastest, it moves about half a degree per day, which is rather pleasantly the size of the sun as seen from Earth.

This means that, if the mirror is small enough, the spot on the wall will also take up about half a degree (32 arcminutes) as seen from the mirror. So, in theory, you could mark the arcs on the wall for each day of the year; they will gradually be displaced by a quarter of a day each year until being reset by the leap year.

This means that, in theory, you could mark two times of day on the wall at each point, one for when the sun crosses that point moving north and one for when it is moving south. Or you could switch out the wall twice a year, on the solstices, and mark one time of day on the wall at each point.

Since an hour is 15° (360÷24), half a degree is about two minutes, so each point along the centerline of the spot's track will be illuminated for about two minutes. This would seem to pose some difficulties for telling time with a sundial with less error than two minutes, but if the mirror is small, the spot is a well-defined circle with the size of the sun's disc, and you can see the location of its center to a precision of something like a tenth of its width; this should permit a timekeeping precision of something like ten seconds.

For those ten seconds to constitute about a millimeter of motion, the distance from the small mirror to the wall needs to be about 1.4 meters, although perhaps this can be productively folded up using additional mirrors. At this distance, the 32-arcminute sun disk will project as a 13-mm-diameter circle convolved with the shape of the

mirror. (If you want a sharp boundary on it without sacrificing as much brightness, you might consider using a mirror in the shape of an annulus; an annulus convolved with a solid circle has the same diameter, but a much sharper boundary, than two solid circles convolved.)

Caustics

Suppose that the wall is itself a mirror, but not a flat one. Then it will reflect the spot elsewhere, for example onto a screen, but distorted and possibly changed in direction. It can form caustics in the reflection, and these caustics can have stronger contrasts than mere solar caustics, because the light falling on the wall comes from a smaller point source than the sun's disc. (It might be worthwhile to make the mirror subtend, for example, 8 arcminutes; at the 1.4-meter distance suggested above, this is about 3.3 millimeters. (Assuming we're using a circular mirror, not an annulus.)

To focus the spot back to a 3.3-mm point at the same distance would merely require a radius of curvature of that same 1.4 meters. The versine of half of 32 arcminutes is about 1.1×10^{-5}, so if you made the 13-mm spot a spherical reflector, its center would need to be cut deeper than its edges by about 15 microns, a number which varies only a little as the focal length and direction vary. See Caustics (p. 1619) for some notes on how to shape nearly-flat surfaces to arbitrary shapes with this kind of precision.

In particular, it wouldn't be that hard for a series of facets to reflect the beam to the same place on the screen as the sun's image passed over them, so that instead of scanning across the screen, the projected image stayed in the same place; but it could vary from one facet to the next. And if the facet is convex rather than concave, it could be larger than the 13-mm illuminated area on the wall, rather than smaller, though at the cost of brightness.

(And there don't need to be actual facets; you can use a smooth curve only occasionally interrupted with the kind of discontinuity you see in Fresnel lenses. Facets are just a crude discrete approximation of the problem.)

Unavoidably, though, since each point along the center of the track is illuminated for two minutes, there will be a certain amount of fading from one image to the next over the course of those two minutes.

One particularly attention-getting image to project might be the current time, written in Arabic numerals, with a colon, like the various "digital sundial" projects that exist.

This poses the problem of how to avoid a vague superposition of numerals during the two-minute transition from one facet to the next. A possible solution is to use a larger number of smaller facets, so that the facets close to the transition zone are projecting not just the current time but the *negative* of the adjacent time; on one side of the boundary of the 12:46 to 12:48 transition, for example, you would project a mostly gray image with "12:4" in white, "6" in white, and "8" in black, while on the other side, you would project the "8" in white and the "6" in black. Thus, as the preponderance of light shifted from one side of the boundary to the other, the "6" would fade to gray and be replaced by the "8".

(To keep the black image from being obtrusive a bit further over,

you'd want to counterbalance it with a dimmer and perhaps blurrier white image, etc.; I think the brightness curve ends up looking something like the derivative of sinc. Essentially you're trying to Wiener-filter out the low-pass temporal filter imposed by the sun's nonzero width in order to get a sharper transition.)

This approach to getting faster transitions by counterbalancing with inverse images probably precludes the use of *caustics* in the sense of places where the Jacobian determinant (of the position of the beam on the screen as a function of its position on the mirrored wall) vanishes, since that could easily create more brightness than you could counterbalance, but you can still vary the magnitude of that determinant substantially to vary the brightness. But your contrast ratio might be limited to 2:1, which sucks.

This poses the additional question of whether the facets would need to be so small that diffraction would pose a problem. If the individual facets were 1 mm across and were effectively planar at the level of 100-μm-diameter "microfacets", which seems feasible, the Airy limit ($1.220\lambda/D$ for a circular aperture, as explained in Caustic business card (p. 255)) would be, say, 1.22·555 nm / 100 μm, about 23 arcminutes of diffraction-limited divergence. So, yes, diffraction would start to pose a problem; the wall might need to be larger and further away, and you might need to use larger microfacets. But it's not so overwhelming that I think it makes the problem infeasible, just challenging.

Scratch holograms

Suppose that instead of using caustics, you use Bill Beaty's scratch holograms. You stick a bronze plaque on a wall, paint it with clear polyurethane, and put a peephole nearby. The reflection off the scratches on the plaque from the sun when you're looking through the peephole displays the current time.

A simple approximation, which is easy to improve on, is to divide the plaque into pixels, and add scratches to each pixel to reflect the sun at every angle where it should be lit up. As long as the scratches aren't too dense, the scratches at different angles will only interfere a little bit with each other, but it still might be a good idea to display different times on different parts of the display to reduce the "burn-in" effect of too many scratches in the same place. If the plaque is facing north (or south, if you're in the northern hemisphere *like a sucker*) and the peephole is in front of and below it, the sun will move through nearly a whole 180° arc each day, but faster close to noon.

Correcting for the Equation of Time can't be done by displaying different images at different times of year depending on the elevation, but it could be done to some extent by moving the peephole; the angle at which a point P on a scratch reflects is when it is perpendicular to the plane including your eye, the sun, and P. So moving the sun a little to the left is equivalent to moving your eye a little to the right, and vice versa. So it might be adequate to mark dates along the bottom of a viewing slit to show you where to position your eye. (Maybe a part of the plate you view from the side instead of from below could tell you what the sun's elevation is and thus what the date is.)

As with the analemma, I'm not going to do the math for the angles

right now.

The scratches, though, I will. The scratch depth needs to be at least on the order of a wavelength of light (say, half a wavelength) in order to scatter incoming light properly — the ray entering at a point should leave as a plane. It is unreasonably challenging to make the scratch walls much steeper than 45°, and indeed with the usual kind of abrasive scratches, you'll get

Topics

An IDE modeled on video games

Kragen Javier Sitaker, 2019-04-08 (5 minutes)

I was thinking about the famous Martin Shkreli screencast with Excel where he spends a few minutes working up a basic analysis of a company's balance sheet. Although in a sense what he's doing is mostly very simple, the demo is very flashy, and I certainly wouldn't be able to do it as quickly. He ends up spending an unreasonable amount of time cutting and pasting individual numbers from his browser into Excel.

Then I thought about the kids at my high school who played half an hour of Tetris per day (between classes) during a school year, which worked out to about 90 hours of Tetris practice spaced over most of a year. They reached fairly impressive speeds at Tetris, though one day I was wandering around Kobe and wandered into a video arcade where I saw some random Japanese dude doing Tetris things I had never imagined a human could do. He probably had thousands of hours of Tetris practice spaced over several years.

Typing games can rapidly improve typing speed once you've learned the basic touch-typing technique, while day-to-day typing, lacking the same time pressure, usually won't.

Programming often involves a lot of fiddling with user interfaces that don't offer a very direct way to get the result you're looking for, or determine whether you've gotten it, even for things that conceptually aren't very complicated. Some aspects of programming naturally involve deep thought, but others just involve rapid trial and error, and the higher the frequency at which this can be done, the better.

I wonder if you could hack together some kind of IDE that would enable interaction at a video-game pace, with video-game-like smoothness, and a series of exercises that would provide you with incentives to learn to use it smoothly, the way video games first guide you through a tutorial to learn their user interfaces and then use the gameplay to bring you gradually to a near-superhuman level of performance with them. Perhaps it could provide you with responsive interaction options you could use to incrementally approximate the program you wanted, getting rich, instant, and varied feedback on the program you had gotten so far and what the next possible steps would look like.

Some example mechanics:

- Extrinsically-organized prizes to guide you.
- Programming by example, as in Excel, gives you the opportunity to see an example output from your algorithm as you are developing it step by step. In some cases, it would be useful to see several outputs from different runs in parallel.
- Whether or not you're programming by example, it's useful to visualize the inputs, outputs, and intermediate values of any algorithm, whether these be test inputs or real inputs. But different kinds of data might have different kinds of visualizations. As in Excel, scalar numbers can be visualized as numbers; univariate functions could be visualized as plots or perhaps played as sounds;

bivariate functions could be visualized as images, 3-D plots, scatterplots, face plots, series of slices, and so on; tabular data can be displayed in tables; strings can be rendered as sequences of glyphs; and so on. Bret Victor's work visualizing Nile offers inspiration here.

• When programming by example, in general, you have some values and some operations on them that can yield further values. If you're focusing on one or two values in particular, there's some set of operations you're likely to apply to them to get more values. For example, if your value is a set of angles, likely operations might include elementwise sine, cosine, and tangent, as well as minimum, maximum, and cardinality. The menu of these operations can include a visualization of the operation results next to each operation, like Instagram filters.
• Tests can go red and green as your implementation changes; implementation code exercised by no green tests can be colored red, while implementation code with no test coverage can be colored grey. Generative tests can run continuously to produce new test cases, and individual test cases can be visualized.
• Quantitative visualizations of input data can be interactively edited, changing the data in real time. This suggests either using constraint-solving systems to specify the visualization or using iterative optimization algorithms to seek an input to an imperative visualization algorithm that will most closely approximate a given output.
• Algorithms learned from test data can be applied to further random data and the results visualized, or input data collated from other sources.
• Tutorials can focus on particular available algebras — integers, GF(2^{32}), strings, and so on — or particular families of algorithms.

Topics

• Programming (p. 3658) (286 notes)
• Human–computer interaction (p. 3493) (76 notes)
• Psychology (p. 3669) (18 notes)
• Education (p. 3427) (8 notes)
• Games (p. 3466) (6 notes)
• Programming by example (p. 3655) (4 notes)
• Spreadsheets (p. 3728) (3 notes)

Notes on a possible household air filter

Kragen Javier Sitaker, 2018-05-05 (updated 2018-05-15) (10 minutes)

I was thinking about filtering the air in my house, which is on a major thoroughfare with lots of diesel buses, burning that dirty high-sulfur diesel fuel which produces a lot of soot.

The standard approach, of course, is HEPA filters, which are made of disposable paper. I was thinking that a possible improvement would be to bubble the air through liquids, which could catch not only particulates but also a number of gases. Also, if you color the liquids, illuminate them with LEDs, and keep them in transparent tanks, it would look a lot cooler than a HEPA filter. You could use a series of such tanks to eliminate a lot of different particulates.

After the bubbles finish the tanks, they would need to pass through a fine screen to filter out any remaining droplets.

My apartment is 101 m³; filtering all of the air in it every six hours would thus require 4.7 ℓ/s of airflow, which is 10 cfm; as described in House scrubber (p. 248), you can get that out of a 75mm-diameter 4-watt 3000 RPM fan designed for a CPU cooler. However, that fan won't generate the head needed to bubble the air through a liquid; if we have a series of four tanks, each with liquid 100mm deep and of more or less the density of water, then we need 18 watts just to counteract the hydrostatic pressure, let alone producing the necessary flow rate through the bubble orifices. This suggests that perhaps using a few axial fans in series, each designed for a substantially higher flow rate, would likely do the trick.

XXX how do you calculate the head to produce a given flow through an orifice?

If the tanks are 10% bubble by volume, which seems like achievable and difficult to exceed by much, and the bubbles take 500ms to rise to the surface, then you need 23½ liters in each tank. If the tanks are 2 meters long, then they need to be 117½ mm wide in order to contain this much liquid.

You would need some system for replacing dirty filter liquids at some point, either manually or automatically. This is ideally done in batches to minimize the mixing of old dirty liquid in the new clean filter liquid.

Possibly useful liquids include:

- Plain water, which increases humidity, absorbs droplets of water-soluble liquids airborne from previous tanks, and absorbs particulate pollutants that aren't hydrophobic. It will also dissolve small amounts of gaseous contaminants, but probably not enough to be useful.
- Vegetable oil, which absorbs hydrophobic particulate pollutants. Any oil would work for this, but vegetable oil has the advantages that it's nontoxic, has very low vapor pressure at room temperature, and has somewhat higher surface tension than many alternative oils, reducing droplet formation. Used frying oil would likely work.
- Propylene glycol, which is nontoxic, has very low vapor pressure,

won't rot, and will absorb both hydrophilic and hydrophobic particulates, as well as vapors of many volatile organic compounds. It has relatively high surface tension (36 mN/m), reducing droplet formation. A lot of VOCs that won't dissolve significantly in vegetable oil are completely miscible with propylene glycol. It's also quite hygroscopic, so it serves to reduce humidity, but this could be a problem if it results in substantial dilution in normal use. At room temperature, propylene glycol reaches equilibrium with 60%-humidity air only once it's absorbed about 20% of its own weight in water!

• An aqueous solution of $CaCl_2$ or a similar salt will reduce humidity, potentially down to a very low level. Aside from directly controlling the humidity of the output air, this could be used to reduce the dilution by humidity of a later propylene-glycol or similar stage. Calcium chloride is also nontoxic. (An anhydrous calcium-chloride desiccator is actually what Dow's *A Guide to Glycols* recommends for drying air that will be in contact with propylene glycol if a nitrogen pad is not feasible.)

• An aqueous solution of a weak acid or weak base, such as sodium bicarbonate, could remove reactive gases such as SO_2 from the air. If you use calcium hydroxide (not weak!) or calcium carbonate for this, you produce gypsum as a bonus.

• Calcium hydroxide would also remove CO_2 from the air. If this is desirable, ethanolamine, diethanolamine or triethanolamine (in aqueous solution) may be a better choice, because it's substantially easier to "regenerate" by heating (to 120° in the case of ethanolamine) to drive out the carbon dioxide (somewhere outside). These unfortunately require at least 5 but ideally 200 atmospheres to absorb the carbon dioxide.

Regeneration

I mentioned above that the ethanolamines absorb a lot of carbon dioxide, which can be driven back out by heating, "regenerating" them. The topic of regeneration is interesting in general: rather than discarding the dirty filter liquid, you go through some kind of process to clean it, thus extending its life.

Particulates can be removed by filtration, but in some sense this is not a solution — you could have just filtered the air. More interesting is if you can remove them by centrifugation or flocculation.

Hygroscopic solutions such as propylene glycol and aqueous calcium chloride can also be regenerated by heating to drive off some of the water.

Pebble-bed alternatives

As mentioned earlier, Dow's recommendation for drying air that will be used to pad propylene glycol is to use an *anhydrous* calcium chloride desiccator. As I understand it, this is a pebble-bed kind of affair, with solid crystals of calcium chloride with air space between them.

Other kinds of pebble-bed-like things include the following:

• Platinum or palladium catalytic converters to remove organic compounds from the atmosphere, including even methane, as well as nitrogen oxides and ozone. Since a substantial part of the pollution

from both Otto and diesel engines consists of unburnt hydrocarbons, nitrogen oxides, and ozone, this could be very helpful in the city. These need to be hot to work, typically requiring a refractory substrate such as alumina, and they produce carbon dioxide, which may need to be managed.

• Activated carbon to adsorb many kinds of contaminant gases.

• Oxide or hydroxide of calcium, lithium, sodium, magnesium, or even potassium, to combine with carbon dioxide or (I assume) nitrogen oxides.

• Sodium bicarbonate is famous for adsorbing unpleasant odors, and would also eat acid gases like SO_2 or nitrogen oxides, though not carbon dioxide.

In general, pebble beds have the advantage over bubble tanks that they have no minimum pressure to operate. Also, as desiccators, they can reach lower humidities than aqueous solutions of salts can. They have the distinct disadvantage in this case of looking substantially less bitchin.

Also, pebble beds are less suited to continuous-flow processes. You can regenerate pebble beds in place by taking them out of service and passing a regenerant over them — typically hot air, but activated carbon needs a hot non-oxidizing gas instead, such as hot carbon dioxide. Steam is not suitable, as it degrades the carbon to produce highly toxic water gas. (It would be nice to have a non-oxidizing gas that isn't flammable or absurdly reactive, and is liquid or solid at room temperature, so that your activated-carbon regeneration gas doesn't pose a suffocation hazard. But nothing occurs to me at the moment.)

Plasma alternatives

If you want an air purifier that looks *really* cool, nothing can beat plasma, especially a reduced-pressure plasma with different alkali metals evaporating into it (from oxide or carbonate feedstocks applied to your electrodes, presumably). This will look especially cool if it uses high-frequency AC and it's inside a thin glass envelope so you can guide the plasma arcs with your fingers! But probably corona discharge in approximately atmospheric pressure is more practical.

Like a catalytic converter, this will also burn unburnt volatile hydrocarbons, and maybe also particulates, but the resulting gas is very far from breathable — it contains a substantial fraction of brown nitrogen oxide (NO_2), plus ozone and nitrous oxide (NO_2). The N_2O is reactive enough that you can combine it with just about anything (maybe bubbling it through sodium bicarbonate would be the easiest choice) but I'm not sure what to do about the ozone and nitrous other than using a catalytic converter from a car.

Oxidizing sodium, lithium, potassium, or even calcium or magnesium into the plasma, in addition to producing super awesome colors, might help to cut down on the nitrogen oxide production, too. But then you need to make sure you filter the generated metal oxides out of the air before you breathe it. Maybe some kind of hot acid refractory would work. Silica, for example, famously combines with sodium hydroxide to produce sodium silicate.

(Sodium and potassium nitrates are "saltpeter", a stable mineral that acts as the oxidant in gunpowder. Calcium nitrate, "norwegian saltpeter", also works for this. Magnesium nitrate is also stable. These are mostly used as fertilizer these days.

??? What are sodium, lithium, potassium, calcium, and magnesium nitrates like?

??? What are the other acid refractories?

Maybe the ozone could be made safe by passing the resulting gas over a "pebble bed" of something like used yerba mate or coffee grounds, thus converting it into relatively harmless carbon dioxide, and maybe a bit of water.

Topics

- Materials (p. 3560) (112 notes)
- Household management and home economics (p. 3504) (44 notes)
- Safety (p. 3693) (9 notes)
- Air quality (p. 3308) (6 notes)
- Phyics

Argentine oscilloscope pricing 2016

Kragen Javier Sitaker, 2016-08-16 (4 minutes)

http://articulo.mercadolibre.com.ar/MLA-602979566-protec-osciloscopio-20-mhz-modelo-p-3502-c-_JM is a more or less typical cheap used analog oscilloscope: the 20MHz Protek P-3502C. The price is AR$3500 = US$232.

http://www.tek.com/oscilloscope/tbs1000b-digital-storage-oscilloscope Tektronix's cheapest current scope is this TBS1000B line; the lowest-end scope in the line is the TBS1032B, with 500Msps and 30MHz analog bandwidth, for US$450, with a recording length of 2500 points.

http://articulo.mercadolibre.com.ar/MLA-620039875-osciloscopio-hantek-6022-be-20-mhz-_JM is a typical USB oscilloscope: the Hantek HT6022BE20MHz: 20MHz, AR$5000 (=US$331), 1MΩ 25pF input impedance, 48Msps, 8-bit resolution, 20mV to 5V gain range.

https://www.seeedstudio.com/item_detail.html?p_id=736 is the Seeed Studio DSO Quad 4-channel Digital Storage Oscilloscope, which costs US$169. It claims 72 Msps but doesn't make any claims about analog input bandwidth on its two analog input channels. IIRC Seeed is an open-hardware shop; they do publish DSO Quad schematics and the Wiki lists user apps and it seems like they're using an AD9288BSTZ-40 for their ADC.

It seems like you ought to be able to meet or exceed the capabilities of the Hantek unit for a much lower price, especially using recycled chips. I mean basically this is two high-speed ADCs (or one with a demultiplexed input) hooked up to a USB interface in a metal box, right?

Three candidate ADCs are the US$3.69 TI ADS830E/2K5, the US$4.50 Maxim MAX19505ETM+T, and the US$3.28 Analog Devices AD9283BRSZ-50, which last is available in quantity 1 from Digi-Key at US$6.01.

The AD9283 family is 90mW, has 475 MHz analog bandwidth, a 46.5 SNR, a 1V p-p analog input range, and runs off 3 volts. "Low-cost digital oscilloscopes" are explicitly called out as a use for the thing in its datasheet; Matthew Lai designed such a scope based on a slightly higher-speed member of the family. It claims ENOB of 7.5 bits at 27 MHz input, which is better than the Hantek unit.

Its input capacitance is only 2 pF. It says its input resistance is only 7 to 13 kΩ, which seem to be pullup and pulldown resistors to the power rails. So you probably need some kind of preamp, like maybe an opamp or something, to get high input impedance.

Lai's design uses an FPGA to buffer the digitized signal. The Seeedstudio design uses both an FPGA and also an ARM Cortex microcontroller.

Lai's design's analog front end was as follows:

Input goes into a 1MΩ metal film resistor in parallel with 20pF ceramic cap (not sure about this). Buffered by a Texas Instruments OPA656 wideband op amp with JFET input (very high impedance, which we need, because the probe has high input impedance. Downside? bloody expensive).

There are a variety of OPA656 parts; Digi-Key has the OPA656U at US$10.69, but I think he may have used a slightly more expensive one, but maybe the price has come down since 2010. The OPA656 family seems like it might be somewhat overkill for this, with a 230MHz bandwidth and gain-bandwidth product, 70 mA output current (!!), and 65dB open-loop voltage gain.

Lai's total BOM cost is US$107.58; a quarter of this is these op-amps, and another quarter is the THS7002 preamp.

Topics

- Electronics (p. 3430) (138 notes)
- Pricing (p. 3646) (89 notes)
- Oscilloscopes (p. 3614) (12 notes)

The Gelfand Principle, or how to choose educational examples

Kragen Javier Sitaker, 2007 to 2009 (8 minutes)

I was reading Doron Zeilberger's Opinion 65 about the Gelfand Principle, which he ascribes to Israel Gelfand. I'll restate what he says here.

He says it is better to begin by explaining that 1+2 = 2+1, and then go on to explain that this is true for all pairs a+b = b+a and that this is called the commutative property of addition, than to begin by giving the name, explaining that it means that always a+b = b+a, and then by giving an example. And he points out that 1+2 = 2+1 is the best example to use, because there's nothing particularly special about it. 0+1 = 1+0 is true, but in general 0+x = x+0 = x, and so someone might think that this commutative property is a special feature of 0. (In linear algebra, the identity matrix works this way: for all conformable M and identity matrices I, IM = MI, even though matrix multiplication is not commutative in general.) And 1+1 = 1+1, but that's because of the reflexive property of equality, not the commutative property of addition. So 1+2 = 2+1 is the simplest nontrivial example. It's a better example than, say, 424 + 501 = 501 + 424, because it's easier to prove: 1+2 is 1+(1+1), and 2+1 is (1+1)+1, and addition is associative, so those are the same.

In general, he argues, "Whenever you state a new concept, definition, or theorem, (and better still, right before you do) [you should] give the *simplest* possible non-trivial example." He also says:

> The Gelfand Principle should also be used in research articles. It is much easier to follow a new definition or theorem after a simple example is first given. Even proofs would be easier to follow if they are first spelled out concretely for a special case.

I agree. In fact, I've often groused to myself about mathematical papers being written in the opposite style. (I've spent some time lately reading John Backus's "Can Programming Be Liberated from the von Neumann Style?", which is not technically a math paper but contains theorems anyway, and it would be considerably improved by an application of this principle.) I've wondered whether other people --- certain mathematicians maybe --- actually find it easier to understand things in what seems to me like a backwards order: theorem first, then example.

(Amusingly, Zeilberger states the principle before he gives the above example of it, thus violating the principle he is trying to promote; however, he follows it in part, in that the commutative principle is probably the simplest possible nontrivial example of stating a theorem.)

Apparently, however, other people also feel that the Gelfand approach is the correct one. In response to Zeilberger's article, Tim Gowers wrote:

> I've just looked at your opinions page for the first time for a while, and read your article on two pedagogical principles. I was particularly interested in the first [the Gelfand Principle], because as a result of editing the Princeton Companion I have become incredibly conscious of it myself -- I'm tempted to say that I discovered it independently. Of course, it doesn't bother me that Gelfand got there first -- it is

SO clearly correct that it would be a miracle if I had not been anticipated. Instead, we have the depressing miracle that something so obvious should be practised by such a small percentage of mathematicians. I feel quite evangelistic about this, and have already started a one-man (except that now I see that you are an ally) campaign to publicize the principle. For example, a few weeks ago I was asked to give a talk about the Princeton Companion, and EXAMPLES FIRST was one of the main themes (which I illustrated by an example first: I gave a ridiculous and unmemorable definition of a "C-space" which was in fact a mathematical model of a car, and as soon as the word "car" was uttered, the definition was magically easier to remember).

I had always been aware, of course, of the value of giving the simplest non-trivial example. The thing that has really struck me is the value of giving it FIRST. I think it is very important to stress that this is an independently important part of the Gelfand principle (or else, if you were not including it, a separate and equally important principle).

Here is my "proof" that it is better to start with concrete examples and proceed to abstract definitions than it is to begin with the abstract definitions. If you give the example first, then it is easy for the reader to understand, so not much effort is needed to remember anything. Then, when you are presented with the abstract definition, you have a mental picture of an example, so the various components of the abstract definition become labels that you attach to this picture. If, on the other hand, you give the abstract definition first, then the components are meaningless, so you have no choice but to memorize them as if you were learning Chinese vocabulary or something. Then when you see the example, you have to go back and see how this meaningless stuff does in fact mean something. But that effort of memorization should have been unnecessary!

Dijkstra, unsurprisingly, disagrees (in EWD757-3):

There exists a school (I wouldn't call it a "school of thought") that believes in "teaching by example" and in "discovery by example". I don't. I concluded EWD376, which describes in detail the actual steps in which I had solved a problem from graph theory, with:

"Finally we draw attention to the fact that we did not draw a single example to explain what we were talking about or (even worse!) to discover what the program should do. And this, of course, is as it should be."

So what's the analogue in computer programming? You could argue that it's test-first programming, where you write the unit test before you write the code, and hopefully people will read it in the same sequence. The unit test is usually a better unit test if it's exactly this simplest non-trivial case. (Maybe it's better if there's an additional test that doesn't try to be simple, just in case you were mistaken about how trivial the case was.)

But in the mathematical case, you don't just write down the premises of the example, write down the conclusion, baldly assert that some theorem exists to connect the two, and then proceed to explaining what that theorem is and proving it in the general case. Instead, you demonstrate that the conclusion is true of that particular example, and then state the theorem and proof for the general case. This is much more similar to walking through the program as it executes the test case in a debugger and looking at all the intermediate values.

There was a thread on LtU about "Ivory Towers and Gelfand's Principle", on motivating language features with examples:

If an example has a solution that is nearly as good without a given language feature, then that example is not a good motivation for that feature. Perhaps not following this principle is partly what earned FP it's ivory tower reputation.

There was also a thread about this on LiveJournal.

Topics

Some notes on reverse-engineering The Wizard's Castle

Kragen Javier Sitaker, 2018-04-26 (9 minutes)

I remember when I was a kid I used to love playing Joe Power's "Wizard's Castle" game, which is about 900 lines and, as I learned today, originally ran on the 16-kilobyte model of the Exidy Sorcerer home computer at the New Directions in Computing computer store, after which the Kingdom of N'Dic is named.

When I was a kid I tried to understand the source code of the game, since after all it is only 900 lines, written in a programming language I was pretty familiar with, and performed a function I was pretty familiar with, having spent hundreds, if not thousands, of hours playing the game. Unfortunately, I didn't know what I was looking for.

I just downloaded the IPCO version of the the game from myabandonware.com and find that it's still somewhat hard to understand the source. Unfortunately this version is broken, or rather cheated — the map is revealed starting from the beginning of the game. So I needed to fix the bug, which turned out to be that line 4150 said:

```
4150 IF Q > 99 THEN Q=Q-100 ' LET Q=34 TO HIDE ROOMS
```

when it should say:

```
4150 IF Q > 99 THEN Q=34 ' LET Q=34 TO HIDE ROOMS
```

It also has a bug on line 3590, which says:

```
3590 PRINT CHR$(27);"E"
```

This is the clear-screen sequence for the Heath H89 and H19, but this version of the program is for GW-BASIC or BASICA; this line should just say

```
3590 CLS
```

Here is a sort of dictionary of the subroutines, goto targets, and variables in the program.

A$: a temporary input variable.

Q: a temporary variable used in many places for many things.

R$: an array of the 4 player race names. RC: the index of the player's race.

SAMP$: a parameter indicating whether the program was invoked from IPCO's SAMPLES.BAS program; if "YES" (due to being invoked as CHAIN "WIZARD", 1010) then, upon exit, it reinvokes SAMPLES.BAS rather than exiting to the BASIC prompt.

1000: program entry point

RF: a boolean indicating whether you have the Runestaff OF: a boolean indicating whether you have the Orb of Zot

L: an array of 512 ints, representing the castle contents. For

whatever reason (perhaps resulting from the conversion from the Sorceror version which stored this array in character RAM), this is a one-dimensional array with indices calculated by FND(Z) rather than a three-dimensional array. L entries have 100 added to them to mark them as "unknown". Valid numbers range from 1–33 and 101–133, with 34 as a sort of special case representation of the whole 101–133 range for the map.

The valid values are:

- . AN EMPTY ROOM
- E THE ENTRANCE
- U STAIRS GOING UP
- D STAIRS GOING DOWN
- P A POOL
- C A CHEST
- G GOLD PIECES
- F FLARES
- W A WARP
- S A SINKHOLE
- O A CRYSTAL ORB
- B A BOOK
- M A KOBOLD
- M AN ORC
- M A WOLF
- M A GOBLIN
- M AN OGRE
- M A TROLL
- M A BEAR
- M A MINOTAUR
- M A GARGOYLE
- M A CHIMERA
- M A BALROG
- M A DRAGON
- V A VENDOR
- T THE RUBY RED
- T THE NORN STONE
- T THE PALE PEARL
- T THE OPAL EYE
- T THE GREEN GEM
- T THE BLUE FLAME
- T THE PALANTIR
- T THE SILMARIL

These are used to index into C$() and I$() mentioned below.

The two other special unique items (the Runestaff and the Orb of Zot) do not have their own codes; the Runestaff is in the possession of a monster (and is thus in the same room with it) while the Orb of Zot is "disguised as" a warp, and in fact behaves as one.

C$(): an array of the 34 strings describing different room contents; the 34th is the dummy entry "X".

I$(): an array of the 34 characters to display different room contents on the map.

9590–9620: a subroutine to pick a randomly chosen empty room and set its contents to Q, returning its results in X, Y, and Z.

C: a 3×4 array of ints. C(1,4), C(2,4), and C(3,4) are used to

represent whether the player is afflicted with particular curses, I think — a leech that drains your gold, lethargy that makes you move more slowly, and forgetfulness, which slowly erases your map. These curses are found in particular empty rooms. This logic is in lines 2940–3050.

Lethargy apparently just makes you eat more often, but not eating doesn't harm you. This has the feeling of an unfinished feature. However, lethargy also gives monsters the initiative in combat.

H: the last time you ate. T: The current time.

W$(): an array of weapon and armor names; NO WEAPON is index 1, while NO ARMOR is index 5.

E$(): an array of 8 dishes one can cook from monsters.

WV: one less than the index of your current weapon in W$(), or 0 if you are empty-handed.

AV: 5 less than the index of our current armor in W$(), or 0 if you are empty-handed. AH: your armor's health or armor's hit points; when this reaches 0, the armor is destroyed.

LF: 1 if the player has a lamp, 0 otherwise. FL: the number of flares the player has. BL: 1 if the player is blind, 0 otherwise. BF: 1 if the player has a book stuck to their hands, 0 otherwise.

T(): an array of 8 ints which are 1 if the player possesses the corresponding treasure or 0 otherwise. These are in the same order as the numbers for the room contents; for valid indices, T(Q) represents the possession of C$(Q+25).

O(): an array of 3 ints containing the coordinates of the Orb of Zot
R(): an array of 3 ints containing the coordinates of the Runestaff

O$: the returned user input from the subroutine at 9830 or other input subroutines

FNA(Q): returns a random number in [1, Q]

FNB(Q): returns Q wrapped to the range [1, 8] (i.e. (Q-1) % 8 + 1), which is what you want for wrapping coordinates to the 3×3×3 hypertorus structure of Zot's castle

FNC(Q): returns Q if Q<19, otherwise 18; used for saturating arithmetic on character attribute values (strength ST, dexterity DX, intelligence IQ), which cannot increase past 18.

FND(Z): returns the index into L at which to find the contents of castle room (X, Y, Z).

FNE(Q): returns, roughly, Q % 100 — used to compute the true contents of a possibly unmapped room.

X: an X-coordinate in the castle, normally that of the player (but also an implicit argument to FND); saved in A when a temporary value is needed

Y: a Y-coordinate in the castle, normally that of the player (but also an implicit argument to FND); saved in B when a temporary value is needed

Z: a Z-coordinate in the castle, normally that of the player; saved in C when a temporary value is needed

A, B, C: used to save the player's X, Y, and Z at times; also used in one place temporarily for the coordinates of the Orb of Zot. Also, during combat, A is used to store the index of the monster type (12 less than its index into C$().)

VF: 1 if vendors are angry with you, 0 otherwise

Y$: a prompt string

NG: the number of games played so far, used to avoid

reintroducing the game

1–1200: initialization code for the whole program

1240–1390: initialization code for a new game ("restocking the castle")

9770–9820: subroutine to print a line of asterisks

9830–9870: subroutine to request a generic one-letter choice from the user in O$, with two entry points — the entry point at 9830 prints a blank line and a prompt, while the entry point at 9850 does not.

2920–8410: the main game turn loop.

Topics

My attempt to learn about jellybean electronic components

Kragen Javier Sitaker, 2017-02-08 (updated 2019-09-29) (22 minutes)

What parts are "jellybeans", and what are they like?

FETs

Normal FETs are N-channel enhancement-mode MOSFETs; JFETs, depletion-mode MOSFETs, and to some extent P-channel FETs are weirder.

G.M. Electronica's power MOSFET list

Here's a table of 16 popular FETs I found, with prices in quantity 1 from Digi-Key.

PN	Vds	A	ohms	Qg (nC)	¢	W	type
2N7000	60	.2	1.9	2	36	.4	
2N7002	60	.115	7	2	38		
IRF630	200	9	.4	45	86	75	
IRF9630	200	6.5	.7	29	151	74	P-chan
IRLI630G	200	6.2	.400	40	229	35	
IRLML6344	30	5	.029	6.8	36	1.3	
IRLML6402	20	3.7	.065	12	40	1.3	P-chan
EPC2036	100	1	.065	.910	97		GaN
SI3483CDV	30	8	.034	11.5	89	4.2	P-chan
FQP27P06	60	27	.070	43	134	120	P-chan
NTD4906N	30	54	.0055	24		2.6	obsolete
IRF7307	20	4.3	.140		83		dual (P&N)
BSS138	50	.200	3.5		24		
CPC3703CTR					70		depletion
2N5457	25	.01			230		JFET
2N5458	25	.01			230		JFET
SiS410DN	20	35	.0048	41	94	52	
PSMN4R0-40YS	40	100	.0056	38	88	106	holy shit
IRF540N	100	33	.044	71	145	130	fuck
IRF9540N	100	23	.117	110	189	110	P-chan
IRF9530	100	12	.300	38	138	88	P-chan SyC

The IRF630 N-channel FET https://www.digikey.com/product-detail/en/stmicroelectronics/IRF630/497-2757-5-ND/603782 is a monster. It's Digi-Key's most popular IRF* part (twenty thousand in stock at 86¢ quantity 1, 39¢ in quantity 1000) and can switch 9 amps (with pulses up to 36 amps) at up to 200 volts in 20 nanoseconds. Even G.M. Electronica, which generally only carries obsolete parts, lists an IRF630 in their catalog. And the TO-220 package can dissipate 75 watts. It includes an antiparallel zener capable of about ten amps for overvoltage and flyback protection. Its main drawbacks for Arduinoish use seem to be that it doesn't turn fully on until about 6 to 10 volts on the gate, its on-resistance is a high-for-a-power-FET 0.4Ω, and its max gate-source voltage is only 20 volts; but even at 4 volts, it's ohmic

with a reasonably low resistance up past two amps, maxing out at about three. Five volts gets you up to 11 amps. SyC Electrónica has it for US$0.75. It has a P-channel counterpart, the IRF9630, of which Digi-Key only has 2000 in stock; it's slightly worse at 6.5A and 0.8Ω.

The logic-level counterparts to International Rectifier's IRF *parts are the IRL* parts. One popular one is the IRLML6344 http://www.digikey.com/product-detail/en/infineon-technologies/IRLML6344TRPBF/IRLML6344TRPBFDKR-ND/2538162 (36¢ for one, down to 11.3¢ each) which switches 5A, 30V, at only 29mΩ, and turns on at 2.5V on the gate. Digi-Key has 650,000 of them in stock. A similar, slightly crappier, P-channel device is the IRLML6402 http://www.digikey.com/product-detail/en/infineon-technologies/IRLML6402TRPBF/IRLML6402PBFCT-ND/812500. The IRF630 itself has an IRL version, the IRLI630G, with the same 200V but a slightly lower current of 6.2 amps and like 70 ns of delay, but this is apparently a rare bird — Digi-Key charges US$2.29 for one.

The 2N7002 is Digi-Key's absolute most popular FET, with 1.6 million in stock; it's another N-channel FET that even G.M. Electronica carries; Digi-Key has 184 models, the most popular being

https://www.digikey.com/product-detail/en/diodes-incorporated/2N7002T-7-F/2N7002T-FDIDKR-ND/1837287, which is 38¢ for one and 8¢ each for 1000. This is a smaller-signal part — although it can handle 60V, it can only do up to 115mA and 300mW, and its on-resistance is multiple ohms (like 7Ω). But it switches at 2.5V, also in the same 20ns as the IRF630. At 5V gate voltage it reaches 500 mA, which (because of the high resistance) will burn it up if you let it continue. The ON Semiconductor version http://www.digikey.com/product-detail/en/on-semiconductor/2N7002LT1G/2N7002LT1GOSCT-ND/917791 is cheaper, as low as 2.958¢ in quantity 1000.

SyC Electrónica doesn't have the 2N7002 (though G. M. Electrónica does), but they do have the similar 2N7000 for US$0.12 http://www.sycelectronica.com.ar/articulo.php?codigo=2N7000. 60V, 200mA, 1.9Ω, switches on at 3V Vgs. eLemon carries it too at almost the same price http://www.elemon.net/BuscarSubRubros.aspx?Action=2&GrupoId=TR&RubroId=1710&SubRubroId=4. Even Tom Jennings recommends it!

The 2N7000/2N7002 datasheet from Fairchild doesn't give its Qg directly, but it seems to be designed for a 10V Vgs and has typically 20pF and up to 50pF of input capacitance, which I guess would mean 500pC? But ST's datasheet for their obsolete 2N7000 gives that for Qgd, plus another 800pC of Qgs, for a total of 1.4 to 2 nC.

The most popular non-2N7002 FET at Digi-Key is the EPC2036, which is a bumped gallium-nitride die (900µm square) with no packaging (to avoid bond-wire inductance!) https://www.digikey.com/product-detail/en/epc/EPC2036/917-1100-6-ND/5224979 for 97¢, down to 39¢. Needless to say, G.M. Electronica has never heard of it. It switches 1.7A at up to 100V with a resistance of only 65mΩ, which is supposedly about a 30 times higher breakdown voltage than can be achieved at that resistance with silicon, with a gate charge of only 910 pC, about an order of magnitude lower than MOSFETs, on a gate capacitance of about 80

pF. As a result, they can switch at over 10MHz. Apparently GaN power transistors like this debuted in 2010, which is why I hadn't heard of them until now.

Digi-Key's most popular P-channel MOSFET is Vishay's SI3483CDV, with six hundred thousand in stock at 89¢, down to 41¢. It switches up to 8 amps at up to 30 volts, and can handle gate-source voltages of up to 20 volts. Being a P-channel device, the drain and source are backward from the more common N-channel type: the source is positive, drain negative, and the gate voltage is measured below the source.

Other candidate P-channel MOSFETs people on #electronics recommend for default use include the IRF4905 (94¢, -55V Vds, 0.02Ω, 74A, 200W, 2-4 V threshold voltage, fully on at 10 volts, 180 nC gate charge, 18 ns turn-on delay time, 99 ns rise time, 61 ns turn-off delay time, 96 ns fall time).

In P-channel MOSFETs, SparkFun recommends the FQP27P06 https://www.sparkfun.com/products/10349 for 95¢; Digi-Key has them https://www.digikey.com/products/en?keywords=fqp27p06 for $1.34 down to 61¢. They switch up to 27 amps at up to 60 volts.

Tom Jennings recommends the NTD4906N. G. M. Electronica doesn't carry it, and Digi-Key marks it as obsolete. I don't understand its datasheet, which has wildly different current and power numbers. I suspect I didn't understand any of them.

For building an H-bridge, it's often useful to use N-channel and a P-channel MOSFETs together, and sometimes they come in a package for this purpose; the IRF7307 is a popular such device at Digi-Key https://www.digikey.com/product-detail/en/infineon-technologies/oo IRF7307TRPBF/IRF7307PBFDKR-ND/1648232 for 83¢ down to 37½¢, with twenty-two thousand in stock. G. M. Electronica also carries it. It can switch 4.3 amps at 20 volts in 100ns; its resistance is 50 mΩ on the N-channel side and 90 mΩ on the P-channel side. This is convenient, but in the standard design, you kind of have to use the same supply for the gate as for the load.

For a MOSFET with lower switching voltage, people seem to suggest the BSS138. http://www.digikey.com/product-detail/en/on-semiconductor/BSSoo 138LT1G/BSS138LT1GOSDKR-ND/1831753 lists it at 24¢ down to 4.6¢, switching 50V 200mA, with a high on-resistance of 3.5Ω. Sure enough, though, it passes almost 400mA at a 2.5-volt gate voltage.

Depletion-mode MOSFETs are apparently very exotic; Digi-Key's most popular one is the high-voltage 70¢ CPC3703CTR, of which it has almost forty-eight thousand in stock.

Of all of these, the only ones I've seen mentioned in Horowitz & Hill are the 2N7000, the 2N7002, and the BSS138. Neither GM nor SyC has the BSS138. Horowitz & Hill also tout LND150, DN3435, BS170, MMBF170, 2N5457, 2N5458, 2N5459, and BS250 parts as popular.

The 2N5457 and 2N5458 are available at SyC, though not GM. Digi-Key has them for US$2.30 (!!) down to US$1.01 but they sound inferior to others mentioned above (25V, 10mA). They're JFETs, though, so not directly comparable to the MOSFETs above.

All of the above are good only up to less than ten amps. The SiS410DN, by contrast, handles up to 35A; it's a 20V N-channel

MOSFET in a somewhat unusual package with almost 215,000 available at Digi-Key for 94¢ down to 43¢. Even more impressive, Philips/NXP/Nexperia's PSMN4R0-40YS, which came out in 2010, does 100A at 40V, up to 106W, costs 88¢ down to 51¢, and has 77,000 available at Digi-Key. Here in Argentina, apparently the only power MOSFETs people sell are higher-voltage International Rectifier (Infineon) HexFET parts like the IRF540N — at Digi-Key, 10,000 in stock, 145¢ down to 66¢, 33A at 100V, nice low 4V threshold voltage, and available from both G.M. and SyC. SyC sells it for 60¢. The P-channel counterpart is the IRF9540N. All SyC has in stock is the IRF9530, which is rated for only 12A.

Other drivers

Amusingly, the TIP120 bipolar NPN Darlington Jennings recommends replacing with an unavailable MOSFET is still "Active" https://www.digikey.com/product-detail/en/fairchild-on-semiconductor/TIP120/TIP120-ND/1052441. It switches 60V at 5A and costs 61¢ down to 26¢, but as Jennings points out, its 1000:1 or 2500:1 current gain is vastly inferior to power MOSFETs'; its voltage drop is many times higher, especially at low current; and its switching time is vastly longer at around 1000 ns.

The ULN2003 costs US$0.40 at SyC: http://www.sycelectronica.com.ar/articulo.php?codigo=ULN2003 It's an array of seven 500mA 50V 250ns current-sink Darlingtons with built-in flyback diodes. It's like a seven-pack of somewhat lower-powered TIP120s.

There's an 8-transistor version, the ULN2803, and (as I found in Microlens vibrating lightfield (p. 1219)) there used to be a high-side switching version UDN2891, but it's obsolete now. One alternative is the TI TLC59123 (US$1.81), which is an 8-output latching high-side driver, which has the advantage that it ignores its inputs except at clock edges. It's only 13.2V, unlike the 60V ULN2003, but it's also 500mA per channel, and it can supposedly be clocked at "up to 1 MHz" with clock pulses of 100ns and 100–200 ns propagation delays.

Bipolar transistors

Aside from the TIP120 and ULN2003 mentioned above, we have transistors. For example, https://www.digikey.com/product-detail/en/diodes-incorporated/MMBT3904-7-F/MMBT3904-FDICT-ND/815727, the MMBT3904, for 12¢ down to 2½¢, a SOT-23 NPN small-signal transistor. That's the SOT-23 version of the 2N3904, thus the weird prefix.

Thyristors

SCRs, triacs, and occasionally diacs are the workhorses of powerline switching. They typically turn on in 100 ns. Unlike MOSFETs, they fail open rather than closed (I think), but they have the disadvantage that you can't turn them off until the next zero-crossing of the waveform. Also, unlike MOSFETs and unlike relays (the other alternative for powerline switching), they have substantial voltage drops and so dissipate substantial power from the load current.

You might think that this would entirely prevent you from using them for switching DC, but the standard trick to solve this problem is to put an LC resonator either in series or in parallel with the thing so that the current drops to zero at some point, turning the thyristor off. This is called a "self-commutation circuit".

Triacs are the usual thing to use for powerline switching, since they have a gate electrode like a regular transistor, but can handle power in either direction like a diac. A typical triac might be the 95¢ STMicroelectronics T405Q-600B-TR ("Applications: Mahjong machines, lighting dimmers"), which can block up to 600 volts; once you trigger it with 1.3V and 5 mA on the gate electrode, it can carry up to 4 amps (dropping 1.5 V or less), and will continue to do so until the load current drops below 10 mA.

Diacs have no trigger electrode; they're triggered just by reaching the breakover voltage, like the no-longer-produced Shockley diode. A diac is like two Shockley diodes in antiparallel. A typical diac is the 49¢ Micro Commercial LLDB3-TP "silicon bidirectional diac", which breaks over at 28–36 volts in 2 µs and can handle two amps. While it's turned on, it drops 5 volts. There are higher-voltage diacs, as you would expect, but they're all fairly low power.

SCRs, the classic thyristors, are unidirectional, like Shockley diodes, but have a gate electrode, like a regular transistor or a triac. A typical SCR is the Littelfuse SK055NRP, which costs US$4.81 and can handle 55 amps RMS when you turn it on (or surges of 650 amps), dropping 1.8 volts, and can block up to a kilovolt when off, leaking 30 µA. You turn it on by feeding 40 mA into its gate at 1.5 volts, and it turns off when the current drops to 60 mA. I'm not sure how much reverse bias it can handle.

Op-amps

The LM324, from 1975. Available (LM324D, LM324N) at G.M. and (LM324, LM324-SMD) at SyC, mentioned in Horowitz & Hill, etc., 35000 available at Digi-Key for 48¢ down to 12¢. A quad bipolar op-amp that runs on 3V to 32V good to gain-bandwidth of 1.2MHz, open-loop amplification of 100k×, not quite rail-to-rail, 10 mA output.

Horowitz & Hill recommends the TLC272 as a popular MOSFET-input alternative to the LM324 and the LF411 and LF412 as JFET-input alternatives. The LF411 (available from GM) touts being "pin-compatible with the standard LM741", has 3 MHz of gain-bandwidth, sucks only 50pA from its input, costs US$1.19 down to 54¢ (but only has one op-amp on the chip), runs on up to 18 volts, 200k× open-loop amplification, ±12V output, like 20 mA output current. Digi-Key has 3000 available. The TLC272 (available from GM) spews out 30mA at up to 16 V and 2.2MHz of gain-bandwidth, sips an incredible 0.7pA of input current, and has two op-amps on a US$1.34 (down to 60.9¢) chip.

The LM741 itself (from 1968!) has stock of 68000, costs 66¢ down to 28¢, does 1.5MHz GBP at 80nA of input, 25mA of output, and up to 36V. I wasn't quite sure what its major advantages over the LM324 are, but it seems to cost about five times as much. Fortunately, both chips are popular enough that people have discussed this very question previously; they claim the LM324 is noisy and has crosstalk, but runs off a single supply instead of needing positive and

negative supply voltages.

Digi-Key's most popular item in the op-amp category is the tiny 70¢–53¢ MCP6031T-E/OT, which is rail-to-rail and can spew out 23mA while taking in 1pA, but only up to 10kHz and 5.5V. It's touted as having super low consumption. They suggest using it for battery current monitoring and "sensor conditioning", whatever that is. They have almost 165000 in stock.

In non-super-low-frequency opamps, Digi-Key's most popular item is the TSV321IDT, with 142000 in stock, but only available in quantity. The similar TSV324IPT, by contrast, is available in quantity 1 for 58¢, down to 24.9¢, with 162000 in stock. It's a quad rail-to-rail 1.4MHz GBP op-amp chip with a 70nA bipolar-class input current, 80mA output current, and up to 6 V.

If we demand another order of magnitude in gain-bandwidth product, we exclude 85% of the op-amps on Digi-Key and the price goes up a little; the LM7321MF/NOPB from 2008 is US$1.83 down to 54¢, has 45000 in stock, and is a rail-to-rail 20MHz op-amp with 1100 nA input bias and 100mA output at up to 32V.

An additional order of magnitude excludes half of the remainder, and we end up with things like the OPA356AIDBVR, for US$1.91 down to 94¢, a 200MHz GBW (450MHz unity-gain!) voltage feedback amplifier (?) with rail-to-rail output, 3pA input bias current, 100mA output at 5V, and 80 dB (!) open-loop gain, 11500 in stock. This seems to be where modern op-amps are normally.

None of the popular Digi-Key opamps seem to be available locally.

Power management

The LM317 is a 100mA 71¢ linear regulator adjustable over 1.2 to 32 volts output and accepting 3.7 to 38 volts input. It's 30¢ at SyC.

The 7805 is a 1A 62¢ linear regulator fixed at 5 volts output. There are other versions with other voltages, like the 7812, and the 79xx series is the negative-voltage counterpart. SyC has the 7805 for 35¢, and hasthe whole family, both negative and positive.

Viper-7 recommends instead the HT7333 and similar for low-current applications: "massively lower dropout voltage (around 1/10th) than 78xx regs, very low quiescent current (4uA), decent tempco (about 1mV per 2°C) and 0.2% line regulation."

Zeners

Digi-Key's most popular zener is the obsolete 50¢ MAZ30430ML, with 429000 in stock: 4.3V±7%, 200mW. Its cheapest available non-obsolete zener is the 5.1V 500mW 1N5231B-ND, with 16000 in stock, 11¢ down to 2.3¢; this is the 5.1V member of the 1N5221 series Horowitz & Hill use as their canonical low-voltage zeners.

Horowitz & Hill also suggest using the LM385 (1.23V, 63¢, 57000 in stock at Digi-Key, 10µA–20mA) instead of a low-voltage zener. Shirriff says the TL431 from 1978, an adjustable substitute for the LM385, is more common (42¢, 29000 in stock at Digi-Key, 2.5–20V, 600µA–100mA).

Viper-7's "Beginner's Shopping List"

Viper-7 from Freenode ##electronics wrote a page about which

3500 electronic components he thinks a beginner should buy for US$110, with links to AliExpress vendors. It doesn't go into a lot of detail in some cases, and I don't want to just copy the list here, but I'll point out a few notable things.

1500 of the 3500 parts are 1% resistors; historically 1% resistors were expensive, but they no longer are, so you might as well use them. However, he only recommends getting 30 different values to cover six orders of magnitude, which is five values per order of magnitude — presumably the standard 1.0, 2.2, 3.3, 4.7, and 6.8. This means you can hit any value to within ±10% with a single resistor, but if you need a more precise value you need to build a network.

In terms of capacitors (420 of them) he only recommends electrolytic and ceramic capacitors, and no large-value MLCCs — the ceramics only go up to 100 nF.

(He also recommends some inductors, of course, and 250 LEDs.)

He recommends relatively few discrete semiconductors, and unfortunately in some cases doesn't say which, but does specifically mention the IRLZ44N and IRF4905 (10 each).

In power supplies, he recommends getting 200 (unspecified) fixed linear regulators, plus ten LM317s; the only voltage reference he recommends is a bag of 140 zeners.

The only connectors he recommends are breadboards, jumper wires, breadboardable through-hole screw terminals, and a MicroSD adapter.

There's a section of recommended boards, including level shifters, Arduinos, serial adaptors, a NodeMCU, and an assortment of Arduino-targeted sensor boards.

The most interesting part is the "other ICs" section: ten 555s, five MCP23017 "16-bit I/O expanders", two TLC5940 16-bit 120mA PWM drivers, five LM348A clones of the 741 quad op-amp, ten LM324 quad op-amps, ten TL084 JFET-input op-amps, and ten LM339 comparators. (This is not counting the microcontroller boards he also recommends.)

Viper-7's a smart person with a fair bit of experience, so it's really interesting to see what they chose to include — lots of op-amps and comparators, a few GPIO kinds of chips, and some 555s — and what he didn't — switching regulators, voltage references, ADCs and DACs, class-D amplifiers, supervisory power chips, transistor arrays like the ULN2003, MOSFET gate driver chips, H-bridges, microcontrollers that aren't already on a board, adjustable LDO regulators, memory, CMOS 555s, SSI or programmable logic.

More generally, he also doesn't recommend a wide variety of discrete parts: no power resistors, resistors below 10 Ω, crystals, LED drivers (even WS2812s), film capacitors, tantalum capacitors, supercaps, relays, batteries, transformers (!), motors, or JFETs; no IGBTs, solid-state relays, or thyristors; no lasers, switches, speakers, microphones, or displays; no protoboards, solder, or solder braid; and not even any 2N7000s, though he says they're often included in BJT assortments, even though they aren't BJTs. (I want to be clear that I'm not disagreeing with their choices — on the contrary, I'm saying that if you disagree with them you should have a reason.)

All in all, a really interesting set of recommendations, one which I think will probably help a lot to make electronics as a hobby more accessible.

Topics

How can we take advantage of 16:9 screens for programming?

Kragen Javier Sitaker, 2012-12-17 (2 minutes)

Modern laptop screens are 16:9 rather than the traditional 4:3. They're also much higher resolution. Could you take advantage of this for programming? It's helpful to see a lot of text on your screen at once for programming, but we're somewhat inflexible about the shape of the text, since it's not easy to reformat program source code for wider and narrower windows.

The traditional terminal format was 80x24 or 80x25, the former of which is also the default size for xterm and gnome-terminal; each glyph contained roughly 5x8 pixels which were almost square. There's a "5x8" font that ships with X11 that shows what this looked like; try `xterm -fn 5x8`. There's also a 5x7 font, which is just about as readable, and there are somewhat less readable 4x6 fonts floating around. The appearance on an LCD is a little blockier than it was on the hardware terminals, because their pixels were not neat little squares like LCD pixels, but rather fuzzy dots or horizontal segments of scan lines.

However, modern LCD displays actually have not only grayscale but also three times the horizontal resolution of a CRT with the same nominal number of pixels. This extra horizontal resolution can be, and is, used to dramatically the readability of text, which correspondingly allows the use of smaller fonts.

So let's suppose, conservatively, that we can use 3x6 fonts, which are really 9x6 --- 54 subpixels per glyph rather than the traditional 32, which ought to be eminently readable. How much text can you fit on the screen?

Suppose you break up the screen horizontally into 80-column columns. Each of these will be 240 pixels wide; if you have a modest 1024x600 screen (very slightly narrower than 16:9) then you can divide 960 of the 1024 pixels into four 80-column columns, with another 64 pixels left over for margins, scrollbars, or other UI chrome. Each of these columns then holds 100 6-pixel-high lines of text, for a total of 400 lines, or six printed standard 80x66 pages.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Displays (p. 3414) (13 notes)

Multiplication with squares

Kragen Javier Sitaker, 2017-07-19 (updated 2019-07-09) (5 minutes)

This is a table-lookup-based multiplication algorithm I wrote about some years ago which is not in much current use, even though it's much less computationally expensive than the standard algorithms.

$(a + b)^2 = a^2 + 2ab + b^2$
$(a - b)^2 = a^2 - 2ab + b^2$
$(a + b)^2 - (a - b)^2 = 4ab$
$ab = ((a + b)^2 - (a - b)^2)/4$

```
def mul(a, b):
    c, d = a + b, a - b
    csq, dsq = c**2, d**2   # Imagine these are table lookups.
    e = csq - dsq
    prod = e/4

    print 'c=%s, d=%s, c²=%s, d²=%s, c²-d²=%s, so %s·%s = %s' % (c, d, csq, dsq, e, a, b, prod)
```

For example, to multiply 7984839 by 11859552, first find their sum and (unsigned) difference: 19844391 and 3874713. Find the square of each: 393799854160881 and 15013400832369. Now find the difference of these squares: 378786453328512. Now divide this by 4 (an easy operation on a binary computer; somewhat trickier in decimal): 94696613332128. And that is indeed the product of these numbers.

If you have a table of squares to do lookups in, this involves a sum, two differences, two table lookups, and a division by 4 (a shift right by two bits, in binary). These all involve a number of bit operations that grows linearly with the number of bits in the input numbers, but require a square table whose size is exponential in the size of the input numbers.

(This algorithm was known to the ancient Babylonians, who tabulated squares divided by 4 to facilitate it; flooring the squares is harmless in this case.)

One disadvantage of this algorithm is that, to multiply numbers up to N, your table of squares must include numbers up to 2N. If the numbers you are multiplying are both even or both odd, a slightly different identity allows you to use a table of squares of only N numbers:

$((a + b)/2)^2 = ¼(a^2 + 2ab + b^2)$
$((a + b)/2)^2 - ¼a^2 - ¼b^2 = ½ab$
$2((a + b)/2)^2 - ½a^2 - ½b^2 = ab$

This requires three table lookups instead of two, and the same three additions and subtractions, plus a couple of halvings and a doubling rather than two halvings. If the table contains the halves of the squares, you save the halvings but need a final multiplication by 4.

So, consider 61·65. (a+b)/2 = 63, and $63^2 = 3969$; twice that is 7938. $½61^2 = 1860½$, and $½65^2 = 2112½$. So our final result is 7938 - 1860½ - 2112½ = 3965, and this is correct. This didn't require us to find the squares of any numbers bigger than our multiplicands.

However, rounding the numbers in the table of half-squares would be unsafe for this algorithm.

All known bounded-space multiplication algorithms take a number of bit operations that grows superlinearly in the number of bits in the input numbers: multiplying by partial products takes $O(n^2)$ operations (which was conjectured to be optimal from 1952 until 1960), Karatsuba multiplication (see Karatsuba (p. 2090)) takes $O(n^{1.59})$ operations, Toom-Cook multiplication takes $O(n^{1.47})$ operations, Schönhage-Strassen multiplication takes O(n log n log log n) operations, and Fürer's algorithm and the De-Saha-Kurur-Saptharishi multiplication algorithm take time that is still superlinear, but only barely.

However, for numbers of under about 100 decimal digits, multiplying by partial products is the fastest of the above-listed algorithms in practice. But this square-table algorithm is faster than multiplying by partial products even for numbers of moderate size.

For example, the above calculation involved calculating 44 decimal digits of results; doing the calculation by summing seven eight-digit partial sums into the 14-digit result would have involved calculating 60 decimal digits of results. So this is an efficiency improvement. However, it presupposes a table of over eleven million squares, which would be several volumes if printed on paper.

The crossover point where this algorithm uses fewer bit operations is probably around 16 bits. The square table becomes impractical in size somewhere between 8 bits and 50 bits, depending in part on how long you can wait to get the results. If you're multiplying a lot of pairs of numbers, it might actually make sense to pipeline a millisecond's worth of products while you're waiting for the square table lookup results to come back from a highly parallel, distributed square table. (See Hardware multiplication with square tables (p. 1886).)

Topics

- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- History (p. 3500) (71 notes)
- Multiplication (p. 3590) (3 notes)

What is the type of lerp?

Kragen Javier Sitaker, 2017-01-08 (5 minutes)

The linear interpolation operation, lerp, is fundamental to a number of algorithms; you can derive Bézier curves and B-splines from it, it provides a differentiable version of the conditional operation, and of course it is directly applied in computer graphics and numerical computation of all kinds.

There are some interesting questions related to its type.

Definition

It's sometimes defined as a function of five parameters:

$\mathrm{lerp5}(x_0, x_1, y_0, y_1, x) = y_0 + (x - x_0)\cdot(y_1 - y_0)/(x_1 - x_0)$

This gives you the interpolated value of y at x, given that it should be y_0 at x_0 and y_1 at x_1.

However, this can be productively decomposed into two functions of three arguments:

$\mathrm{howfar}(x_0, x_1, x) = (x - x_0)/(x_1 - x_0)$
$\mathrm{lerp}(y_0, y_1, t) = y_0 + t\cdot(y_1 - y_0)$
$\mathrm{lerp5}(x_0, x_1, y_0, y_1, x) = \mathrm{lerp}(y_0, y_1, \mathrm{howfar}(x_0, x_1, x))$

And it is the lerp function defined in the middle there that we are concerned with.

Given the usual algebraic identities, we can transform its formula into a different form that is sometimes easier to use. Writing out the algebraic transformation in great detail, we have

$y_0 + t\cdot(y_1 - y_0) =$
{distributivity}
$y_0 + t\cdot y_1 - t\cdot y_0 =$
{unity}
$1y_0 + t\cdot y_1 - t\cdot y_0 =$
{definition of subtraction in terms of adding negative}
$1y_0 + t\cdot y_1 + -t\cdot y_0 =$
{commutativity}
$1y_0 + -t\cdot y_0 + t\cdot y_1 =$
{distributivity}
$(1 + -t)\cdot y_0 + t\cdot y_1$
{definition of subtraction in terms of adding negative}
$(1 - t)\cdot y_0 + t\cdot y_1.$

I'm going to consider lerping in domains where some of these identities don't apply, and (I assert) it will be interesting to consider which ones.

This leads to the definition

$\mathrm{lerp2}(y_0, y_1, t) = (1 - t)\cdot y_0 + t\cdot y_1$

I believe this was the computation Turing originally proposed to compute the definition of a conditional branch instruction in his proposal to build a computer.

Type signature of lerp

What are the types of the parameters and result of the lerp function given above? They need not all be the same; in the fully general case, there are six of them:

$\mathrm{lerp}(y_0\colon T_0, y_1\colon T_1, t\colon T_2) = y_0 + (t \cdot ((y_1 - y_0)\colon T_3)\colon T_4)\colon T_5$

which gives us the type signatures for the component operations:

$(x\colon T_1) - (y\colon T_0)\colon T_3$
$(x\colon T_2) \cdot (y\colon T_3)\colon T_4$

$(x: T_0) + (y: T_4): T_5$

So far, this is sort of vacuous. But we only have to add a couple more constraints and it gets interesting! The usual case in lerping is that you want the result to be y_0 sometimes, y_1 other times, and somewhere in between at still other times. For that to be a coherent wish, those three values need to have the same type:

$T_0 = T_1 = T_5$

This reduces our component operation signatures to the following:

$(x: T_0) - (y: T_0): T_3$
$(x: T_2) \cdot (y: T_3): T_4$
$(x: T_0) + (y: T_4): T_0$

If we arbitrarily add the additional constraint that $T_3 = T_4$, we have a simple algebraic structure that looks like an affine space:

$(x: T_0) - (y: T_0): T_3$
$(x: T_2) \cdot (y: T_3): T_3$
$(x: T_0) + (y: T_3): T_0$

Here T_0 is the affine space, T_3 is its associated vector space, and T_2 is the underlying scalar field of the vector space. Computing lerp doesn't depend on the validity of any of the eight vector axioms or the axioms of the affine space, but to the extent that those axioms hold, more interesting properties will hold of lerp's results. For example, when $t=1$, normally you want the result to be y_1, but in many practical cases with floating-point numbers, it won't be!

To take a concrete example where the three types are different, T_0 might be (the type of) a mapping from a set of (lat, long) pairs to a temperature reading represented as a floating-point number interpreted in degrees Celsius; T_3 might be a mapping from a set of (lat, long) pairs to a temperature difference represented as a floating-point number interpreted in kelvins; T_2 might be simply a single unitless floating-point number; and the three arithmetic operations might operate pointwise on their values.

This is practically useful in finding errors in programs because it is physically meaningless to multiply a number interpreted in degrees Celsius by some number. It is 22°C here right now; if I multiply 22 by 2, getting 44, and then interpret that 44 as 44°C, I have computed a temperature with no meaningful relationship to the original temperature. In Fahrenheit, these temperatures are 71.6°F and 111.2°F. If a program is doing such a computation, it is very likely erroneous, although not in every case.

This definition of lerp does not do such a computation.

Topics

• Programming (p. 3658) (286 notes)
• Graphics (p. 3483) (91 notes)
• Math (p. 3564) (78 notes)
• Algebra (p. 3309) (11 notes)
• Types (p. 3758) (5 notes)

Phosphorescent laser display

Kragen Javier Sitaker, 2016-08-16 (8 minutes)

Scanning an ultraviolet diode laser over a cheap phosphorescent screen should give you a very inexpensive, high-resolution, very-low-refresh-rate display screen.

Projecting an image with a laser suffers from a few problems. One is that, if the part of the laser beam that would fit through someone's pupil is bright enough, it can be an eye safety hazard. Another is that the beam must be scanned very rapidly over the screen in order to provide the illusion of a stable image, and equipment for scanning the beam very rapidly is energy-hungry and expensive.

We can solve both of these problems with multiple ultraviolet lasers.

Eye safety

Ultraviolet light below about 400nm is blocked by the lens of the eye, so it will not be focused onto the retina and cause the instant retinal burns that are the principal danger of visible and especially infrared lasers. However, it can cause lens damage if continued over a period of time, including cataracts. Below 315nm, it cannot even penetrate the cornea, so it instead will cause only acute photokeratitis, which will heal unless it is very severe indeed.

However, in discussing the *kinds* of damage that can be caused, we risk losing sight of the *quantitative* safety factor. The IEC 60825 maximum permissible exposure for a 355nm ultraviolet laser over the course of a millisecond is 100 watts/cm², falling to under 1 watt/cm² if the exposure extends to an entire second, and down to 1 mW/cm² if the exposure continues for a kilosecond. By contrast, the MPE for a visible-light laser is 10 milliwatts if for a millisecond, or 3 milliwatts if for an entire second, converging with the ultraviolet MPE levels at 1000 seconds.

That means that the safe power levels for brief exposure to ultraviolet lasers are around 100 times higher than for visible-light lasers. This means you can send a great deal more energy to your screen at a safe power level. It might be a good idea to wear UV-blocking goggles and to couple your UV laser with a lower power, but painfully bright, but not dangerous, red laser, in order to trigger people's blink reflex. 1mW should be plenty.

An alternative way to provide eye safety under normal circumstances is to project the laser onto one side of a screen while you look at the other side, enclosing the laser, scanning apparatus, and back of the screen inside a sealed, opaque box.

One unfortunate aspect of this approach, though, is that there aren't any laser diodes of shorter wavelengths than 370nm commercially available yet. https://www.thorlabs.com/NewGroupPage9.cfm?ObjectGroup_ID=5400 offers Thor Labs's new 375nm 70mW ultraviolet laser diode L375P70MLD for US$4300.

Refresh rate

A phosphorescent screen will not only convert a certain fraction of

the laser illumination into light, but also continue to glow over a long period of time, exponentially decaying. This acts as a single-pole low-pass filter on the image signal, attenuating frequencies faster than the time constant of the phosphor by 3dB per octave.

Zinc sulfide's phosphorescence decay time constant is a few seconds to a few minutes. (I found some paper claiming 9', but that seems implausibly long to me from experience with glow-in-the-dark toys). This means that once a glowing image has been drawn on the zinc sulfide with the laser beam, it will stay there, gradually fading, for a few minutes.

This means that you can draw an image on the screen with a laser over a period of seconds or minutes, and it will continue to be visible. This means that you can draw a fairly complex image even with a fairly slow apparatus for scanning the laser beam across the screen. It also means that you can't erase anything: you have to wait for it to fade.

It also means that it takes seconds to minutes for the image to reach full brightness, but because of the logarithmic brightness perception of human vision, this is not as much of a problem as you would expect. (I'm guessing this from my experience with analogue oscilloscopes with zinc sulfide screens.)

Copper-doped zinc sulfide is by far the most common glow-in-the-dark material.

It might be worthwhile using a secondary, say, red laser to draw a smaller amount of graphical information that can be instantly erased. This will work better if the screen is not sensitive to the wavelength of the secondary laser.

Erasing

https://physics.stackexchange.com/questions/79860/why-is-a-laserpointer-able-to-erase-a-glow-in-the-dark-sticker reports that a red laser pointer was able to *erase* the glow from a glow-in-the-dark sticker (presumably ZnS:Cu). There is a video of this phenomenon at https://www.youtube.com/watch?v=kUteUH7mz0A, but the erasure seems temporary.

Multiple lasers

Laser diodes themselves are relatively inexpensive; Digi-Key has 1.5mW infrared lasers at US$5.76 and red 5mW lasers at US$12.52. But as the power goes up, cost increases sharply. Their cheapest 20mW laser is US$46.07 (green), their cheapest laser over 40mW is a 120mW 405nm near-ultraviolet unit for US$78.44 (this is the wavelength used by Blu-Ray players), and the only more powerful laser diode for which they list a price is an 175mW near-ultraviolet unit for US$452.

Given this price curve, you can probably get not only more visible light output but also more information on the screen by using several different laser diodes, each pulsed rather than CW, so that sequential points on the screen are often drawn by different lasers. Using two to six separate lasers will increase the energy throughput of the laser bottleneck significantly, without affecting the rest of the system.

Also, laser diodes can be controllably pulsed much more rapidly than mirrors can scan the beam — MHz in the common case or GHz

in exceptional cases — and you can draw minimally readable letterforms by interrupting three or four vertical lines:

```
 #  ##  # #  ##  ###  #  # #  #
# # # # ### #  # #  # #### ## #
### ##  ### #### ###  #### # ##
# # # # # # #  # #  # #  # #  #
# # ##  # # #  # ###  #  # #  #
```

Given three or four laser beams with a slight angle offset between them horizontally, you could sweep them vertically with a single movement of the mirror while pulsing different dash patterns on them to draw the letters.

Resolution calculations

You should be able to do a few megapixels this way, but probably not much more.

Suppose you're using a 2m² screen, with the laser 1.5m away, and you have a 1mRad-divergence beam, which is a pretty normal divergence for a laser pointer. Then your spot will be 1.5mm across, so your screen is only about 1300×1300 "pixels", for a total of 1.8 megapixels. If you can get a better-quality laser spot of 0.5mRad, you can get four times that, or 7 megapixels.

(The diffraction-limited divergence angle is $2\lambda/(\pi w)$, where λ is the wavelength and w is the beam-waist radius. So for a 650nm red laser to have 1mRad divergence, you need $w > 2 \cdot 650\text{nm}/(\pi \cdot .001) \approx 0.4$ mm. Larger collimating optics can produce smaller divergence, but you aren't going to get that beam waist below 0.4mm even if the beam waist is on the screen.)

However, you may only be able to illuminate a small fraction of these pixels at a time; even expensive laser-show galvos are rated at under 50kpps, and even in two minutes, 50kpps is only 6 million points. Simpler scanning apparatus, perhaps driven by scavenged hard disc voice coils or by paper cone speakers, might only hit 1kpps, and thus only 120k pixels illuminated per laser. With multiple lasers and dash patterns, you could actually paint all of those pixels, a few thousand per second.

Topics

Can you bitbang USB with an ATMega's RC oscillator?

Kragen Javier Sitaker, 2017-04-04 (1 minute)

An overview of USB says low-speed data is "clocked at 1.50Mb/s with a data signalling tolerance of ±1.5% or 15,000ppm" so you can clock it with a [ceramic] resonator instead of needing a crystal.

AVRs' internal RC resonators are mostly not quite this precise; even at a fixed temperature and voltage and after user calibration, they're rated to vary by 2%, at least on the ATtiny2313. But on the ATMega series (at least the 48/88/168/328 series used in the Arduino), they're rated to ±1% under these circumstances. So a resonator is needed to bitbang low-speed USB on the ATtinies, but maybe not the ATMegas.

Topics

Granite texture

Kragen Javier Sitaker, 2019-05-08 (updated 2019-05-09) (5 minutes)

(Probably far from an original idea.)

I was thinking about texture generation today, and in particular what you can do in a fragment shader, where deciding which pixels to affect is not a thing you can do. This seems like it could be a real problem, since many real-world textures are the result of a lot of different objects moving around; for example, the exposed stones in a sawn concrete surface are in some sense scattered randomly, as are the leaves on a forest floor. But in a fragment shader you can't just generate some random points and place leaves at them, at least not in a way that scales when you zoom out.

Random points don't look random

I was reminded that when people look at independent identically distributed random points, they generally think they don't look very random, because clusters of points randomly occur, and so the density of the points varies even at fairly large scales. Many natural textures — the stones in the concrete I mentioned before, for example, but also hairs on the skin, cones on the retina, bubbles in a foam — break up such clusters by a sort of "relaxation" in which points move away from one another, evening out the medium-scale density variations, and eventually the large-scale ones too.

Maybe random-looking points can be a perturbed periodic lattice

But maybe another approach would be to start with a very even dot distribution and perturb it enough that it looks random. You could have some perfectly regular lattice of cells, with a dot at the center of each — a square or hexagonal lattice — and generate a two-dimensional value of Perlin noise by which to perturb the dot at that center. As long as the dot doesn't overlap the next cell, the algorithm to determine the color of a pixel is very simple; if z is the pixel coordinate:

```
c = round(z)
fragColor = hypot(c + r1 * noise2d(c) - z) > r2 ? bg : fg
```

If we expand r2 or especially r1 to the point that the dots start to wander into adjacent cells, that simple seven-instruction algorithm starts to fail; if we run it for all the adjacent cells, though — 5, 7, or 9 of them, depending on how many candidates there are — we can determine which of the neighboring cells' dots are overlapping us, at the cost of a work multiplier of 5, 7, or 9.

Maybe you can build a fake Voronoi diagram this way

In the same way, you can draw an almost-Voronoi diagram by having no dot-radius threshold, just coloring the pixel according to which dot's center it's closest to. This will occasionally depart from

the real Voronoi diagram because long, sharp projections will occasionally be truncated early by cell boundaries; perturbing the cell boundaries slightly with more noise may be a good way to keep that subtle.

How to draw granite

Well, what's granite? Granite consists of a lot of crystals of minerals of different colors which grew as the magma cooled, each one nucleating with some random position and orientation and growing around that center, faster along some planes of their crystal structure than along others. At first only zircon and calcium-rich plagioclases can crystallize (or forsterite, but it doesn't occur much in granite), but as the temperature drops, other minerals like pyroxenes, more sodic plagioclases, micas, and eventually even quartz can crystallize. As the crystals grow, they deplete the local magma of their own mineral, which means that when the leftover magma eventually does crystallize, it will be of a different color.

This is precisely the kind of prolonged multivariate field dynamical process that's hard to simulate in a fragment shader†, but perhaps we can generate a similar-looking result by perturbing the Voronoi distances according to a random skew matrix. That is, before computing the magnitude of the displacement from the pixel to a dot center, multiply that displacement vector through a skew matrix particular to that dot (generated from yet another call to noise). This should make the "crystal" tend to be longer in some apparently-random directions, and shorter in others. If the skew matrix also has a random determinant, some "crystals" will be larger and others smaller.

This is a terribly goofy way to generate this image, though, as you can solve precisely where the grain boundaries are going to be. You don't really need to do all this computering for every pixel.

† Could be harder — at least crystallizing magma isn't turbulent.

Topics

Marking metal surfaces with arcs

Kragen Javier Sitaker, 2016-10-06 (4 minutes)

Engrave a metal surface with a controlled low-energy arc because, like a laser, it can deliver very high power levels, but it's a lot easier to build than a laser is. You can vaporize a little bit of metal, or you can melt it and then blow it away with air.

There are a couple of things that make it difficult to melt metals. One is that most metals don't melt until they're pretty hot, so you can't melt them at all with things that only get moderately hot, like most flames. (Jet fuel, as they say, can soften steel beams until they can support almost nothing, but not melt them.) The other is that it's a good heat conductor, so if you deliver the heat slowly, it gets conducted away and the metal never gets very hot. Aluminum especially is a good heat conductor.

The easiest approach is to bring an electrode, charged with a capacitor, toward the surface of the oppositely-charged metal until the air breaks down and the energy discharges in a spark.

Striking an arc in air needs on the order of 300 volts at a minimum at a distance of about 13 microns. If you have a 1-microfarad low-ESR capacitor supporting that arc, it has an energy of about 45 millijoules. Vaporizing aluminum takes about 14 kilojoules per gram `(((2470 - 20) K 24.20 J/mol/K + 10.71 kJ/mol + 284 kJ/mol) * 27.0 g/mol)`, the majority of which is from its heat of vaporization. That means that this spark can vaporize about three micrograms of aluminum, which sounds insignificant, but if it's hemispherical, it's actually a crater about 160 microns across, which you will note is more than ten times the distance from the electrode to the workpiece. A 160-micron crater is clearly visible and palpable; it's comparable to the kerf you get from a laser cutter.

That's kind of a best case, though, because some of the heat will go into heating the electrode and the air, some of it will be conducted away, and some of it will go into heating the already boiled metal in its gaseous form. If the workpiece is connected to the negative side of the circuit (the cathode), most of the heat of the arc will be deposited at the surface of the workpiece, as it is bombarded by ionized air, rather than on the marking electrode, which is receiving only electrons.

Thermal runaway concentrates the electrical current on the hottest part of the cathode, as that's the part that can emit the largest number of electrons, so the spot that the arc heats can be very small indeed.

That still leaves the question of how fast the whole discharge happens, which depends crucially on the E/I curve of the arc, where most of the resistance in the circuit is found. The RC time constant of just the electrode and wires can easily be around a microsecond, which would imply a power of around a kilowatt and a power density of 50 gigawatts per square meter or 50 kilowatts per square millimeter, which is in the neighborhood of what metal-cutting lasers put out, so it should probably work okay. (WP says 1500 watts in a 25-micron laser spot is common, and you can cut 1-mm aluminum at 14.82 cm/s at 1000 W; if we figure the kerf is 200 μm, that works out to 12.5 mJ/μg, very close to the value of 14 I calculated above; and at

that rate the laser is vaporizing a volume of material comparable to our crater every 36 microseconds, which should be an easily achievable speed for the spark.)

Topics

A principled rethinking of array languages like APL

Kragen Javier Sitaker, 2015-05-16 (updated 2019-09-30) (31 minutes)

What accounts for the power and convenience of array-processing languages like Matlab, R, and APL? Can we get it without having to pay the high readability and bugginess costs associated with these languages?

I feel that I may finally have an account of what's going on here, and not only can we get better error-checking out of it, we can get a language that is more expressive (in the sense of more power in less tokens), more readable, and more efficient than traditional array-processing languages.

Though I think the present note is self-contained, the initial development of the idea was in Index set inference or domain inference for programming with indexed families (p. 1434) in probably 2007 or 2008.

The basic idea

A variable in a computer program has a value that varies. Sometimes when a piece of code runs, the variable will have one value; other times, it will have another value. (In many languages it can change even during the same run, but that is not relevant to what I am considering here.)

These values are functions of some arbitrary set of inputs, often implicit in their context. In image processing, a brightness value might vary by X, Y, color channel, and frame number; in statistical processing, an elapsed-time measurement might vary by trial number; in rendering, a pixel color might vary by shading algorithm, camera position, and the states of all the objects in the input scene. It can be hard to tell what they depend on, and indeed assuming that a variable is constant relative to a given input, when in fact it ought to vary, is a common source of bugs. (You could argue that this is the source of the difficulty of caching.)

We could think of these inputs as being dimensions in a many-dimensional space, and each point in this space as being a possible universe — perhaps a very small possible universe with only a few variables in it, but a universe nonetheless.

Array languages allow us to reify this space in runtime variables, and thus to write programs that act across entire subspaces of it, rather than the pointwise and one-dimensional approach taken by ALGOL-family languages like C or Java.

Some examples from ray-tracing

Consider this C code, from My Very First Raytracer:

```
static color
pixel_color(world here, int ww, int hh, int xx, int yy)
{
  vec pv = { (double)xx/ww - 0.5, (double)yy/hh - 0.5, 1 };
  ray rr = { {0}, normalize(pv) };
```

```
  return trace(here, rr, 1.0);
}
```

This code is invoked a number of times with the same `world`, `ww`, and `hh` variables, but with varying values for `xx` and `yy`; but you can't tell that from looking at it. Similarly, when it invokes `trace`, it invokes it many times with the same `here` value, different `rr` values, and the same `importance` value of 1.0. But it needs to be written as a function, or at least a loop, to allow this flexibility in `xx` and `yy`.

In a sense, in this code, `xx` represents not a single integer, but an entire plethora of possible values, maybe even an infinite series of values; it's not a scalar, but rather a function of which pixel we're looking at. When we write the division of `xx` by `ww`, we are not writing the division of a single floating-point number by a single integer, but rather all the floating-point numbers `xx` will ever convert to in the lifetime of humanity, by all the possible image widths `ww` will ever contain in the lifetime of humanity. Or, if we limit our perspective to a single execution, that division instruction will eventually be used to divide all of the horizontal image pixel coordinates by the image width — redundantly, many times, in fact, once for each line; so it's a machine instruction that implicitly represents a vector operation.

But this is a rather unusual hermeneutics of the C and machine code.

The C code enforces a particular order of evaluation: it is not capable of beginning to evaluate a second call to `trace()` until the first one is done, and no way to evaluate a second call to `pixel_color` until the first one is done. But this may not be the most efficient way to find out what color the pixels should be.

The C code, you'll notice, also has a fair bit of syntactic overhead associated with allowing these variables to vary; they have to be declared as parameters. What if, instead, we were programming in a language where these variables explicitly, in the source text and in memory at run time, represented an entire vector of possibilities — a sort of more principled APL? Maybe we could write it something more like this:

```
pixel_color = trace(here,
                    ray(vec(0,0,0), normalize(vec(xx/ww-.5, yy/hh-.5, 1))),
                    1.0);
```

(We're also getting some brevity here by not having to name the temporary structs.)

A language with an APL-like evaluation strategy could figure out that `xx` and `yy` vary independently, while `ww` and `hh` don't vary at all, and so generate a ρ`xx`×ρ`yy` space of possibilities for the `vec`s that we're normalizing, where ρ is the APL operator that gives the shape of a matrix. (More detail on how to do this is in the next section.)

I think that's the ultimate philosophical justification for APL's conformability and broadcasting rules; if you're ray-tracing 640 columns and 480 rows of pixels, for example, then a value that is constant for all those pixels is merely a scalar; a value that varies by column but not by row will be a 640-element vector; a value that varies by row but not by column will be a 480-element vector; and a

value that varies by pixel will necessarily be a 640×480 matrix. So it makes sense to divide `xx` by `ww`, or `yy` by `hh`, but doing anything with the two of them together requires an outer-product operation (which in APL is explicit) or reducing one or the other of those axes of variation into being some kind of dummy variable, like an index of summation or whatnot.

But APL of course can't tell that your 3-element vector representing the X-coordinates of the three spheres in your scene isn't really compatibly dimensioned to a three-element vector that represents the X, Y, and Z coordinates of the camera, say. Array-dimensions typing is weak typing, much like the currently fashionable approach of typing everything as a string. And this is why outer products are necessarily explicit in APL, while I think a more principled array-processing language could infer most of them.

Here's another example, from the same program:

```
static vec
scale(vec vv, sc c) { vec rv = { vv.x*c, vv.y*c, vv.z*c }; return rv; }

static ray
reflect(ray rr, vec start, vec normal)
{
  // Project ray direction onto normal
  vec proj = scale(normal, dot(rr.dir, normal));
  // Subtract that off twice to move the ray to the other side of surface
  vec reflected_dir = sub(rr.dir, scale(proj, 2));
  ray rv = { add(start, scale(reflected_dir, 0.001)), reflected_dir };
  return rv;
}
```

(The .001 fudge factor there is to keep the reflected ray from hitting the same surface again from the inside due to rounding errors.)

The `scale` function here obviously only exists because C is not an array-processing language. Or does it? If we were trying to write a `reflect` that handled many normals at once in arrays, it wouldn't be totally insane to use three separate arrays of X, Y, and Z components of the normals. Taking just the first line of `reflect` and translating it into C written as if it were Fortran:

```
static void
scale(sc vx[], sc vy[], sc vz[], sc c[],
      sc vox[], sc voy[], sc voz[], int n)
{
  for (int ii = 0; ii < n; ii++) {
    vox[ii] = vx[ii] * c;
    voy[ii] = vy[ii] * c;
    voz[ii] = vz[ii] * c;
  }
}

static void
proj(sc vx[], sc vy[], sc vz[], sc dx, sc dy, sc dz,
     sc vox[], sc voy[], sc voz[], int n)
{
```

```
  sc dots[n];
  for (int ii = 0; ii < n; ii++) {
    dots[ii] = vx[ii] * dx + vy[ii] * dy + vz[ii] * dz;
  }
  scale(vx, vy, vz, dots, vox, voy, voz, n);
}
```

You may not agree yourself that this would not be totally insane, but hopefully you can agree that this is a way to do this that Fortran programmers would think was not totally insane. Also you can see that an enormous benefit of APL over Fortran for this kind of thing is that you have at least some hope of changing your mind about whether it was the rays or the normals or both that you wanted to have more than one of, because making the loops and indexing implicit there means that you don't have to change the code to index into a `dx` array and maybe not index into a `vx` array.

(Also there are probably some combinations of dimensions and CPU models for which the more predictable memory access of this version would actually make it faster despite its smaller ratio of computation to memory locations accessed. And obviously if your loops are implicit and your non-implicit operations are subject to interpretation overhead, as in Numpy or normal APL implementations, the array approach is going to be hugely faster.)

Coordinates in three-space, though, are definitely the kind of thing that it's reasonable to think of as numbering from 0 to 2 or from 1 to 3, rather than being unrelated attributes of the same thing. Then you might want your array of normals here to be represented as an n×3 array rather than three arrays of n or an array of n structs with three fields. And then things like the `scale` function fall away entirely, but you need some way to specify which dimension you're summing over when you compute that dot product. APL +/ has a default of operating over the *last* dimension, and the option of specifying a different dimension by its numerical index, as in `+/[1]`.

This seems ad-hoc and unreadable to me, like much of APL. But if you have named your axes of variation, and one of them is the XYZ distinction, then you could very reasonably say `XYZ.sum()` or `+/[XYZ]`, and it would be clear, turning the XYZ variation into a dummy variable; if you applied it to some kind of aggregate with more than one XYZ distinction (introduced with an explicit outer-product operator) or no XYZ distinction, you would get an error.

And then you could write

```
proj = normal * XYZ.sum(raydir * normal)
```

instead of the 20 lines of crap above, and furthermore keep that abstract over whether you have a single normal and many ray directions, a single ray direction and many normals, many normals of which each corresponds to a different ray direction, or even (what APL could never do implicitly) many ray directions and many normals, of which we implicitly want the Cartesian product.

And then maybe you could write the whole function like this:

```
proj = normal * XYZ.sum(raydir * normal)
reflected_dir = raydir - 2 * proj
```

```
reflect = ray(start + reflected_dir * 0.001, reflected_dir)
```

When it comes to implicitly broadcasting operations over different dimensions, C is equivalently succinct to an array language — modulo the data typing and abstraction overhead that it requires in order to give you variables at all. But because C values are only implicitly, in an esoteric hermeneutics, vectors or universes of possibilities, it is difficult to write something like the `XYZ.sum` function above; instead we are reduced to writing explicit loops, or as in this case, explicitly textually repetitive code.

Getting more rigorous: a functional semantics with implicit arguments

Okay, "rigorous" and "semantics" may be an exaggeration. But I'll try to at least outline a rigorous semantics here.

Suppose that, instead of considering variables such as `normal` to hold scalars, vectors, or matrices of some finite size, we instead consider them to hold computable functions, but functions whose domain is not necessarily known or finite. This is not a big leap: we can consider the vector [6 832 4] as a function f over a domain of three integers: it returns 6 when invoked as f(0), 832 when invoked as f(1), or 4 when invoked as f(2).

In this interpretation, we lift the usual arithmetic operators to operate over functions of one argument in the usual way: `*`, for example, is the function we would usually write in the λ-calculus as `λf.λg.λp.(f p)*(g p)`, or in Python as `lambda f, g: lambda p: f(p)*g(p)`; and we consider constants such as `2` to denote constant functions like `K 2` or `lambda p: 2`.

But what is this mysterious `p` argument? It's a context or point in this multidimensional possibility space mentioned eralier, the one that's usually left implicit, so it needs to include all the axes of variation we were talking about earlier; to get traditional APL semantics, you would want it to be something like a stack of numbers, but dicts are more fashionable these days than stacks, arrays, or lists, so let's consider it to be something like a Lisp alist indexed by symbols, each symbol denoting some axis of variation.

So now we can interpret this line:

```
proj = normal * XYZ.sum(raydir * normal)
```

as meaning (in Python):

```
def proj(p):
    return normal(p) * sum(raydir(q) * normal(q)
                           for q in XYZ.possibilities_augmenting(p))
```

Here `possibilities_augmenting` is a method of the `XYZ` dimension that gives you versions of the point `p` with all possible values of `XYZ` substituted into it. Thus the first call to `normal` might return either the `x`, the `y`, or the `z` component of some particular normal; but all three of those will be multiplied by the same dot product.

Of course, this is not intended to suggest that it must be calculated in this fashion, which would be immensely wasteful; it's intended as a specification of the relationship between inputs and outputs.

This suggests an implementation of the `vec` function mentioned earlier, which in the C program was a `struct` type:

```
def vec(x, y, z):
    values = {XYZ.x: x, XYZ.y: y, XYZ.z: z}
    return lambda p: values[p[XYZ]](p.without(XYZ))
```

That is, the functions produced by `vec` consume the `XYZ` dimension of their input and dispatch to any of the three functions that were passed in as their X, Y, and Z components. So this expression from the `pixel_color` function:

```
vec(xx/ww-.5, yy/hh-.5, 1)
```

when invoked with `z` will dispatch to the constant function denoted by `1`; when invoked with `y`, will dispatch to the function denoted by `yy/hh-.5`, which itself will dispatch to `yy`, which in this program varies by pixel, and to `hh`, which doesn't vary at all during a run of the program, and to another constant function that returns 0.5.

Another useful higher-order function is a "renaming" or "aliasing" or "axis rotation" or "reshaping" function which turns one axis into another:

```
def rename(a, b, f):
    return lambda p: f(p.without(a).with(b, p[a]))
```

Considered spatially, this prevents `f` from varying over axis `a`, rotating the motion of `p` along `a` into motion along the new axis `b`. Considered relationally, this renames column `b` of the inputs to relation `f` to `a`. This is the operator you need for carrying out explicit outer products; if `f` and `g` are both vectors on axis `b`, then `rename(a,b,f)+g` gives you their outer product sums, with the values of `f` varying along the new axis `a` and the values of `g` varying along axis `b` as before. (This `rename` function also gives you general axis transposition, in a sense.)

This "context" or "point" object may carry a whole collection of context attributes with it that most of these functions don't bother to access, and can pass along to their callees without mentioning them explicitly.

(If we think in terms of N-ary relations rather than in terms of functions, you could think of this "point" as a query-by-example partial record. But that's not very consistent with the functional semantics described above.)

In theory, if all of your component functions being combined by means such as lifted operators, `rename`, axis-construction functions like `vec`, and dummy-axis-introduction functions like `XYZ.sum`, have finite discrete domains along some axis, you ought to be able to compare those domains and detect mismatches, and then you ought to be able to describe the multidimensional domain of the combined function. This is a lot like type-checking. You might even be able to do it at compile time, and if you have compile-time axis variables analogous to type variables in parametric polymorphism, you might be able to do the type-checking polymorphically at compile time.

(Also note that this eliminates run-time bounds-checking, just as structs do.)

APL has some axis-transformation functions: compress, expand, take, drop, and the sort of hidden operation of indexing a vector by another vector, which is like binary relation composition or like a different form of compress. You could consider these either as generating new axes or as subsetting the domain along an existing axis. In APL, it's the former, and so you have to be careful to compress all the attributes you care about by the same boolean vector, or index all of them through the same index vector.

It seems like it might be more useful here to implicitly intersect domains along the same axis, which is after all what we are doing when we implicitly take the outer product of a scalar and a vector. However, the operation of *obtaining the valid indices* along some axis or all axes (i.e. ρ) then must introduce a new axis to arrange its results along.

Inter-loop dependencies

So far, all of the above, however nicely it motivates and elaborates APL's default rules for conformability and broadcasting, only covers scalar operations and nearly trivially parallelizable vector operations with no interloop dependencies. Operations like reverse, rotate, grade-up (indirect sort), scan (prefix sum), and even take and drop don't treat the points along the axis as floating in space independently, but rather having a total order, with even predecessor and successor operations, and correspond in languages like C to loops with interloop dependencies.

I can imagine a bunch of different possible ways to handle these: all axes could be ordinal; grade-up could create an ordinal axis from a non-ordinal axis or axis subset, and scan, reverse, rotate, take, and drop could simply fail to compile when applied to non-ordinal axes; instead of rotation you might have a "next index" or "previous index" function which, since it knows which axis it's acting along, knows when to wrap; and so on.

This is an important area to do a good job in, and there will be nonobvious interactions among factors. These are, of course, the areas in which ALGOL-family programs have to declare data structures and SQL optimizers start having to plan out join plans, so we shouldn't expect easy wins in this area.

My raytracer example is in some sense carefully chosen to minimize this; it constructs almost no intermediate data structures, unless you count 3-vectors as "data structures".

Efficiency

GPUs, but also multithreading and SIMD instructions and cache prefetch and improved locality by avoiding memory access to unused columns. "Blocking" or "tiling" for efficiency; also "deforestation".

Parallel prefix sum and parallel sorting are well-studied problems. To the extent that these operations are sufficient to efficiently solve computational problems, we should expect programs written in this fashion to benefit from fine-grained parallelism more easily than regular programs.

Rethinking this again, the basic idea is that some variables have values that depend on the circumstances, and there are a variety of circumstances (or dimensions or scales) that may or may not be relevant to the value of any given variable. The latitude and

longitude of each house on a block are different, but perhaps we consider the temperature of the entire block to be identical — but it varies by time of day, which the latitude and longitude do not.

There are different kinds of dimensions; Stanley Smith Stevens described them as "levels of measurement". Some are categorical/nominal rather than quantitative; quantitative dimensions can be ordinal (sortable), interval dimensions (subtractable; affine?), or ratio dimensions (with a zero element and thus divisible). Also, quantitative dimensions may be discrete or continuous.

Pointwise operations on variables are straightforward.

Rethinking yet again, the idea is sort of that each variable is sort of a function of other variables:

```
f = a * b + c
```

Or we could say it invokes those other variables as subroutines, and eventually it bottoms out in inputs (the dimensions). So the above statement is isomorphic to something similar to this:

```
def f(x, y, t):
    return a(x, y) * b(y) + c(x, t)
```

But when we quantify over a dimension (or, perhaps, even a dependent variable?) we are generating an argument locally rather than merely passing it along; similarly when we index, which is a form of composition!

```
f = c + +/[x] a * b
```

If `+/[x]` is summing along x, this decodes to something like the following:

```
def f(x, y, t):
    return sum(a(xi, y) * b(y) for xi in xs) + c(x, t)
```

The difference, of course, is that all this default parameterization is purely implicit.

This is closely analogous to dynamic scoping in Lisp.

Comparison to GNU MathProg and ZIMPL

GNU MathProg, also known as GMPL, and ZIMPL are two languages for defining models ("linear programs") for a linear optimizer to optimize. (This is called "linear programming". See Some notes on the landscape of linear optimization software and applications (p. 1285) for details.) They have essentially the same data model; in what follows I will use the MathProg terminology and syntax.

I wish I had read about these systems earlier, because much of what I describe above is already present in them. Unfortunately, I didn't learn anything about them until 2019.

Here is a slightly tweaked example of a MathProg model from the GLPK distribution, licensed under the GNU GPL version 2. It describes some kind of industrial metallurgy planning problem,

perhaps finding the cheapest way to mix two tonnes of an alloy within desired concentration limits from a combination of recycled scrap and fresh metal.

```
/* plan.mod */
var bin1, >= 0, <= 200;     var bin2, >= 0, <= 2500;
var bin3, >= 400, <= 800;   var alum, >= 0;
var silicon, >= 0;
param sival := .38;

minimize
value: .03 * bin1 + .08 * bin2 + .17 * bin3 + .21 * alum + sival * silicon;

subject to
yield: bin1 + bin2 + bin3 + alum + silicon = 2000;
fe: .15 * bin1 + .04 * bin2 + .02 * bin3 + .01 * alum + .03 * silicon <= 60;
cu: .03 * bin1 + .05 * bin2 + .08 * bin3 + .01 * alum <= 100;
mn: .02 * bin1 + .04 * bin2 + .01 * bin3 <= 40;
mg: .02 * bin1 + .03 * bin2 <= 30;
al: .70 * bin1 + .75 * bin2 + .80 * bin3 + .97 * alum >= 1500;
si: 250 <= .02 * bin1 + .06 * bin2 + .08 * bin3 +
    .01 * alum + .97 * silicon <= 300;
```

(`glpsol -m plan.mod -o plan.out` reports that the optimal values for the design variables are `bin1` ≈ 44.3, `bin2` ≈ 844, `bin3` ≈ 534, `alum` ≈ 421, and `silicon` ≈ 156, resulting in a `value` of $307.46.)

In the above example, all the variables and parameters are scalars, but MathProg also supports models incorporating aggregates. Here's another tweaked example from the GLPK distribution:

```
/* Chvatal V. (1980), Hard knapsack problems, Op. Res. 28, 1402-1411. */

param n > 0 integer;
param log2_n := log(n) / log(2);
param k := floor(log2_n);
param a{j in 1..n} := 2 ** (k + n + 1) + 2 ** (k + n + 1 - j) + 1;
param b := 0.5 * floor(sum{j in 1..n} a[j]);
var x{1..n} binary;
maximize obj: sum{j in 1..n} a[j] * x[j];
s.t. cap: sum{j in 1..n} a[j] * x[j] <= b;
data;
param n := 15;
```

In these languages, the expressed values — the values that expressions can evaluate to — are called "elemental values", and they can be strings (which are mostly treated as opaque atoms and are sometimes called "symbolic values"), integers, real numbers, logical or binary values, or sets of these, called "elemental sets". Their denoted values — the values that can be identified by names — include the expressed values, and also "model objects", which consist of "sets", "parameters", "variables", "constraints", and "objectives". All of the model objects other than objectives are "indexed" by n-tuples drawn from some "subscript domain" which is the Cartesian product of a possibly-empty sequence of sets.

The logical or binary values are *true* and *false*. The other kinds of

elemental values have their usual programming nature and support the usual operations, including string concatenation, substrings, transcendental functions, exponentiation, and set arithmetic.

Parameters and variables are essentially identical in structure; they are partial functions from their subscript domains to elemental values. The only difference is that when you run the optimizer, you choose the parameters, while the optimizer chooses the variables. There are rules limiting the structure of numerical expressions to ensure that they are linear in the variables; I will not be concerned with that here, but that is the reason for thus drawing the distinction at the language level rather than, for example, specifying it with the objective.

Confusingly, there are two related entities called "sets": the expressed value, called an "elemental set", and the model object, which is a denoted value. The difference is that the denoted value is indexed, like parameters and variables; it is a partial function from its subscript domain to elemental sets.

Constraints are partial functions from their subscript domains to Boolean values. The optimizer looks for a "feasible solution" which makes them everywhere true, and which maximizes or minimizes the objective among all feasible solutions.

The only iterative operations are `sum`, `prod`, `min`, and `max`, `forall` and `exists` for sets, the implicit `forall` on each constraint, and a set-comprehension operator on (elemental) sets that amounts to a Cartesian product filtered by a logical expression. All of these implicitly include an elementwise transformation function.

There is, I think, no way to compute an elemental set whose contents depend on the value of a variable, and the other kind of sets (the model objects) are specified as part of the model, so they can't depend on the value of a variable either. This serves the purpose of ensuring that MathProg and ZIMPL models can be "translated" into the formats MPS and CPLEX LP, which do not support any aggregate values at all, only affine equations and inequalities in real, integer, and logical variables.

Since (elemental) sets are the only aggregate values that expressions can evaluate to, expressions as such can only describe pointwise computations. Consider this line from the above example:

```
maximize obj: sum{j in 1..n} a[j] * x[j];
```

This defines an objective called `obj` which is to be maximized, defined as $\Sigma_j a_j x_j$. `1..n` is a range expression which produces a set of 15 integers from the integer parameter `n` = 15; `{j in 1..n}` declares the "dummy variable" `j` (confusingly, not related to the meaning of "dummy variable" in statistics) which is scoped to the "integrand" of `sum`, `a[j] * x[j]`, which indexes the real-valued parameter `a` and the binary-valued parameter `x` with the 1-tuple `j`, producing respectively a real and a binary value for a given value of `j`, which are then multiplied together; and finally `sum` produces a real number by summing 15 values computed from its "integrand".

This approach to aggregate operations is very different from the APL approach, in which you would write `+/a×x`, which is at least considerably fewer user interface operations. In the principled APL I'm thinking about, `+/` would need to somehow indicate which axis

(or dimension or index) it wants to sum along: `+/j a×x`, for example, if the axis were in fact called `j`, or `(sum (j) (* a x))` in Lisp S-expression syntax. However, for reasons described in A formal language for defining implicitly parameterized functions (p. 144), I think this approach will probably lead to variable-capture problems of the same kind encountered in Lisp macros, and so it's probably best to scope the variables bound by such iterative operations lexically; this leads to the formulation `+/j a[j=j]×x[j=j]`, which abbreviated to `+/j a[j]×x[j]` looks very similar to the MathProg approach. The biggest difference is that the limits of the sum come from the domain of the "integrand" rather than being explicitly specified.

(The indexing or reshaping operation `v[k=e]` is a special case: it binds variables dynamically rather than lexically, since otherwise it would not work.)

MathProg variables and parameters correspond roughly to the array variables described above, and indexes and sets are conflated above as "axes of variation" or "inputs". A big difference from APL is that in MathProg elementary sets are first-class values, while in APL axes are not any kind of value at all, except in the implicit sense that they are functions from your program's inputs to some dimension of some array. Also, sets in MathProg are unordered, while many operations in APL only make sense if some axes are ordered sequences of index values, notably prefix sum or scan \, encode ⊤ and decode ⊥ (the names come from their use for number base conversion), grade-up ⍋ and grade-down ⍒, reversal and rotation ⌽, take ↑, drop ↓, and catenate ,.

Topics

- Performance (p. 3621) (149 notes)
- Graphics (p. 3483) (91 notes)
- Programming languages (p. 3656) (47 notes)
- C (p. 3359) (28 notes)
- Python (p. 3671) (27 notes)
- Arrays (p. 3326) (17 notes)
- SIMD instructions (p. 3711) (10 notes)
- APL (p. 3320) (9 notes)
- Predicate logic (p. 3644) (6 notes)
- Types (p. 3758) (5 notes)
- GPGPU (p. 3479) (2 notes)
- Prefix sum

Can you make a vocoder simpler using CIC filters?

Kragen Javier Sitaker, 2017-06-28 (updated 2018-06-17) (2 minutes)

I was thinking about how to make a vocoder on the way to the office today, and it occurred to me that a fairly simple approach might be to use CIC filters.

Essentially the idea is that, for a single band, you use a moving-average filter (or more than one) over past samples to get a decimated (and thus low-pass-filtered) version of the signal; then you use a recurrent comb filter fed with, say, a subtraction between adjacent decimated samples, to estimate how much it's oscillating at harmonics of the comb filter's fundamental. Because the signal is decimated, you have very little frequency precision, which is what you want for vocoder band detection.

In particular, most human voice energy is between 64 Hz and 2048 Hz, which is a range of only five octaves; Wikipedia says, "There are usually between eight and 20 bands," so something like half-octave resolution is called for. Dudley's original 1939 Voder used 10 filter bands. I think it might be necessary to subtract amplitude signals from overlapping bands to get such high frequency precision with simple decimated signals.

Then you can use a similar process in reverse to apply the band coefficients to the carrier signal you want to vocode.

I'm thinking that it might be possible to get a working vocoder in about ten or twenty lines of C with this approach.

...having tried it in Numpy, I'm no longer as optimistic, although maybe there's a way. The issue is that you need really good attenuation in the stopband, and to be able to invert by subtracting, you need really, really precise zero-phase response — one milliradian of phase error is already a limit of -60dB on your stopband attenuation, and 10 milliradians is -40dB. I think it might still be possible.

Topics

Reduced affine arithmetic raytracer

Kragen Javier Sitaker, 2017-05-10 (1 minute)

I want to do a reduced-affine-arithmetic raytracer.

The idea is that an "image" is a function from pixel coordinates x and y to color component intensities r, g, and b, and we merely want to compute an adequate approximation to that function. We recursively subdivide the image into rectangular regions, and restrict ourselves to a linear approximation within each region, so that the overall approximation is piecewise linear (though not necessarily continuous between the pieces).

In this way, we can avoid spending much computation time on smooth gradient regions, concentrating on the regions where aliasing is possible.

Extending this, a "video" is a similar function, but has three independent variables: x, y, and t. This allows us to avoid spending computation time on parts of the scene that don't change much from frame to frame.

You can derive such an approximation by applying a self-validated arithmetic model from a mathematical description of the ray-traced scene. Most self-validated arithmetic models only give you zeroth-order approximations in any given region; interval arithmetic and the use of Lipschitz constants are examples. Affine arithmetic gives you a first-order approximation, but it is crushingly computationally expensive; reduced affine arithmetic, though it doesn't provide such tight bounds, is more efficient, and has been successfully used for raytracing.

Topics

Improving Lua #L with incremental prefix sum in the ∧ monoid

Kragen Javier Sitaker, 2018-12-18 (7 minutes)

Lua's #L operator, to find the length of a list, is implemented by a binary search through indices to find an index j such that L[j+1] is nil but L[j] is not. This is not constant-time and produces erratic results, but it allows the indexed assignment operation L[j] = nil to be, I think, amortized constant time.

The solution

A more predictable approach (or less euphemistically, an approach that always gives the right answer) would add a binary tree onto the side of tables and make L[j] = nil logarithmic-time like #L. The idea is simply that you maintain a set of user-invisible keys full(n, m) such that

$$\texttt{L[full(n, m)] == true} \iff \exists i \in \mathbb{Z} : i > 0 \wedge b^i = m \wedge m \mid n \wedge \forall j \in [n, n+m): \texttt{L[j] != nil}.$$

(We ignore the abomination of L[0] here.)

The base or branching factor b is a tradeoff between storage cost and efficiency; 2, 3, 4, 8, or 16 might be reasonable choices. The node size m is a power b^i of b, and its start index n is a multiple of m.

Algorithms in more detail, with asymptotic costs

When you set an index L[j] to nil, you need to look for full($\lfloor j/b^i \rfloor$, b^i) nodes that contained that index, deleting them and increasing i until you don't find one; at worst this takes $\lceil \log(|L|)/\log(b) \rceil$ iterations, where $|L|$ is the number of items stored (at integer indices) in L; so, for example, deleting an item from a list of up to 16 777 215 items and b = 4 might require up to 6 iterations.

Conversely, when you set an index L[j] that was previously nil to a non-nil value, you may need to create up to a logarithmic number of full($\lfloor j/b^i \rfloor$, b^i) nodes; each node that is created requires verifying the existence of b-1 existing child nodes that are siblings of the node just created, treating indices L[k] as if they were full(k, 0) nodes. In our example, this potentially involves creating or examining 4 nodes at each of 6 levels, a total of 24 iterations.

These are worst-case numbers; in most cases, deleting a numeric-index table entry would not require the deletion of any full(n, m) nodes, and adding one would not require the creation of any full(n, m) nodes, and for typical usage patterns they would be amortized constant time, just like the current implementation. But it would be easy for a program to provoke the worst-case logarithmic

slowdown: create a large list and repeatedly delete and recreate its first element.

Computing #L would follow a logarithmic-time process similar to the present process, first walking up through the full(0, b^i) items until it found the largest one, then walking back down looking for successor nodes to last children. In the worst case, this requires examining b full(n, m) nodes at each of ⌈log(|L|)/log(b)⌉ levels; our example list of 16 777 215 items would require examining 24 full(n, m) nodes, just like insertion. The difference from the current implementation is that the answer it gives is precisely the index of the first non-nil item whose successor is nil.

The storage overhead is up to |L|/(b–1) invisible table items that do not exist in the current implementation.

This is precisely the parallel prefix-sum algorithm used for incremental rather than parallel computation in the ∧ monoid. Using the ∨ monoid gives an alternative definition which could be computed in a similarly efficient way, also complies with the current definition, and is more similar to the definitions in sister languages like JS and Perl, would return the index of the *last* non-nil item (whose successor is therefore nil). I think this is wrong for Lua, because it's unnecessarily incompatible with the behavior of `ipairs`.

Improving constant factors

As a locality and space optimization, it might be desirable to store the full(n, m) items in a way that somehow tacks them on to L[n+m-1] rather than as separate table items, sort of like a skip list. For example, you could store them in a bitmask indexed by log(m)/log(b). (This allows the initial examination of the full(0, b^i) items in #L to be done with a single instruction on some CPUs.) Alternatively, you could store them separately as ⌈log(|L|)/log(b)⌉ bitvectors of logarithmically decreasing numbers of bits, but that seems like it could be complicated if you start storing things at high indices.

As a complicated optimization, you could lump these bits into lumps big enough to amortize storage overhead; taking again b = 4, in a 128-bit lump of level j, you could have 64 nodes of full(n, b^{5j+1}), 16 nodes of full(n, b^{5j+2}), 8 nodes of full(n, b^{5j+3}), 4 nodes of full(n, b^{5j+4}), and 1 node of full(n, b^{5j+5}), using 93 of the 128 bits. You need one such level-0 lump for each run of 1024 list items, a level-1 lump for each run of 1 048 576, a level-2 lump for each run of 1 073 741 824, and so on; and you need zero such lumps until you have a run of 4 items that's properly aligned to generate a lump. This limits the worst-case space overhead to 25%, if the lumps are the same size as normal table entries.

In the case of b > 2, this storage optimization would also allow the use of simple bit masking operations to simultaneously test for the existence of all children. But, for insertion and deletion, this mostly helps with the worst case, because in the average case, you're overwhelmingly testing membership in the table itself.

The b tradeoff

Choosing b to be larger makes assertion of new values and computation of length almost proportionally slower (O(N/log(N))), but also reduces storage overhead by b–1 and makes deletion

logarithmically faster (O(1/log(N))).

Since deletion is much less common than insertion, it would be nice if there was a way to shift that factor of N over to the deletion algorithm. I haven't found one. You might try the simple approach of making nodes at a given level conditional on the existence of not just all of their children but all of their previous siblings. Thus insertion creating a new node at a given level need only check for the existence of its previous sibling to decide whether to create its "parent", but deletion potentially needs to delete all of its following siblings. But insertion at the beginning retains the same worst case, and the usual case is amortized constant time, just as before. So I think this doesn't really help.

Topics

Trellis-coded buttons to run a whole keyboard off two microcontroller pins

Kragen Javier Sitaker, 2013-05-17 (updated 2019-06-13) (30 minutes)

(I think this was published previously on kragen-tol, but this version has been improved somewhat.)

Based on some discussions with Nick Johnson, I was thinking about a "downloadable tinkerer's tricorder", software that would turn a commodity microcontroller into a powerful LCR meter, and an additional application occurred to me: a keyboard with up to several dozen keys, supporting N-key rollover ("NKRO"), using only two microcontroller pins, one with ADC and the other with digital output, or even a single pin, and a couple of dirt-cheap passive components per key, with latency of a few milliseconds or even down into the submillisecond range.

This represents an order-of-magnitude reduction in cost for NKRO keyboards, which are essential for some kinds of video games and for advanced chording input methods.

Background: relative cost of different parts

Pins on chips are among the most expensive resource in modern circuitry. A small diode, resistor, or capacitor, with no special requirements, will cost between a sixth of a cent and a whole cent (diodes are the expensive items here), but going from an 8-pin chip to a 14-pin chip will cost you at least a dollar; each pin, in effect, costs as much as 20 to 100 resistors! And adding another 8-pin chip will cost you about 50 cents, but at least three of those pins will be tied up with power and communication, costing you ten cents per pin, or more.

Pushbutton switches cost nearly 10 cents if you buy them as separate components, but they're cheaper if you make an assembly with many of them built into a single component. As a result, if you dedicate a microcontroller pin to each pushbutton, you nearly double the cost of the pushbutton, and maybe much more if your keyswitches are really cheap.

More than one key per pin

So, in addition to the standard matrix multiplexing approach, designers occasionally hang several pushbutton switches off one I/O pin on a microcontroller, each controlling a different resistance. Then you can use a voltage divider so that each pushbutton creates a separate voltage, which you can measure with an analog-to-digital converter.

It sounds crazy to use an ADC and some resistors to save an I/O pin or two, but the fact is, lots of microcontrollers have one or more ADCs built in already, and you can time-share it between different uses on different pins. So it doesn't cost you anything extra.

The most famous design that does this is probably Limor Fried's Monochron. A variant that avoids the need for an analog-to-digital converter is the PaperTecladoRC, which charges up a capacitor

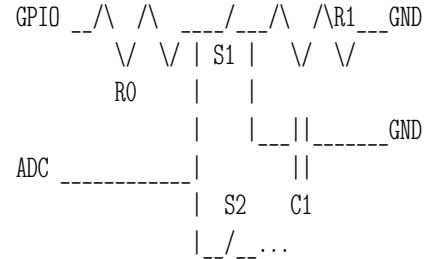
("condensador") and then measures its time to discharge to logic zero through the resistor network, rather than making an analog voltage measurement.

(In the standard matrix multiplexing approach, which is how basically all but the most expensive keyboards work on everything from pocket calculators to Macbook Airs, you make an N×M matrix where each keyswitch connects one of the N rows to one of the M columns, and you alternately energize each row to see which columns get energized. You can get two-key rollover with this approach, but not NKRO, and you still need a substantial number of I/O pins, typically around 20.)

Trellis-coding your keys to get N-key rollover on a huge number of keys

It occurred to me that you can do much better than this. Rather than merely discriminating between different *resistances* attached to each button, you could discriminate between different *complex impedances*: connect each button through not only a resistor but also a capacitor. Now we can subject the pin to a time-varying signal, such as a transient pulse, and watch its response.

To consider one possible concrete realization of this approach, consider this layout:



Your GPIO pin, a switchable voltage source, connects through a resistor R0 to a common bus to which all the pushbuttons connect. Your ADC is also connected to this common bus. Each pushbutton Si connects the bus to a resistor Ri and a capacitor Ci, each of which is grounded on the other side; that is to say, they're in parallel.

This is more or less the configuration of a *single* AVR I/O pin in input mode: you can either connect the pullup resistor R0, which is an imprecise and nonlinear resistance in the 20kΩ–50kΩ to VCC, or leave it in a high-impedance mode where it won't source or sink more than a few microamps. However, it's convenient to us to use *two* pins, both because it lets us use a higher-resistance pullup resistor and correspondingly lower-capacitance capacitors, and because it makes our pullup resistor much more precise.

Suppose one of the pushbutton switches S1..SN is pressed; let's say S1. Now, if we bring the pullup high, the voltage measured by the ADC will rise eventually to VCC R1/(R0+R1). If the various Ri are sufficiently different, we can use the ADC to distinguish which of them it was. Indeed, if they're sufficiently different that the sum of each subset of their conductances is sufficiently different to be distinguished by the ADC, we can distinguish *any subset* of them, which is to say that our keyboard has "N-Key Rollover", or NKRO, an advanced gaming keyboard selling point. (According to the

documentation for Plover, a Stenotype input method that can get 250 words per minute on a regular NKRO keyboard, you can't find a full-size NKRO keyboard for less than US$50.)

If you have a 10-bit ADC, you could conceivably distinguish 1023 different keys merely by their resistances by this method, but you can't distinguish more than 10 of them if you want to be able to distinguish all of the possible chords (that is, get NKRO). Also, your resistances need to really be distinct, which is difficult: we're talking about 1023 different resistances, with better than 0.1% tolerances here for the highest resistances, which is hard to find. (From browsing Digi-key, it seems like regular sixth-of-a-cent resistors these days are ±1%; regular antediluvian resistors are ±20%; ±0.5% resistors cost half a cent; the resistance of the same resistor normally varies by more than 0.1% with temperature.)

Worse, the resistances need to be precisely aligned with the bit transitions of the ADC, and the top one needs to have 1022 times higher resistance than the pullup. So in practice I think you'd be lucky to distinguish 8 different keys on resistance alone with NKRO unless you trim each resistor after assembly.

But that's what the capacitances are for! If you have two different keys with the same resistance to ground, but different capacitances, they'll both eventually arrive at the same steady-state voltage — but they'll take different amounts of time to get there. Better yet, you can measure the charging speed with greater precision than you can the final voltage, and taking multiple measurements as the capacitor charges can give you better than 10 bits of precision by averaging over time.

(In effect, you're using trellis-coding: the complex impedances of each key at a given arbitrary frequency are a sort of exponentially distributed lattice in one quadrant of the complex plane. Although we're actually adding the *reciprocals* of the impedances (we're adding the complex analogues of conductance; maybe there's a word for this), those are also exponentially distributed such that every subset of them has a unique complex sum, and all of these complex sums are far enough apart that we can distinguish them despite measurement error.)

Suppose tracing the curve over time gives us 12 bits of precision on the resistance (by averaging 16 samples) and 12 bits of precision on the RC time constant. Intuitively I think it should be easy, then, to distinguish 6 different resistances and 9 different capacitances, for a total of 54 keys with NKRO, on a single ADC pin!

To be slightly more precise, we probably want the largest resistance value to be somewhat larger than the pullup resistor value. Suppose we choose 220kΩ for our pullup. We want the smallest resistance value to be less than the error in the largest; and we want all the sums of resistance values to differ by more than the sums of their errors. This suggests using resistances of 500kΩ, 220kΩ, 100kΩ, 50kΩ, 22kΩ, and 10kΩ; if they, improbably, all had an error of 1%, that would be 5kΩ+2.2kΩ+1kΩ+500Ω+220Ω+100Ω = 9020Ω, which is less than 1kΩ. A seven-bit ADC measurement would probably be enough to distinguish between them, and 10 bits definitely will. IIRC, experiments have shown that the nominally 10-bit successive-approximation ADC on the ATMega328 and friends can still give you an effective number of bits ("ENOB") of 7 at about

1Msps, which is to say a microsecond per measurement. In the "worst case", you'd need about 11 bits' worth, which I think is about 600 microseconds (four successive measurements) on the ATMegas.

(I'm a little fuzzy on the exact ADC numbers, unfortunately, but I think they may actually be significantly better than the above.)

Then we need to do the same trick for the capacitors. The smallest capacitance we can reasonably measure will be one that charges fully through the pullup in a single 600-µs measurement, so the RC time constant should be around, I don't know, 50µs. If the pullup is 220kΩ, that puts it at 227pF (say 220pF for practicality). Then you could use nine capacitances of 220pF, 470pF, 1000pF; 2200pF, 4700pF, 10000pF; and .022µF, .047µF, and 0.1µF.

The slowest RC constant, then, should be 0.1µF × 220kΩ, or 22 milliseconds. But since you're using an ADC, you don't need to wait for the capacitor to charge fully, just enough that you can accurately measure its rate of charging — say, ten samples or ten out of 1024 ADC counts, whichever is faster. In the worst case, the 0.1µF capacitor will be in parallel with a 10kΩ resistor, which means its final charged voltage would be 10/(220+10) of the 1023-count max: 45 counts. Charging 10/45 of the way will then take 1.25 time constants, or 28 milliseconds.

That's only marginally acceptable, so it's probably worthwhile to blacklist the one, three, or six slowest combinations. The two next-worst are 0.1µF in parallel with a 22kΩ resistor, which will have the same RC constant, but the final voltage is almost twice as high, so you'll rise by 10 counts in only 14 milliseconds; and 0.047µF in parallel with 10kΩ, which will have less than half the RC time constant and the same asymptotic voltage, so will also need only 13 milliseconds. The next three worst are 0.1µF with 47kΩ, .047µF with 22kΩ, and .022µF with 10kΩ, all of which need about 7ms. So if you eliminate these six keys, you can measure any keychord in under 4ms.

(Another way to improve this: on the AVRs, there's the possibility of using an internal 1.1V bandgap voltage reference instead of VCC. At a typical 5 volts, this will increase the precision of the measurement of this initial voltage rise by a factor of almost five, and thus decrease that 28ms to some 6ms.)

So that gives you a 48-key NKRO keyboard with worst-case 4ms latency for the cost of two microcontroller pins (20¢), 49 1% resistors of six values (10¢), 48 1% ceramic capacitors of nine values (25¢†), and 48 keyswitches (480¢ if discrete, much less if made as one piece, much more if you use Cherry high-end mechanical keyswitches) for a total of some US$5.35, or US$0.55 as the cost of the keyswitches themselves approaches zero.

This is an order of magnitude cheaper than the US$50 cost of existing NKRO keyboards, or two orders of magnitude cheaper if the keyswitches are free.

† I'm actually having a hard time finding cheap capacitors with reasonably precise capacitances. This could limit the effectiveness of the idea — but see the section at the end for how to fix this.

Information theory shows it won't work quite that well

There's some kind of problem with my calculations, though. You need 48 bits of entropy to get NKRO on a 48-key keyboard. If you have an estimate of the RC time constant with a precision of 50µs with an upper limit of 3ms, as I've postulated above, that's almost 6 bits of entropy. That, combined with a 12-bit measurement of voltage at some known point in the charging curve (the result of averaging some 16 samples) gives you 18 bits. That's less than half of what you need. In the absence of nonlinear components such as diodes, those two parameters will be able to predict any further measurements down to the limit of measurement noise. So at best you can get 18-key rollover — and that's ignoring that many of the combinations of final voltage and RC time constant are outside of the feasible region!

In practice, I think that given 18 bits of data like that, you can probably identify a unique RC time constant and asymptotic voltage for each key, even if the nominal values of the resistors and capacitors were equal. (Precision of ±1% steals the top 6 bits of each parameter, leaving 6; using lower-quality, more-variable components would help by adding more randomness.) If you train the microcontroller for a given keyboard, storing calibration constants for each key and then adjusting them with a linear correction for an estimated temperature (assumed to be constant across the keyboard), you could probably reliably distinguish several thousand keys — but without NKRO.

Ways to improve performance

If the problem is that we only get 18 bits of entropy and we need 48, here are some approaches.

The most glaring problem is that we need at least 9 bits of precision on the capacitance measurement, but our hypothesis above is that we're only getting about 6 bits of RC time constant, which represents the contribution of C. Maybe my calculation is off: maybe from a series of 10-bit voltage measurements, even over only a maximum of some 60 sample times, we can extract some 9, 10, or even 11 bits of precision for RC. Maybe use a least-squares fit in some kind of transformed space.

But that only gets us to, like, 23 bits, when we need 48! We need more entropy! (Even an ADM-3A had 59 keys.) Unless we just want a Stenotype, which has exactly 23 keys.

A second approach, as mentioned earlier, is to incorporate nonlinearity. For example, if you add a diode somewhere in the network for each switch, your RC time constant changes with voltage, according to the diode's characteristics. This might provide another 12 bits of entropy, if you have diodes with enough variability whose voltage drop is comparable to that of the resistor at comparable currents. By itself, that could get us to maybe 30 bits, at a cost of another 24¢ at half a cent per each of 48 keys.

A third approach is to use another ADC pin and pullup resistor from the same GPIO pin, giving you, in essence, a second separate keyboard, at a cost of some 10¢. This approach scales linearly to as many ADC pins as you have available on your microcontroller, at some cost to sampling rate per pin and therefore necessitating somewhat larger capacitors. If this is the only approach you take to improve performance over the basic approach, so that you can only handle about 18 keys per ADC pin, then one GPIO pin and three

ADC pins (40¢ of pins) should get you to 54 keys, or maybe a bit more if you push it. That's enough for a full compact keyboard, at a cost of 75¢ (55¢ - 20¢ + 40¢) of electronics, plus the keyswitches.

If we combined all three of the above approaches, you'd be getting 35 bits on each of three input pins, for 105 bits in all: enough for a full keyboard with keypad, at a cost of, I don't know, a dollar or two of electronics, plus the keyswitches.

However, there's also a solution that works with *no extra hardware*! Since we're able to scan the entire keyboard at 250Hz, it's really completely unnecessary to consider all possible new keyboard states as equally likely. An actual human being won't be able to press and release more than one or two keys in any 4ms interval. So of all the candidate new keyboard states, we can take the one with the smallest hamming distance from the current keyboard state, and we'll essentially always be correct.

(That insight is from reading posts by Talkingjazz about their rotary switch decoding problem: http://www.arduino.cc/forum/index.php/topic,20125.0.html.)

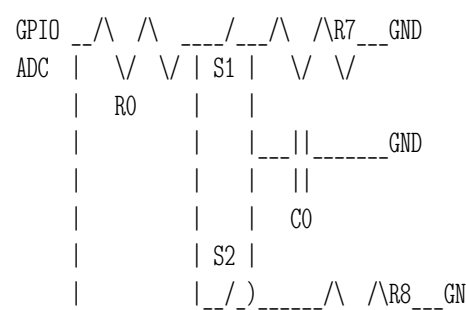
Yeah, or use a shift register like a normal person

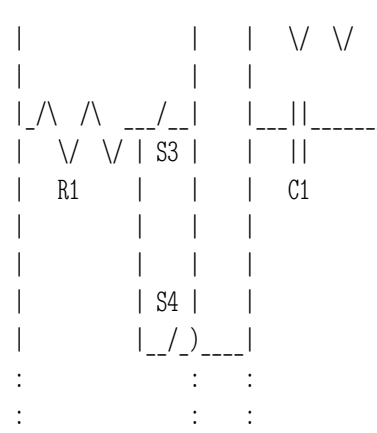
You could use a ten-cent 74HC165 shift registers and eight pullup resistors for each set of eight keyswitches, and chain them all together in a long line feeding into one pin on your microcontroller. You need two more microcontroller pins to clock the shift registers and to enable them. Cost for 48 keys: 30¢ for the microcontroller pins, 60¢ for the shift registers, 24¢ for the pullup resistors, total of US$1.14, plus the keyswitches themselves. This is maybe more expensive than the trellis-coding approach, but it sure is a lot simpler.

The improved single-pin trellis-coding approach

Here's a better version. In the simplified form I analyze here, it doesn't support NKRO, but although I haven't analyzed the more-complex NKRO variant, I anticipate that hacking NKRO in will be feasible.

The entire keyboard has two connections: the probe/power/sense connection and ground. Each (logical) row is connected to the sense connection through a different-value resistor; each column is connected to ground through a different-value resistor in parallel with a same-value capacitor. The microcontroller charges the sense connection for different periods of time and then switches its pin to analog input mode to measure the decay curve, then repeats the process.





(Here in this diagram we have two rows and two columns; the first row charges through R0 and contains S1 and S2, while the second row charges through R1 and contains S3 and S4. The first column charges C0 through S1 or S3 and discharges it through C7, while the second column charges C1 through S2 or S4 and discharges it through R8.)

Let's be concrete. Consider a US$1.30 48MHz STM32F031C4 with its 1Msps 12-bit ADC, running at 3.3 volts. (See Notes on the STM32 microcontroller family (p. 3176) for more details.) It's driving a 7×7 keyboard, which is missing 12 of its potentially 49 keys in the corners, leaving 37 keys; row 0 has 4 keys (missing columns 4, 5, and 6), row 1 has 5 keys (missing columns 5 and 6), row 2 has 6 keys (missing column 6), row 4 has 6 keys (missing column 0), row 5 has 5 keys (missing columns 0 and 1), and row 6 has 4 keys (missing columns 0, 1, and 2.) All the capacitors are 0.001-μF MLCCs. The resistors for row and column 0 are 1 kΩ; for row and column 1, 2.2 kΩ; for row and column 2, 4.7 kΩ; for row and column 3, 10 kΩ; and thus 22 kΩ, 47 kΩ, and 100 kΩ for the remaining three rows and columns.

Column 0 discharges with a time constant of 1 μs; column 2 discharges with a time constant of 2.2 μs; and so on, until column 6 discharges with a time constant of 100 μs. This is true regardless of which row it was charged by. During the discharge cycle, the only voltage across the row resistor comes from the I/O pin's input leakage current, specified in the datasheet as ±0.2 μA, thus producing an error of ±20 mV across row 6's 100 kΩ resistor, and proportionally less on the other rows, so the microcontroller can sense the column voltage with quite adequate precision. That is, the discharge time constants vary like this across the keys:

```
[  1.    2.2   4.7   10.                      ]
[  1.    2.2   4.7   10.    22.               ]
[  1.    2.2   4.7   10.    22.    47.        ]
[  1.    2.2   4.7   10.    22.    47.   100. ]  μs
[        2.2   4.7   10.    22.    47.   100. ]
[              4.7   10.    22.    47.   100. ]
[                    10.    22.    47.   100. ]
```

The timing precision needed to distinguish these columns is thus only about a factor of 2.

The voltage from which the discharge curve is falling depends on the charging time, asymptotically approaching the voltage in the

middle of the voltage divider formed by the row and column resistors. So it ranges from 3.00 volts for r0c3, r1c4, r2c5, and r3c6 to 0.30 volts for r3c0, r4c1, r5c2, and r6c3. The other voltages, though they vary slightly, are around 0.60 volts, 1.06 volts, 1.65 volts, 2.27 volts, and 2.72 volts. Note that errors in the capacitance do not affect these asymptotic voltages at all. Here's the full table of asymptotic voltages for each key:

```
[ 1.65  2.27  2.72  3.                    ]
[ 1.03  1.65  2.25  2.7   3.              ]
[ 0.58  1.05  1.65  2.24  2.72  3.        ]
[ 0.3   0.6   1.06  1.65  2.27  2.72  3. ] V
[       0.3   0.58  1.03  1.65  2.25  2.7 ]
[             0.3   0.58  1.05  1.65  2.24]
[                   0.3   0.6   1.06  1.65]
```

However, a third parameter also distinguishes the rows: the charging time constant. If the charging time is short enough, the voltage will not quite reach the asymptotic voltage described above, which can be ascertained by doing multiple measurement cycles with different charging times. The charging time constant is the RC product of the 1-nF column capacitor and the parallel combination of the column discharge capacitor and the row charge capacitor. The microcontroller cannot directly observe the charging curve, since during the charge cycle it sees 3.3 volts on its I/O pin (an unknown fraction of which is dropped across the row charging resistor), although perhaps it could pause the charging periodically to take a sample.

The charging time constant ranges from 0.5 µs (r0c0, each with a 1-kΩ resistor) to 50 µs (r6c6, each with a 100-kΩ resistor). The whole array of charging time constants is as follows:

```
[  0.5    0.69   0.82   0.91                        ]
[  0.69   1.1    1.5    1.8    2.                   ]
[  0.82   1.5    2.35   3.2    3.87                 ]
[  0.91   1.8    3.2    5.     6.88   8.25          ] µs
[         2.     3.87   6.88   11.    14.99  18.03]
[                4.27   8.25   14.99  23.5   31.97]
[                       9.09   18.03  31.97  50.  ]
```

Note that the contours of this time-constant table run nearly at right angles to the contours of the asymptotically-approached voltage; that is, where the asymptotic voltage provides the least information, the charging time constant provides the most information. Avoiding zones where this charging-time-constant information is weakest is the reason for omitting the corners where the resistance values are too far apart.

Since these charging time constants are using the same capacitance as the discharge time constants, over the same range of voltages even, errors in the capacitance will affect them both by the same factor. However, if we treat the capacitance as entirely unknown, we entirely lose the information about the magnitude of the resistances — the ratios of the charge and discharge times follow the same diagonal pattern as the asymptotic voltages.

You *could* recover some of that resistor-magnitude information by

doing a charge cycle through on-chip pullup resistors, as suggested for the earlier-outlined design, or the pulldown resistors the STM32 also has (making it a sort of discharge cycle instead — you'd want things to be charged first). The pullup and pulldown resistors are not very precise, and their values probably vary widely with the chip temperature, since they're presumably polysilicon and not, you know, metal film or carbon or something. But they might be good enough to help you compensate for capacitor errors.

So this design gives you 37 keys for the cost of one microcontroller pin (which I said earlier was 10¢), 14 resistors of 7 values (2.3¢), 7 ceramic capacitors (3¢), and 37 keyswitches (370¢ if discrete, much less if made as one piece, much more if you use Cherry high-end mechanical keyswitches) for a total of some US$3.85, or US$0.15 as the cost of the keyswitches themselves approaches zero — two thirds of which is the imputed cost of the microcontroller pin! None of the components need to have a precision of better than 10%, but it's particularly insensitive to the values of the capacitors, which could easily have capacitance errors of a factor of 2 or 3 without doing much harm. And that's good, because (as I mentioned earlier but didn't fully appreciate at the time) ceramic capacitors, which are the cheap ones, have pretty imprecise capacitances which vary a lot by voltage.

Earlier I priced the pin at 10¢, but perhaps on an STM32 I should lower that cost; the STM32F031x4 has 39 GPIOs and only costs US$1.30, so maybe its GPIO pins only cost 3¢. On the other hand, it wouldn't be surprising if you had to dedicate the entire STM32 computer to the keyboard-processing task.

How much measurement error are we going to have on these charge and discharge curves? Let's suppose the ADC is using the µC's internal bandgap reference, which is specified as 1.2–1.25 V. This gives us plenty of sensitivity for even the 0.3-volt signals: those start out as a third of full scale, 983 LSBs. The ADC's total unadjusted error is specified as ±4 LSBs, or ±1.22 mV.

The hardest measurement to take might be a 3.0-volt signal with a 100-µs discharge time constant — if we let it charge fully, it'll be out of range for the first 92 samples. (I'm assuming that'll just give us a full-scale reading, not a blown chip. Also I'm ignoring the issue of charging the 8-pF sample-and-hold capacitor in the ADC's frontend, which could add a significant error to quickly-changing signals when the input impedance is high.)

So we have about ±2% error from the bandgap reference, ±0.1% error from the ADC, ±1% from the resistors, and probably ×/÷ 2 from the capacitors — an error I'm optimistic we can calibrate out.

Every millisecond of measuring the charge/discharge waveforms gives us close to 1000 samples of data. In theory, two samples are enough to fit an exponential decay toward a known zero voltage, and three samples are enough to fit an exponential increase towards an unknown asymptotic voltage, so these 1000 samples should give us about 23 dB of noise immunity — an ENOB increase of 4, giving us effectively 16-bit precision on the measurements of the three components we're trying to measure.

This suggests to me that 37 keys is really aiming low, and we could potentially do orders of magnitude more.

What about NKRO? As described, the circuit topology has a

really serious key-ghosting problem: there's no way to distinguish the sets of keys {r2c2, r4c2, r4c4} from {r2c2, r2c4, r4c4} because they both connect exactly the same sets of wires together. The simplest solution to that aspect of the problem is to put a different precision resistor in series with each keyswitch. Additional measures that might help include adding capacitors to ground from the row lines, providing a pair of capacitances to come into equilibrium through the keyswitch resistor during the discharge cycle, and giving every component a slightly perturbed value, rather than having many resistors and many capacitors of the same value, which produces ambiguity.

Topics

Memoize the stack

Kragen Javier Sitaker, 2015-09-03 (5 minutes)

Memoization is a fairly general technique for making pure functions faster to compute, particularly in the presence of incremental changes to input data, but it usually has to be applied judiciously. The memoization runtime overhead on a function call is fairly large compared to a primitive function call (a bit larger than an order of magnitude), the space overhead of the memoized arguments and return values is potentially many orders of magnitude larger than that of the original program. To make things more complicated, some functions are worth memoizing because, although they run for less than a microsecond, they are called many thousands of times per second, while others are worth memoizing because, although they are only called a few times per second, they run for tens of milliseconds; and, furthermore, if a function A does most of its work in a function B, and function B is mostly called by function A, it may be the case that either A or B is worth memoizing, but not both A and B, since memoizing either of them will dramatically reduce the cost of the other. Finally, two functions C and D, which would seem to be equally worth memoizing given their respective runtime costs, might have vastly different memory costs to memoize, and so we might vastly prefer to memoize the one that will use less space.

In the face of all of this uncertainty, I propose that perhaps a simple and reasonably effective way to figure out what to memoize may be to scan the stack at every minor garbage collection (a task which cannot be omitted from garbage collection in any case) and note the particular activation records currently present upon it. If our activation records are in fact allocated on the heap in the nursery, then GCs almost cannot fail to occur regularly during ordinary computation, so the probability for an activation record to be present at garbage-collection time is a good approximation of how much computation time it represents. If we then evacuate these activation records to the next generation, they will then be safe from minor garbage collection, and furthermore if we mutate them by writing a pointer to their eventual return value into them, the write barrier will shanghai the return value from the nursery into their generation upon the next minor collection.

This, then, should provide us with a reasonably good facsimile of a memoization policy that retains in memory the activation records of calls whose past results we are likely to wish to consult once more. Still, it cannot yet inform us of which *functions* we ought to instrument with a memo-table probe, nor does it of itself organize them into a queryable memo table. After all, even if dozens of activation records for a given function are copied out of the nursery, that function might be called dozens of times with different arguments, or dozens of millions; in the second case it may not be worth patching memoization overhead into the function's preamble!

But let's suppose that we come up with some kind of approach to solving these problems, like the invocation counter HotSpot uses to decide which methods to optimize harder, sweeping the second generation to collect saved activation records into a table when it's

time to collect it, and estimating the amount of otherwise-garbage that each activation record hauls out of the nursery, or something.

A really strange benefit of this memoization mechanism is that it can possibly memoize functions whose return value turned out to be very expensive to compute even from the very first time they are invoked, with no ahead-of-time indication that they will be expensive. As long as they lived long enough and survived enough nursery collections, their return value will be properly saved, and future invocations will be able to use it.

With a sufficiently powerful and clever memoization mechanism, you could replace most or all intermediate data structures in a program with function calls that purported to compute a value from the original (externally provided) inputs, and trust that if those functions take a long enough time to run, then their return values will be stored in a hash table without any explicit intervention on your part. The best thing about that is that you wouldn't have to worry about when to update those intermediate data structures or how much of them to update. This is probably kind of a fantasy, though.

Topics

Rasterizing polies

Kragen Javier Sitaker, 2017-07-19 (3 minutes)

I was thinking about the problem of rasterizing a set of polylines filled with colors; for example, for rendering text from an outline font after some arbitrary geometric transformation. It occurred to me that it's probably reasonable to approximate each smooth segment of the polyline with a quadratic or cubic spline providing a fractional X-coordinate given the Y-coordinate; this explicit spline may need to have more knots than the parametric spline defining the original polyline in order to achieve adequate precision.

Note that computing the X-coordinate as a function of the Y-coordinate is transposed from the usual convention for graphing functions, established I suppose by Descartes from, probably, the Greeks' convention of writing their letters from left to right.

It should be possible to use interval arithmetic to conservatively approximate how precisely you need to approximate the parametric spline with the explicit spline.

Anyway, once you have a bunch of explicit splines, the inner loop of updating all of the X-coordinates for a new raster is just this:

```
x += Δx;
Δx += ΔΔx;
// and if they're cubic splines:
ΔΔx += ΔΔΔx;
```

These vector-addition operations can be carried out in SIMD or SPMD fashion, which can be up to 512 bits per instruction on modern computing hardware.

When we encounter a knot in the spline, we just need to update an element of the ΔΔx or ΔΔΔx vector with the new value. If we instead encounter a corner in the polyline, we also update Δx. A top edge, corner, or curve involves adding two new items; a bottom one involves removing them.

This gives us a vector of x-coordinates; if we sort that vector, we get partitions dividing the scan line into regions alternating inside and outside. (It may be worthwhile to keep all of the vectors sorted, since most of the time the order won't change from one scan line to the next.)

To keep the knots at least 16 scanlines apart without the errors getting over some size, I think we need an extra 4 fraction bits per order of polynomial; that is, for quadratic splines, we need 8 more fraction bits for ΔΔx than we would need for our desired precision of x, and for cubic splines, we need 12 more fraction bits for ΔΔΔx. This probably means that 32-bit fixed point with 16 fraction bits is adequate for most purposes, providing 16 levels of antialiasing with cubic splines, while 16-bit fixed or floating point is not, which means that a 512-bit vector is only 16 elements. But that's still enough to give a dramatic speedup.

Topics

Forth looping

Kragen Javier Sitaker, 2007 to 2009 (16 minutes)

Warning! None of the code I wrote for this post is tested, so it probably has bugs.

Warning! This message is almost entirely concerned with micro-optimizations for obsolete languages, obsolete CPUs, and CPUs that never shipped, and I don't even know very much about them.

Forth Definite Looping Constructs

I remember reading some years ago about how Chuck Moore tries to avoid use of the "DO...LOOP" construct these days. In http://www.ultratechnology.com/moore4th.htm he says:

```
Some of the people who don't like Forth might take to this. In 20
blocks of code I have no conditional statements or loops.  Good
Forth minimizes the number of conditional statements. The minimum
is zero.  I can say

: 5X X X X X X ;  : 20X 5X 5X 5X 5X ;

This is just as good as a loop. When running through memory the
code should compare an address to terminate rather than use a loop
count.
```

(Attributed to Chuck Moore's 1997 Silicon Valley FIG presentation: http://www.ultratechnology.com/color4th.html)

This struck me as crazy, since it makes the program considerably more opaque.

(In http://www.ultratechnology.com/1xforth.htm from 1999 he says he prefers tail-recursion to other looping constructs, and much of that is actually on display in the 1997 presentation as well.)

Moore has been working on top of his stack-based MuP21 and F21 CPUs since the early 90s, or at least pretending to by emulating them with assembler macros on Pentiums, which run a lot faster. These chips implement something like Forth operations as their native instruction set. But the run-time semantics of LOOP are not among their primitive operations; instead there are JZ (T0), which jumps if the top-of-stack is zero (excluding its carry bit), and JCZ (C0), which jumps if the carry bit on the top-of-stack is zero. (On these chips, each register on the stack drags along its own carry bit.)

I've been implementing a toy Forth system myself this week, and I got to LOOP, and you know, it's a little bit complicated. My implementation was something like 19 words in length. In view of Moore's overriding desire to get stuff done with a minimal amount of resources, his avoidance of LOOP no longer seems quite so incomprehensible.

So first I'll explain what I wrote, and then I'll show the much better solution from F-83.

What I Wrote

```
    defbytes _loop          # twiddles return address
    # ( -- ) ( R: X limit-1 limit -- X+1 ) in end-of-loop case
    # ( -- ) ( R: X counter limit -- X-N counter+1 limit ) in normal case
    # N is stored at X (in the instruction stream after
    # the call to _loop)
    # XXX we need a way to sign-extend characters; it's
    # confusing to write a positive number for a backward
    # jump.
    ## Also, this is way too long.
    .byte b_rpop, b_rpop, b_add1, b_rpop, b_twodup, b_xor
    .byte b_branch_on_0,7, b_rpush, b_rpush # not done yet, so jump back
    .byte                  b_dup, b_c_at, b_sub, b_rpush, b_exit
    .byte b_twodrop, b_add1, b_rpush, b_exit # done
```

That's hopefully equivalent to this Forth:

```
: _loop r> r> 1+ r> 2dup xor
    if >r >r dup c@ - >r exit then
    2drop 1+ >r ;
```

So it pulls its own return address off the return stack, pulls off the loop counter and increments it, pulls off the limit, compares the incremented loop counter to the limit, and if they're still not equal, it pushes them both back onto the stack, then fetches a jump offset from where its return address points and subtracts that from the return address before sticking it back onto the stack, then returns. But if they are equal, it throws them away, increments its return address by one to skip over the jump offset, and sticks it back on the return stack before returning.

That's a lot of work for something that has to be done inside every tiniest little definite-iteration inner loop!

I tried coding an assembly version, but it was complicated too, at 14 instructions.

F-83's LOOP implementation

The definition of LOOP from my copy of F-83 is in screen #75 of KERNEL86.BLK:

```
: LOOP
     COMPILE (LOOP)  2DUP 2+ ?<RESOLVE ?>RESOLVE    ; IMMEDIATE
```

So when you say LOOP in F-83, it compiles a call to (LOOP) into the word you're compiling, then does some compile-time jump-resolution stuff.

(LOOP) is in screen #11:

```
CODE (LOOP)   (S -- )   1 # AX MOV
LABEL PLOOP   AX 0 [RP] ADD   BRAN1 JNO
  6 # RP ADD   IP INC   IP INC   NEXT END-CODE
```

(I found these by running F83.COM in DOSBOX and saying "view loop" and "view (loop)". There's a built-in screen editor vocabulary that I don't really know how to use, but it brings up a screen editor on the definitions if you say "fix loop", at which point the word "a" switches between viewing the source code and the

"shadow" block containing the comments.)

I think this means the following in gas syntax:

_loop: mov $1, %ax # RP is the return stack pointer, defined in CPU8086.BLK screen 5 ploop: add %ax, (%bp) jno bran1 # jump if no overflow to "bran1" add $6, %bp # pop three items from return stack # IP is the interpreter pointer inc %si inc %si next # a macro defined as >NEXT #) JMP, which I assume means # "jump to the address >next"

"bran1" was previously defined in screen 9:

```
CODE BRANCH   (S -- )
LABEL BRAN1   0 [IP] IP MOV   NEXT END-CODE
```

which I'm pretty sure means branch: bran1: mov (%si), %si next

That just fetches a new address to jump to from the place the interpreter pointer currently points.

Notice how clever this is --- the common-case path length inside (LOOP) is only three instructions: MOV, ADD, JNO; and then there's one more MOV instruction inside BRANCH before we get to NEXT. But it's a little bit opaque.

In "Inside F-83" (insidf83.ZIP, 312170 bytes; see Chap4, 39424 bytes) C. H. Ting explains:

> F83 provides a solution by using three numbers on the return stack to handle the indexing and looping. The number at the bottom of the three is the address of the word right after LOOP, providing LEAVE with the return address to terminate the looping. The second number is the loop limit, offset by 8000H so that the index range from 0 to FFFFH becomes contiguous. The top number is the difference between the index and the limit, also offset by 8000H. At the end of the loop, LOOP increments the top number on the return stack by either one or the amount specified in the case of +LOOP, and tests for overflow from bit 14 to bit 15. The overflow condition occurs when the 8000H boundary is crossed from either direction. Therefore, both the positive and negative increments are handled correctly with a single run-time loop routine.

So the loop counter is one of the largest positive integers representable, and we increment it until it overflows. This does imply a bit more work on behalf of (do) though:

```
CODE (DO)   (S l i -- )   AX POP   BX POP
LABEL PDO   RP DEC   RP DEC   0 [IP] DX MOV   DX 0 [RP] MOV
  IP INC   IP INC   8000 # BX ADD   RP DEC   RP DEC
  BX 0 [RP] MOV   BX AX SUB   RP DEC   RP DEC   AX 0 [RP] MOV
  NEXT END-CODE
```

If I understand this, it means

```
_do:    pop %ax         # initial loop counter
        pop %bx         # loop limit
pdo:    dec %bp
        dec %bp
```

```
        mov (%si), %dx  # loop end address
        mov %dx, (%bp)
        inc %si
        inc %si
        add $0x8000, %bx # loop limit + 0x8000
        dec %bp
        dec %bp
        mov %bx, (%bp)
        sub %bx, %ax    # ax := initial loop counter - (loop limit + 0x8000)
        dec %bp
        dec %bp
        mov %ax, (%bp)
        next
```

If you're like me, when you read that code, you will shout, "What the fuck kind of sense does that make?" and haul out your Intel instruction set manual to see if maybe you've gotten the operands to SUB backwards or something. But of course the code is correct. (Skip the rest of this paragraph if that's already obvious to you.) The quantity -(loop limit + 0x8000) is the same as -0x8000 - loop limit, and -0x8000 is 0x8000, one more than the largest representable integer. So if the loop limit is 1, then this quantity will be 0x7fff, which will overflow after one iteration if the initial loop counter is 0. If it's 2, then it will be 0x7ffe. And so on. So then the "initial loop counter" is added to possibly reduce the number of iterations of the loop.

I think you could write this more briefly as follows, but hey, it's harder to build than to criticize, right? The one difference is that "pdo" now takes its parameters in %dx and %bx instead of %ax and %bx.

```
_do:    pop %dx
        pop %bx
pdo:    xchg %sp, %bp
        lodsw
        push %ax
        add $0x8000, %bx
        push %bx
        sub %bx, %dx
        push %dx
        xchg %sp, %bp
        next
```

Needless to say, this representation of loop state requires a little bit of computation to recover the loop counter in the word "I", in KERNEL86.BLK screen 15:

```
CODE I   (S -- n )
  0 [RP] AX MOV   2 [RP] AX ADD   1PUSH END-CODE
```

That is:

```
i:      mov (%bp), %ax
        add 2(%bp), %ax
        jmp push1
```

"1push" is one byte earlier than "next" --- presumably it pushes %ax, saving one byte on each of those primitives ("CODE words") that finish up by pushing a result. There's also "2push".

The Carry Variant

It seems like it would work just as well, and be somewhat clearer, to use the carry flag rather than the overflow flag --- so the loop is over when the loop counter increments up to 0. Here's my untested attempt at that variant:

```
_loop:  mov $1, %ax
ploop:  add %ax, (%bp)
        jnc bran1
        add $6, %bp
        inc %si         # couldn't we just lodsw?
        inc %si
        next

_do:    pop %dx         # initial loop counter
        pop %bx         # loop limit
pdo:    xchg %sp, %bp
        lodsw
        push %ax
        push %bx        # loop limit
        sub %bx, %dx    # dx := initial loop counter - loop limit
        push %dx
        xchg %sp, %bp
        next

i:      mov (%bp), %ax
        add 2(%bp), %ax
        jmp push1
```

The only changes are that we JNC instead of JNOing, and we don't need the add $0x8000. So in the 1 0 DO LOOP case, the thing pushed on top of the return stack will be -1, which will set the carry flag after one iteration.

The Carry Approach In High-Level Forth

Suppose we wanted to implement that same approach in high-level Forth (although without the third address on the stack for LEAVE). Instead of my ugly version:

```
: _loop r> r> 1+ r> 2dup xor
    if >r >r dup c@ - >r exit then
    2drop 1+ >r ;
```

we could have the nicer

```
: (loop) r> r> 1 um+
    if drop rdrop 1+ >r exit then    \ loop is done, or
    >r dup c@ - >r ;                 \ jump back and continue loop
```

in which the four normal-case instructions (mov add jno mov) of

the assembly version have become nine Forth addresses: r> r> 1 um+ if >r dup c@ - >r. (UM+ adds two numbers and leaves the carry on top of the sum on the stack.) This is slightly shorter than the 10 we had before, and probably considerably more efficient, because (at least at the moment, in my toy Forth) many of the words in the old version are interpreted, while in this version, all but "1" are machine-code primitives. The overall length of the thing also shortens from 19 cells (or bytes, in my case) to 17.

(do), however, gets very slightly hairier. My current version is as follows:

```
defbytes _do            # 10 0 DO ... LOOP loops 0, 1...9.
## This works by ( limit initial -- ) ( R: X -- X initial limit )
.byte b_swap, b_rpop, b_swap, b_rpush, b_swap, b_rpush
.byte b_rpush, b_exit
```

(I'm writing the return stack effect with the top-of-stack on the left, which is unorthodox.)

Which is like this:

```
: (do)  swap  r>  swap >r  swap >r  >r ;
```

Now we want (limit initial --) (R: X -- X initial-limit limit), so we have

```
: (do)  over -  swap  r>  swap >r  swap >r  >r ;
```

So this approach doesn't change the total program size, but it might be faster.

The Carry Approach Translated Into F21 Opcodes

These "real" Forth definitions:

```
: (loop) r> r> 1 um+
   if rdrop 1+ >r exit then
   >r dup c@ - >r ;
: (do)  over -  swap  r>  swap >r  swap >r  >r ;
```

would become something like this, in the four-instruction 20-bit words the F21 uses:

```
label (loop)  pop pop # nop
              1
              nop + c0 nop     \ jump if no carry; IIRC need NOPs before +
              cont
              pop drop drop #  \ carry, so exit loop
              1
              nop + push ret   \
label cont    push dup A! @A   \ fetch jump offset
              + push ret nop   \ and return to it
label (do)    pop A! over com  \ save return address in A
                               \ no swap, must use over
                               \ com is bitwise not; no subtract
              # nop + nop      \ no increment either; must use literal?
```

```
1
\ now we have on data stack: limit initial-limit
+ over push push \
drop A push ret  \ discard extra copy of limit, return
```

Now, I've never written an actual F21 or even MuP21 program, so I may have inflated that by a factor of two or three. (I'm not sure where jump targets are stored, for example, or in what format --- I assume as the next word --- or whether # and co and ret abort the instruction word, or whether there's a delay associated with memory references.) It's 14 20-bit words as I've written it, or 35 bytes, which is a reasonable code density --- my token-threaded Forth above makes it 27 bytes, and it's probably a bit over 35 bytes of x86 machine code.

But considering it from a speed point of view, it looks awful. The common-case path through (loop) is 15 instructions, if I counted it correctly --- that's 30ns on a 500MHz F21. While that compares reasonably with the time to do an empty delay loop on a 500MHz Pentium-II-class machine (my PIII Coppermine 700 runs x: loop x at 8.6ns per iteration) it's a heck of a lot slower than straight-line execution.

As it happens, looping is also a heck of a lot slower on my PIII. When I added four more instructions to that empty delay loop, it only slowed down by 3ns per iteration.

So Chuck Moore isn't crazy after all. He just prizes efficiency very highly.

Topics

Vanagon mail

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

Thanks to the wonderful people of the Vanagon list, I was able to find the parts I needed to rebuild our 1982 air-cooled engine last year, and our beloved Magic Bus has served my mother-in-law faithfully for about 2000 miles in the intervening year.

My wife and I have been visiting the San Francisco Bay Area during the last few months, and sometimes using the van. Unfortunately, this last week, the transmission died --- it feels pretty dead, really. The van is now at Valley Wagonworks in San Rafael, where Larry's Towing carried it this morning on a flatbed.

So now I'm faced with the choice of what to do with it. We need to sell it, because we're about to go back to Argentina in another month, and it won't do us any good there, and I think it may be telling us it's too tired to go on.

Paul at Valley Wagonworks suggests that he could put an overhauled GoWesty tranny in it for $1400, plus the $700 core charge (minus whatever is left of the old tranny --- synchro wheels and the casing, I imagine), plus I think some other incidentals. So that would cost around $1400. GoWesty would warrant the new tranny for four years. So then I'd have a fairly-good-condition 1982 Vanagon to sell.

Or he could put in a used transmission in exchange for the canopy and those 15-inch alloy rims that some previous owner put on it. This would also result in a fairly-good-condition 1982 Vanagon --- if the tranny is any good.

Or we could try parting it out. I just rebuilt the engine last year with a new Chinese head and set of pistons and two cylinders, and the engine sounds good, so that might be worth something; the battery is new; there's also the starter, the alternator, the 15" rims, a Dometic dual-power fridge, the propane stove, the aftermarket pop-top (this was an aftermarket camper conversion by some shop in Oakland), three good (but orange) doors, and I don't know what else.

But I don't know what's the best thing to do. Maybe I should try selling it for $5000 with the used tranny (and normal 14" rims, and no canopy), or $6400 with the GoWesty one? Are these realistic prices? What are the parts worth?

These are sad questions to be pondering, but perhaps their resolution can bring some joy to the new owner.

Topics

Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop

Kragen Javier Sitaker, 2018-10-28 (updated 2019-05-05) (3 minutes)

So the "i915" Intel graphics are actually part of the N3700 SoC https://en.wikichip.org/wiki/intel/pentium_(2009)/n3700, launched in 2015, which is a 1.6GHz quad-core amd64 CPU with 16 Braswell "HD graphics" execution units, running at 400MHz.

https://ark.intel.com/products/87261/Intel-Pentium-Processor-N3700-2M-Cache-up-to-2-40-GHz-

https://en.wikipedia.org/wiki/List_of_Intel_graphics_processing_units#Eighth_generation says this is of Intel's "eighth generation", and its core config is "128:16:2" with 25.6 GB/s memory bandwidth, which means 128 FP32 ALUs, 16 EUs ("execution units", each containing 2 SIMD-4 FPUs), and 2 subslices, each containing 8 EUs and a 4 texels-per-clock sampler. It also says that each EU is capable of 8 multiply-accumulates per clock cycle (one per ALU, I guess) in single-precision floating-point, but 16-bit floating-point is capable of "2× floating-point performance". Also they can do integer operations at the 32-bit floating-point speed, or 64-bit floating-point at ¼ the 32-bit speed.

Doing the math, that suggests that it can do 400 * 128 million single-precision multiply-accumulates per second, or 51.2 billion. Intel likes to double-count these multiply-accumulates, as if the addition and the multiplication were two separate operations, which dishonestly inflates their published flops counts by a factor of 2.

By contrast, https://en.wikipedia.org/wiki/Nvidia_Tesla says the NVIDIA Tesla V100 GPU Accelerator, released in 2017, reaches 14900 single-precision Gflops, about 300 times faster. And https://en.wikichip.org/wiki/intel/hd_graphics_630 says the later Kaby Lake GPUs reach 134 single-precision GFlops with 24 EUs at 350 GHz, so maybe it's plausible. https://news.ycombinator.com/item?id=18146625 says the V100 also costs US$10k, which is about 30× what my laptop costs, so the laptop has 10× worse price-performance.

It supports OpenCL and GLSL.

https://github.com/Themaister/GLFFT is an FFT implementation that has been tested on another GPU in the family. https://software.intel.com/en-us/intel-opencl is OpenCL which I think supports it. https://en.wikichip.org/w/images/f/f4/Compute_Architecture_of_Intel_Processor_Graphics_Gen8.pdf is a detailed description of the compute architecture.

https://01.org/linuxgraphics/downloads/2018q1-intel-graphics-stack-recipe calls Gen8 "Coffee Lake".

https://software.intel.com/en-us/articles/introduction-to-gen-assembly is an introduction to its assembly language.

So an interesting thing here is that the GPU can do 51.2 billion single-precision multiply-accumulates per second, or 51.2 billion 32-bit integer ops. But the CPU is 1.6 GHz with four cores, each of which supports the SSE4.2 instruction set extensions, which supports SIMD operations on 128-bit vectors of, among other things, four 32-bit floating-point or integer values. Supposing that each core is capable of running one such instruction each cycle, that's $1.6 \cdot 4 \cdot 4 = 25.6$ gigaflops of single-precision floating-point. This is only half the power of the GPU, but still, I think, substantially more than the CPU's non-SIMD capabilities. There are even SIMD instructions that allow you to use these 128-bit registers as 8 16-bit integers or 16 8-bit integers, though nothing for half-precision floats.

The problem with the SIMD i386/amd64 instructions — other than being only having half the total computrons — is that there are a shitload of them, and the instruction set is quite irregular, in large part due to backward-combatibility concerns.

Topics

How to generate unique IDs for IMGUI object persistence?

Kragen Javier Sitaker, 2014-09-02 (3 minutes)

So, um, immediate-mode GUIs are pretty cool. They allow you to avoid using data memory for objects on the display, which not only reduces your memory usage, but also prevents desynchronization and avoids heap allocation.

One problem with IMGUIs is that they generally need an ID for each widget, which you can generate somewhat clumsily using `__LINE__` in C, as explained by Jari Komppa. This clutters up your code a bit. Adrien Herubel's IMGUI library avoids this problem by maintaining a hidden `widgetId`; in imgui.cpp, for example:

```
bool imguiCheck(const char* text, bool checked, bool enabled)
{
        g_state.widgetId++;
        unsigned int id = (g_state.areaId<<16) | g_state.widgetId;
    ...
```

This means that instead of writing code like this:

```
 toggle = imguiCheck(GEN_ID, "Checkbox", checked1);
```

you can write code like this:

```
 toggle = imguiCheck("Checkbox", checked1);
```

But it has the problem that if you insert more widgets inside the same scroll area, for example because you have a variable-length subsequence of widgets, then your IDs will shift, and your widgets will get confused about which of them is active or hot.

It's possible to avoid this problem by using a sort of DOM-tree-ish approach and expanding each widget identifier from a single ID number out to an entire path from the root of a widget hierarchy — perhaps you limit each level of the hierarchy to 256 elements, and then you can identify a widget up to 6 levels deep with 6 bytes.

(Additionally if client code can increment the current widget ID up to some number after passing through a region of variable widget count, that could help.)

The hierarchy need not correspond to any nesting of rectangles on the screen. It can be entirely conceptual; in the environment where I came up with it, it was a menu tree, where only the innermost menu of your navigation was ever displayed. It enables you to tell not only on which widget the current keyboard focus is, but also all of the things it's nested within.

Many times, though, you'll have more possible widgets to display than you want to display at a given moment. In these cases it would be nice to leap past all these widgets without, for speed, actually having to execute the code for each one — and if the number is arbitrary, executing the code for each one would take infinite time. So this containment hierarchy can serve as a way to efficiently disable

different parts of the world.

Sigh. I'm not sure if this approach really generalizes well...

I wonder what reactive programming has to offer here. Meteor-style reactivity has a similar feel to IMGUIs; is there a unity between the approaches I'm not seeing?

Topics

Inductor thermocouple sensing

Kragen Javier Sitaker, 2019-06-01 (21 minutes)

I was wondering if a simple way to measure the temperature of a thermocouple is to use an inductor, and it looks like a viable approach, perhaps even providing precision comparable to or better than the standard op-amp approach, though it requires much thicker sensor leads.

Thermocouples can generate substantial power — those used for safety cutoffs in gas appliances typically power solenoids directly — but very low voltages — typically 20–40 μV/°, with 41 μV/° for the common type-K thermocouple (which is the one commonly used in gas appliances). Being made of solid metal, they have very low output impedances, but the low voltages are not particularly friendly to precise sensing with essentially capacitive CMOS.

So what if we use an inductor and a switch to generate a higher-voltage waveform from the low-voltage thermocouple signal, then measure the waveform with a cheap microcontroller?

An LR circuit would amplify the voltage, but requires fast measurement

The conventional approach to solving this problem is to use an op-amp (maybe a chopper to mask out flicker noise) to amplify the output voltage to a level where you can conveniently digitize it, but it occurred to me that perhaps you could put a thermocouple in series with an inductor, then open the circuit so that the inductor is obliged to discharge through a higher, known resistance.

To be more specific, consider the problem of trying to measure a temperature between 600° and 1200° (against a reference temperature we will assume is 0°) using a type-K thermocouple in series with a 100-μH, 1-Ω inductor in parallel with a 100-Ω resistor. At ΔT = 1200°, the thermocouple generates 49.2 mV, thus pushing 49.2 mA through the inductor (and 0.492 mA through the resistor). When the thermocouple is disconnected, the 49.2 mA starts to flow through the resistor instead, raising the voltage across it to 4.92 V, which is easily measured without further amplification. The current is then declining at 49.2 A per microsecond, so we only have a few microseconds (L/R = 1 μs) to measure it in, which is challenging but not infeasible with an off-the-shelf STM32.

A larger inductor would give us more time, but to avoid annoyingly high voltages, would require a correspondingly smaller resistor, and so parasitic resistances would start to play a larger role.

An LC circuit should make the measurement task easier

A different approach is to use a capacitor instead of the resistor for the inductor to discharge through, so that the current can oscillate back and forth several times. With the same 100-μH, 1-Ω inductor, our inductor energy $\frac{1}{2}LI^2$ is ½·100μH·(49.2 mA)² = 121 nJ; if we want our capacitor to charge up to ±2.5 V or less (so we can bias the

whole oscillation into the +0–+5 VDC range), then $C \geq 2 \cdot 121\ \text{nJ}/(2.5\ \text{V})^2 = 39\ \text{nF}$. Using a conventional 0.047 μF capacitor, then, we get up to ±2.27 V, and the oscillation frequency is $1/(2\pi\sqrt{(LC)}) \approx 73$ kHz, so we need to digitize the result at at least 150 ksps to avoid aliasing, which is straightforward.

Q = sqrt(L/C)/R, which works out to about 46 — about 200 μs per factor of e decay (τ), so the signal should remain detectable for up to a millisecond or two. I think this is probably a higher Q than we'll actually get in practice, but maybe not much higher.

So the procedure is to open the switch (which needs to operate in less than about 10 μs) and measure the ringing-down signal in order to get a reasonably precise indication of its amplitude over time, and in particular to extrapolate what its initial amplitude was when the circuit opened.

The sources of error I can see here are:

• Error estimating the time-domain start of oscillation; an error of 1τ results in an error of e ≈ 2.7, so to get 10°C of error at 1200° we need about τ/120 = 1.7 μs. Measuring this *directly* would mean we'd need to sample at hundreds of kHz, but that isn't necessary — we know the precise phase of the oscillation start — the current was at near-max and the capacitor was charged only according to the resistance of the inductor (49.2 mV in the above example). So we only need the accurate phase of the signal (to within 0.78 radians in this case) and to successfully detect its first oscillation. (In theory we don't need to estimate this because we're controlling it, but there might be delay in the switch.) Roughly half of our samples give us phase information whose precision is comparable to the precision of the amplitude information in (roughly) the other half; if we have 1000 useful samples (1 ms) then we have a few hundred samples each telling us the phase with something like 12 bits of precision, so we have maybe 16 bits of precision on the phase: something like an error of 15 microradians, which gives us a time-domain error for the cycle start on the order of 30 ps, translating to an amplitude error of 0.15 ppm, and thus totally overkill precision of 180 μK.
• Error estimating the amplitude of the signal, which is potentially a sort of weighted average of contributions from all the samples, maybe with some nonlinear filtering to reject out-of-band noise. Our 2.27 V initial amplitude will decrease proportionally by some 2 mV/°, so even 10 bits of precision (in the weighted sum, not each individual measurement) should be adequate. Offsets are inconsequential. If we figure we have about a millisecond of useful 12-bit samples at a megasample per second, that's 1000 samples, which gives us a precision of about 17 bits, a precision of about 17 microvolts on the 2.27 V initial measurement, or 375 nV on the original low-impedance thermocouple voltage, working out to a totally overkill precision of 9 mK. (Of course, the precision of your analog voltage reference is going to be a limiting factor here, too; if it's not accurate to 17 μV, you're not going to hit that.)
• Error in estimating the voltage amplification factor of the circuit — the 49.2 mV to 2.27 V factor described above, which comes from $\frac{1}{2}L(V_0/R)^2 = \frac{1}{2}CV_1^2$, and thus $V_1^2 = (L/(CR^2))V_0^2$, so $V_1 = (1/R)\sqrt{(L/C)}\ V_0 = Q\ V_0$, precisely the Q factor of the resonator. This is most sensitive to the precise value of the inductor's resistance,

which will vary with the inductor's temperature, since the inductor is not a precision resistor. However, the fact that this is precisely the Q factor of the resonator is wonderful — it means that we can measure the rate of ringdown and get a very precise measurement of this very amplification factor, accurate as of that moment. If we let the 2.27 V ring down to 0.5 mV (the limit of a 12-bit ADC) that's about $8.4\tau \approx$ 1.6 ms, and getting a $10°/1200° \approx 0.8\%$ error in the time domain would require an error of about 13 µs — not plausible, since we're talking about a systematic timing error over many measurements. In the voltage domain, an error in estimating the Q factor of 0.8% would amount to a (differential!) voltage error of $1 - (1.008)^{8.4} \approx 7\%$, which is even less plausible.

The only remaining source of error in this voltage amplification factor, then, will be parasitic resistance in the thermocouple, its leads, and the switch, which are outside the resonating loop, and the capacitor, which is in the resonating loop but not part of the series resistance that sets the initial inductor current. As an example, a 2N7002's on-resistance is 7Ω (see My attempt to learn about jellybean electronic components (p. 1974)). This could be improved by adding resistance *inside* the resonating loop, in series with the inductor, thus reducing the Q factor and thus increasing the earlier-mentioned errors; this would probably be a good tradeoff.

• Switch slowness error: we're indirectly measuring a discontinuous step function in current through the thermocouple by way of measuring its 73-kHz component. But discontinuous step functions of current cannot exist in the real world of nonzero parasitic inductances; any real switch, whether solid-state, thermionic, or electromechanical, will take some finite time to open and close, and electromechanical switches have the even more alarming behavior of bouncing when they close. If these errors attenuate or strengthen the 73-kHz component of the step function, you could get potentially totally incorrect measurements or even a blown input circuit.

This is somewhat exacerbated by the fact that MOSFETs with low on-resistance are big power MOSFETs like the IRF540N (44 mΩ), and so they have much larger Qg (71 nC in that case, compared to 2 for the 2N7000 — see My attempt to learn about jellybean electronic components (p. 1974).) Driving 71 nC into the gate in under 100 ns, which I guess would keep the switch-slowness error under about 1% and thus the temperature error under 12° at 1200°, requires a large transient current of 710 mA; the IRF540N datasheet does claim its turn-off delay time is 39 ns and its fall time is 35 ns.

This problem, too, can be ameliorated with a Q-spoiling resistor in series with the inductor, reducing the error due to the switch's on-resistance and thus allowing the use of smaller transistors. ON Semiconductor's 2N7000 datasheet claims a 10-ns turnoff time and 1.2–5Ω on-resistance at room temperature.

It could also be ameliorated by using an exotic GaN power transistor like the 39¢ EPC2036 with its 65 mΩ and Qg = 0.91 nC, or possibly an electromechanical relay.

• Thermocouple error: the metals in the thermocouple may not be pure, and their purity may change over time, especially at higher temperatures. Eventually it will probably burn out. It should be easy to make this the dominant form of error, but presumably it's a form of error we can calibrate out, since at any given time it depends only

on the temperature we're measuring.

• Cold-junction compensation error: thermocouples really measure a *difference* in temperature between two points, one of which is typically inside the measuring instrument. This is probably not a major problem for the ±10° precision I'm looking for, since a calibration at 18° is good from 8° to 28° even without any temperature compensation.

Here I've talked about the possibility of using an extra resistor in series with the inductor — which could take the form of just winding your inductor with thinner wire, and therefore perhaps just buying a cheaper inductor — but another possibility is using a precision resistor of 10–100 Ω *outside* the resonant loop. This would avoid reducing the Q (though it's probably excessive anyway) and reduce the error introduced by resistance in the thermocouple's leads, but by dropping most of the thermocouple's voltage, it would also attenuate the signal being measured and make the results sensitive to the unknown drifting parasitic resistance of the inductor.

Since the dominant sources of error here are parasitic resistances (which can be measured), the unknown thermocouple characteristics (which can be calibrated), and switch slowness, you can probably get better measurements by using an electromechanical relay.

A thing to consider is that this approach can quite reasonably measure 100 samples per second, but the thermocouple's temperature can't vary under non-disastrous circumstances by more than a hundred degrees per second or so, and in many circumstances, not more than a degree per second. So errors that pertain to only a single measurement, like phase error, can be averaged out over hundreds of measurements to obtain another 4–5 bits of precision.

(XXX I am not sure my calculations of how many bits of precision you get from averaging over some number of samples are right; I think I may have dropped a squaring or a square-root somewhere. The above calculations are based on the idea that quadrupling your number of samples gets you an extra bit of precision, which is 6 dB.)

(This design might also be adaptable for use as an energy-harvesting frontend.)

Comparison with a variant of the usual design, which may perform worse

Horowitz & Hill (2015 edition) talks a bit about the problem of amplifying thermocouples. On p. 936 it explains that you can use a US$16 Cirrus CS5532 24-bit delta-sigma converter to digitize the signal "directly" — by way of its integrated chopper-stabilized programmable-gain amplifier, which you can set to 64× gain to get a ±40 mV input range, or 32× for twice that. They mention that they balanced their thermocouple lines around ground to reduce EMI pickup.

On p. 1033, they also mention the Maxim MAX6675, which is a "thermocouple-to-digital-converter chip" with SPI output, and on p. 1083 they mention the MAX31855 SPI thermocouple ADC with 0.25° resolution from -270° to +1372° and built-in cold-junction compensation. The 1989 edition has a good deal more information on the subject of thermocouples on pp. 989–992, in fact recommending 10kΩ or more of input impedance to avoid errors from lead resistance.

But basically you just rig up an op-amp to amplify the small low input voltage to a larger output voltage, in, for example, the simplest non-inverting configuration: connect the thermocouple to the + input and run a voltage divider across the - input between the op-amp output and ground, with, say, a precision 47kΩ resistor and a precision 1kΩ resistor. This circuit makes no difficult demands of bias current, input range, bandwidth, slew rate, output power, power consumption, stability, gain, or broadband (as opposed to flicker) noise, but it does require a low input offset voltage (and precision resistors). Horowitz & Hill suggest using a chopper-stabilized op-amp, which typically requires a couple of external capacitors, like the US$1.60 MAX9617 — typical offset voltage 0.8 μV + 5 nV/°, with a 10–140 pA bias current. 0.8 μV in this context is a temperature error of 0.02°. This is fine for my intended application of controlling a pottery kiln, but it seems large compared to the much smaller errors I was talking about above. Worse, its worst-case offset voltage is specified as 10 μV, which is a temperature error of 0.2°. Other choppers are comparable — some a little better, others a lot worse.

Horowitz & Hill (1989) recommend rigging up the op-amp as a differencing amplifier in order to get good common-mode rejection of EMI, which I think is in large part a function of the high input impedance they've chosen due to their fine wires; but they emphasize that low offset voltage, a microvolt or better, is essential.

In the land of garden-variety op-amps, things are much worse. The US$0.24 Microchip MCP6001T-I/OT I declared in Jellybean ICs 2016 (p. 817) to be "the world's cheapest op amp" is specified as ±4.5 mV input offset voltage, ±2μV/°. ±4.5 mV would be an error of ±110°. The LM324 is ±2–7 mV, the LM741 is ±1–5 mV, and the CMOS TLC272 is ±1.1–10 mV, though its precision-trimmed TLC277 variant is ±500μV. And that's before we even get into flicker noise and drift! So if you were in the unfortunate position of having to improvise the thermocouple amplifier with whatever op-amps you could scrounge (see Ghettobotics: making robots out of trash (p. 2747)) the straightforward circuit wouldn't work; the resonant tank circuit this note is about would be a lot better. You might be able to get the straightforward circuit, or an instrumentation-amplifier variant of it, to work with a trimming resistor, since you only need about an order of magnitude improvement to get to the 10° (400μV) precision I wanted.

But can the inductor hack compete with the precision of a chopper-stabilized op-amp for thermocouple sensing? The chopper here is giving us 10 μV of error on a 49.2-mV signal, an error of about one part in 5000, 200 ppm. The inductor hack above is estimated to get an error of about 0.15 ppm from the time-domain start error, 7 ppm from the amplitude estimation error, some small error I didn't actually estimate from estimating the Q factor (probably dominated by the same voltage-measurement error as the amplitude estimation, and so probably of similar magnitude), and some potentially large errors from parasitic resistances (≈4000 ppm if we assume that we're using the IRF540N and its on-resistance varies unpredictably by 4 mΩ) and switching softness (say, the IRF540N's 35-ns fall time reducing the measured voltage at 73 kHz by 0.25%, 2500 ppm — although I may be estimating that wrong). So it might be

possible to get the inductor hack below 100 ppm of error, especially by using exotic components like electromechanical relays or GaN FETs.

(But then Horowitz & Hill get serious about precision design and explain the design of HP multimeters with 0.1 ppm error.)

Measuring parasitic lead resistance with multiple measurements

The problem with the switch on-resistance and the resistance of the thermistor and leads in the above design is that it drops some unknown amount of voltage before getting into our resonating tank circuit. Parasitic resistances in the tank itself introduce no error, as long as they're on the inductor side, because they show up in our Q measurement of the ringdown.

The problem is that we're faced with a Thevenin voltage-resistance pair, the voltage coming from the thermocouple and the resistance coming mostly from its leads, and we're trying to estimate the voltage from a single measurement from outside of it. This is clearly going to introduce significant error, and is the reason Horowitz & Hill suggest using a high input impedance, even though the thermocouple itself is very low impedance.

A different approach is to add some more switches with different known, precision, low-drift series resistances. Then we can measure how much the voltage across our tank changes with these various resistances, and thus the internal resistance of our voltage source. In the absence of measurement error, two switches would be enough.

However, if we're talking about MOSFET switches, we have the problem that the resistance of each switch is significant, significantly different from other switches, and variable over time and with temperature. So it might be worthwhile to instead seek a way of taking multiple measurements to calibrate out the source resistance that lets us keep using the same switch.

Specifically, we can leave the switch closed for less time than is needed for the current through the inductor to reach its steady state, which we've calculated above is about 1.6 ms. If we leave the switch open for only 100 µs or 200 µs, for example, the current through the inductor won't have ramped up to its max, and so we'll get a smaller pulse when we open the switch again. These shorter pulses will depend less on the parasitic lead resistances, because until the current gets going, those resistances aren't dropping any voltage, so the initial ramp-up of current in the inductor is independent of lead resistance and indeed of any resistance at all. By taking several different such measurements, it should be possible to calculate the outside-the-loop parasitic resistances with good precision — not just the resistance of the leads but also of the switching MOSFET.

Topics

Energy storage in a personal water tower: pretty impractical

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

Reading http://physics.ucsd.edu/do-the-math/2011/09/got-storage-how-hard-can-it-be/ I was inspired to think about gravitational potential energy storage, which at 3 m of head is only about 29 J / kg, about 500× worse than lead-acid batteries' energy density.

But what if you had a personal water tower? You could totally make it 30 m tall. Better still, if you live in a very dry area, you could drill 100 meters down and put a water turbine down the pipe. `100meters * gravity` is 981 J/kg.

He suggests that maybe for a house you want to have like 30 or 100 kWh of storage capacity to ride out cloudy days when your solar panels don't produce energy. 50 kWh is 180 MJ, so at 100 meters of head, you need 180 tonnes of water. That's a spherical tank some 7 meters across.

This is far from an impossible construction project, but it's considerably harder than building the rest of the house.

If you have 180 tonnes of clean, potable water stored, that could in itself be a useful form of preparedness; water and oxygen are the only life-essential resources that it's not usually considered practical to store a year's supply of.

Alternatively, you could drill a 1km-deep well and lower a weight into it on a rope, perhaps slowly enough that the water resistance wouldn't cause too much inefficiency. If you have a 200mm-diameter borehole and a 150mm-diameter weight that is 500m long, it has a volume of 8.8m³; if it's mostly made of quartz, as inexpensive things of that size probably would, it will have a specific gravity of about 2.6, or 1.6 if we subtract buoyancy. So it will weigh about 14 tonnes, net of buoyancy, and thus have an energy capacity of almost 70 MJ (19 kWh).

You might think that a rope capable of lifting 14 tonnes safely would be too large, but Dyneema can handle that load at about a 6mm diameter (although at retail that rope might cost you a couple thousand bucks). A 20mm-diameter rope would be a more than adequate safety factor; it will occupy 160 liters of space above ground when lifted all the way.

Topics

Broadcast ECC with graceful degradation, or avoiding the cliff effect

Kragen Javier Sitaker, 2018-12-18 (5 minutes)

Digital TV has a severe "cliff effect": due to the excellent error-correction codes employed, you have no warning that the signal-to-noise ratio is worsening until suddenly you can't decode the signal. We can see this as an engineering deficiency — the channel capacity has not fallen to zero, but probably to just below the bit rate of the signal. And in other situations, when the signal-to-noise ratio was better, there was excess channel capacity that the system is just wasting.

Analog TV, though worse in many ways, was better at this. When the signal levels were severely degraded, you'd see more noise on the screen, which effectively amounts to a lower resolution.

Could you do something similar with digital TV?

Since the failure of the MBONE, the standard approach with internet TV (CUSeeMe, YouTube, Skype, Netflix, etc.) is to downrez the stream until the receiver is able to receive all of it. Could you do something similar without bidirectional transmission, e.g. for recording media or broadcast TV?

Consider transmitting a 1920×1080 signal — first in 16-bit RGB without compression, to simplify the discussion. You can mipmap a 1920×1080 frame into a 240×135 frame, a 480×270 frame, a 960×540 frame, and a 1920×1080 frame. If you do it by summing the 4 pixels in each higher-resolution frame to get the corresponding pixel in the lower-resolution frame, then someone in possession of any 4 of those pixels can produce the fifth pixel. This means that if you have successfully received the lower-resolution frame, you would be satisfied with ¾ of the pixels in the next-higher-resolution frame.

Since the higher-resolution has ¼ of the pixels, that would seem to suggest that encoding the image in this way is "free", in that it doesn't cost any extra bits, but that's wrong; the lower-resolution frames need more bits of precision to make this work. But the cost is moderate: if you have 5 bits of precision per color component in the 1920×1080 signal, you need 7 at 960×540, 9 at 480×270, and 11 at 240×135. This totals to 35 186 400 bits, 4 398 300 bytes, while the raw signal is 31 104 000 bits, so the cost is about 13%. But the smallest frame size is only 1 069 200 bits, about 3% of the total.

If you encode the smaller frame sizes with higher levels of redundancy, then a receiver who's experiencing more noise might be able to receive the smaller frames even if the larger frames are lost. If you double the redundancy at each mipmapping level, then in effect you are using 5, 14, 36, and 88 bits per color component per pixel at the different mipmapping levels, which gives you 58 708 800 total bits per frame, almost twice the original bit rate.

Perhaps you can decrease this overhead by doing similar subsampling along the time axis — update the smaller frames less frequently, without the ¾ reduction on the larger frames when they

aren't accompanied by a smaller frame — but the tradeoff is there. Any bits you dedicate to redundancy for the sake of noisier receivers are in some sense not available for increasing the resolution for quieter receivers.

This may not be a fatal flaw, though, because the usual system doesn't serve those quieter receivers that well either — the wide SNR margin they enjoy is being completely wasted. This system could provide them with an even higher-resolution and possibly higher-frame-rate signal encoded with even less redundancy, with the consequence that those receivers get better service, too.

So it's only a narrow range of SNRs, those just above the threshold of the usual system, where the usual system is superior. From perhaps 0 to 6 dB or 9 dB above completely failing to work, the usual system is an improvement. But anywhere below that, this multirate system provides degraded service instead of no service, and anywhere above that, this multirate system provides enhanced service instead of baseline service.

Apparently one scheme for degradable modulation like this is called "hierarchical modulation", and "scalable video coding" is the name for the H.264 feature that implements another aspect of the above.

Topics

Spark particulate sieve

Kragen Javier Sitaker, 2016-10-06 (updated 2016-10-11) (7 minutes)

Particulate matter in air is generally bad for your lungs, and it can be a problem under some other circumstances, like in hospital operating rooms, biology labs, or semiconductor or hard disk clean rooms.

One way to remove particulates from air is to filter it by passing the air through a thin sheet with a lot of small holes in it. Particulates that are bigger than the holes won't pass at all; some fraction of smaller particulates will stick to the filter. Ideally, all the holes would be the same size, because if there are a fair number of really big holes, most of the air will go through them.

The particulates that are of most concern for health are in the categories PM10, which is 2.5–10 μm in diameter, and PM2.5, which is 1–2.5 μm in diameter. So it would be useful to have a sheet of material that was full of holes under 1 μm in diameter.

Make the sieve from a thin sheet of metal with sparks

A spark of a well-controlled energy applied to a thin sheet of metal, such as gold leaf or the metallization layer on a sheet of Mylar, should vaporize a well-controlled amount of metal in a pretty round hole, requiring about 14 kJ/g to vaporize aluminum `(((2470 - 20) K 24.20 J/mol/K + 10.71 kJ/mol + 284 kJ/mol) * 27.0 g/mol)`. A cylindrical hole of 1 μm radius through an 0.5 μm layer of aluminum metallization would amount to about 400 attoliters or about a picogram of aluminum, requiring about 15 nJ of spark energy.

Gold leaf is probably more practical, since it can be free-standing at as little as 0.1 microns thick. I'm going to assume that the energy needed to vaporize an area of gold leaf is close to the energy to vaporize the same area of aluminum metallization, because gold is thinner, but denser, and with higher specific heat and heat of fusion.

Electrical circuit considerations

Getting a spark that's only 15 nJ in air may be actually kind of tricky, because Paschen's Law has a minimal voltage of about 300 volts to start an arc in air. It's tough to get parasitic capacitances below about 10 pF, and at 300 V, that holds about 450 nanojoules. So the hole you're going to blow in the metal with just the parasitic capacitance, if you go around charging things up to 300 volts and then touching an electrode to the sheet, that hole will be about six microns across.

As an alternative, you could maybe use an inductor and strike a spark like you were using a stick welder: first bring the electrode into contact with the workpiece, allowing current to flow through an inductive load, then move it away, striking an arc which continues until the inductor's energy is dissipated.

How much current can you have flowing without heating things up? A 34-gauge wire is 160 microns across (thus a cross-sectional area of 20 000 square microns) and can handle 300 mA. If your area of

contact is, say, 100 square microns, then the same current density would be 0.5% of that, or 1.5 mA. Let's say 1 mA to be sure it'll work. ½LI² = 15 nJ at I = 1 mA if L = 2·15 nJ/(1 mA)², which comes out to a very conveniently large 30 millihenries. It's easy to build a circuit with less than that amount of parasitic inductance.

The most popular off-the-shelf inductor in this range at Digi-Key is the Murata 13R336C inductor: "33mH Unshielded Wirewound Inductor 60mA 68 Ohm Max Radial". It costs 59¢. It's ferrite-cored, and craps out at a couple hundred kilohertz; I'm not sure what would happen if you try to break the contact faster than ten microseconds, but I suspect that you lose most of the energy to core losses. If that happens, that's a problem.

An air-core coil with 10 mm diameter and 20 mm length has 25 mH once it gets to 2500 turns, which is totally feasible to wind with magnet wire. But I suspect you'll have enough parasitic capacitance to prevent it from responding within less than a few microseconds.

So you can build a power supply circuit that provides up to 1 mA at some low voltage (say 1–10 V), run it through a 30 mH air-core inductor, hook that up to your graphite electrode, and you're in business. Then you just have to move the electrode away from the metal sheet fast enough that the inductor's energy is mostly dissipated in the arc and not in the resistances of the rest of the circuit.

(Will 1–10 V be enough to drive a milliamp through such a small point contact? I think so.)

Hole concentration

By peppering a sheet of metal with sparks of well-controlled energy, you should be able to get the sheet up to 1% hole even without controlling the hole position without a significant number of "hole collisions" producing oversized holes. Even up to about 10% hole, almost all the "hole collisions" will produce holes that are only oversized in one dimension. If you can control the hole position closely, then with hexagonal close-packing you can get arbitrarily close to $3\sqrt{3}/2\pi \approx 82.7\%$ hole; if you reach that limit, the sheet falls apart into tiny triangles.

Non-air-filter applications

You can use the same spark approach to blow holes up to nearly a millimeter in diameter in the metal sheet, with fairly precise control of hole size (under 1% error, say). This provides a way to produce very fine-grained "mesh" sieves with precisely controlled hole sizes, which in theory could separate particles up to a particular size with very high precision, at least if they don't stick to the filter.

Separating particles into precisely graded size groups is the first step to being able to separate them by density and morphology with an air updraft.

Topics

Finite function circuits

Kragen Javier Sitaker, 2017-02-16 (updated 2019-05-17) (29 minutes)

I was considering how to specify finite state machines, and some interesting ideas occurred to me.

This is related to the notation for directed graphs in Graph construction (p. 3226) and the algebra of textures in An algebra of textures for interactive composition (p. 1283). (Circuit notation (p. 1161) is more concerned with describing the topology of circuit netlists, as you might use for designing analog circuits, while this note is concerned with designing digital circuits, and in particular sequential digital circuits, in terms of finite functions on alphabets; when considering a circuit, this is a higher level of abstraction, though it may well be a lower one when considering some dynamical system you want to simulate with the circuit.)

Introduction and outlook

A synchronous sequential digital circuit is a finite state machine, FSM. At each clock cycle i, it has exactly one state $S_i \in \Sigma$, and on each transition of the clock, it enters a new state $S_{i+1} \in \Sigma = F(S_i, I_{i+1})$, where $I_{i+1} \in \Upsilon$ is its input at the end of clock cycle i, and $F \in \Sigma \times \Upsilon \to \Sigma$ is its "state transition function". In this way, it generates a word in Σ^* of letters from Σ from an initial state S_0 and a word in Υ^* of letters from Υ.

(This is "circuit" in the sense of an electronic device, not in the theory-of-computation sense of a DAG of combinational logic.)

For example, here's an execution trace of a finite state machine that transliterates from lowercase Greek into titlecase Roman letters:

```
Ii:  π λ α τ ο ν ␣ κ α i
     ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
Si: א→P→l→a→t→o→n→␣→K→a→i
```

It should be noted that, since Σ and Υ are finite, so too is $\Sigma \times \Upsilon$ to which the domain of the state transition function F belongs, so it is possible to list F explicitly as an exhaustive case analysis. The state transition function for the above example would include cases like these:

```
(א, π) → P
(P, π) → p
(P, λ) → λ
(Q, λ) → λ
(א, λ) → Λ
(n, ␣) → ␣
(␣, λ) → Λ
(␣, κ) → K
```

In this case the input alphabet Υ and the state alphabet Σ are not the same; although they each include a symbol represented as ␣ here, that is just a notational pun — it doesn't make sense to say that those symbols are the same symbol or that they are different symbols, since

they belong to different alphabets.

A full exhaustive case analysis here, supposing that Υ contains the 24 lowercase Greek letters plus $\_$, and Σ contains the 26 uppercase modern Latin letters, the corresponding 26 lowercase, $\_$, and the initial state $S_0 = \aleph$, would require $25 \cdot 54 = 1350$ cases. This table is awkward to write out by hand, but it is small enough to be easily manipulated by computer. However, nearly all circuits of interest have far too many possible transitions to fit such a table in the memory of a conventional computer.

In the registers of the circuit we store a S_i, represented as some arrangement of bits; we interpret the set of inputs as I_{i+1}; and we represent the state transition function F as the RTL of our circuit, which can be realized with LUTs, combinational logic made of DAGs of gates (the theory-of-computation meaning of "circuit"), an EPROM, or whatever.

To design the circuit, it is not necessary to list the registers or even Σ — they can be inferred from F and S_0. The representation of Υ can be specified or free for the optimizer to optimize.

Thus we can design a sequential digital synchronous circuit simply by specifying its state transition function. Since that function's domain is finite, it is possible to do so by enumerating all its cases. However, usually it is preferable to use a more powerful language to be able to compose the desired function from sub-functions that are separately understandable, verifiable, and reusable.

It bears repeating that these are not the "functions" of JS or C, that is, subroutines. Subroutines are executed over some period of time, possibly terminating and possibly crashing. I'm talking about the mathematical object called a function: a mapping from each domain value to exactly one range value.

I will outline here a design for a language for specifying such finite functions that, I think, combines a number of advantages in a way no previous design I've seen does.

Design goals

Concision, flexibility, tractability, simulability, synthesizability

A relational algebra

Above, a sequential circuit was described according to its state transition function — its domain being the cartesian product of some finite alphabet of states Σ and some finite alphabet of inputs Υ and its range being Σ — and some assignment of Σ and Υ to concrete bitstrings. To each item in the domain it associates precisely one item in the range. We can generalize this to a state transition *relation*, in which each item in the domain is associated with any number of items in the range; this describes a potentially nondeterministic circuit, or a design specification with some liberty. A common way for this to occur in real life is for the circuit to have "don't care" combinations which we expect not to arise in practice. For example, a specification for adding BCD numbers need not specify what happens when the inputs aren't BCD numbers, but some other bit pattern. Also, though, we will see that it's often convenient to construct a deterministic function out of nondeterministic relations. We can also generalize to binary relations between any alphabets whatsoever.

(Historically, "relation" has meant this kind of binary relation, but

since the 1970s a rich theory of N-ary relations has grown up around databases. Here we will completely ignore N-ary relations, being concerned only with binary relations.)

Mostly we will be concerned with finite relations among finite sets, though sometimes they are exponentially large. This means that most of the algorithms we're interested in are trivial in the sense that at worst they need only enumerate some finite set of cases.

The algebra defined

(Very little of this is original.)

Here's the definition of the algebra of relations we'll be using, which is explained in more detail below. Consider a relation as a set of (domain element, range element) pairs; this gives us standard definitions of ∪, ∩, and \ operations, to which we add the following definitions of →, K, ∘, ⁻¹, car, cdr, ×, and *; note in particular that × is *not* the Cartesian product of sets:

$x \in a \to b \Leftrightarrow x = (a, b)$
$x \in \mathrm{K}(b) \Leftrightarrow \exists a\colon\ x = (a, b)$
$x \in \mathrm{A} \circ \mathrm{B} \Leftrightarrow \exists a\colon\ \exists b\colon\ \exists c\colon\ x = (a, c) \wedge (b, c) \in \mathrm{A} \wedge (a, b) \in \mathrm{B}$
$x \in \mathrm{A}^{-1} \Leftrightarrow \exists a\colon\ \exists b\colon\ x = (a, b) \wedge (b, a) \in \mathrm{A}$
$x \in \mathrm{car} \Leftrightarrow \exists a\colon\ \exists b\colon\ x = ((a, b), a)$
$x \in \mathrm{cdr} \Leftrightarrow \exists a\colon\ \exists b\colon\ x = ((a, b), b)$
$x \in \mathrm{A} \times \mathrm{B} \Leftrightarrow \exists a\colon\ \exists b\colon\ \exists c\colon\ x = (a, (b, c)) \wedge (a, b) \in \mathrm{A} \wedge (a, c) \in \mathrm{B}$
$x \in \mathrm{A}^* \Leftrightarrow \exists i \in \mathbb{N}\colon\ x \in \mathrm{A}^*_i$, where
$\mathrm{A}^*_0 = \mathrm{A}$
$i > 0 \Rightarrow \mathrm{A}^*_i = \mathrm{A}^*_{i-1} \cup \mathrm{A} \circ \mathrm{A}^*_{i-1}$

(This is somewhat related to Binate and to the function notation in Pattern matching and finite functions (p. 1235).)

Sometimes people consider the domain and range sets to be part of the content of a relation, aside from its pairs of elements, in the sense that two relations are not equal if they have the same sets of pairs of elements but different domains or ranges (which necessarily in this case include some elements that don't occur in any of their pairs). Here, however, when we mention the "domain" or "range" of a relation, we mean the sets of elements that occur on the left or right side of its pairs.

XXX typing this fucking Unicode is getting on my fucking nerves. Maybe I should switch to "." (the other way around) for ∘, ";" for ∪, ":" for →, "~" or "!" or something for ⁻¹, "&" or something for ∩, and some other things?

Constructing relations from items: → and K

The operator → produces a relation consisting of a single pair: $a \to b$ is the set $\{(a, b)\}$. So, applied to a, this relation gives us $\{b\}$, and applied to anything else, it gives us the empty set.

(Originally I was going to use ↦ rather than →, but I have → on my keyboard, and I gave in to expediency.)

The operator K(x) gives us a constant function whose value is x everywhere.

These are the only two operators of our eleven-operator algebra which take elements of relations, rather than relations, as operands. We could consider the tuple-construction operation "(,)" that constructs (u, v) from u and v as an operation on elements, but it doesn't belong to the algebra proper; we only use it in the meta-notation we're using to *describe* the algebra.

Constructing relations with set operations: ∪, ∩, and \

The ∪ set union operator allows us to construct a possibly larger relation from two relations. With ∪ and → we can construct all finite relations simply by listing the cases: (ℵ, π) → P ∪ (P, π) → p ∪ (P, λ) → λ, and so on. If we sort the clauses, we can consider this a canonical form, in the sense that two finite relations will have different canonical forms if and only if they map some domain element to different sets of range values.

The ∩ set intersection operator allows us to construct a possibly smaller relation from two relations, as does the \ set-subtraction operator. So, for example, (α → a ∪ υ → y ∪ υ → u) ∩ (υ → y ∪ υ → v) is just υ → y, and (α → a ∪ υ → y ∪ υ → u) \ (υ → y ∪ υ → v) is (α → a ∪ υ → u).

We could define these in the same style as the above definitions, but it hardly seems necessary, since we're just using them in the conventional way on sets that happen to be relations:

- $x \in A \cup B \Leftrightarrow x \in A \vee x \in B$
- $x \in A \cap B \Leftrightarrow x \in A \wedge x \in B$
- $x \in A \setminus B \Leftrightarrow x \in A \wedge x \notin B$

Call a relation R **deterministic** if $\neg\exists x\colon\ \exists y\colon\ \exists z\colon\ (x, y) \in R \wedge (x, z) \in R \wedge y \neq z$, which is to say that it's a function, and **reversible** if correspondingly $\neg\exists x\colon\ \exists y\colon\ \exists z\colon\ (y, x) \in R \wedge (z, x) \in R \wedge y \neq z$. (I was going to use "injective" for "reversible", but apparently sometimes "injective" is used to mean what I mean by "deterministic".) In particular, the empty relation ∅ is trivially deterministic and reversible.

All three of these operations preserve finiteness; they cannot construct an infinite relation from finite relations. (This is the reason for using \ rather than the more conventional set complement.) Only ∪ preserves infinity, in that the intersection or difference of infinite relations may be finite, but generally we don't care about preserving infinity.

∩ preserves determinism and reversibility — if either argument is deterministic, the result is deterministic, and similarly for reversibility. Importantly, though, it does not preserve nondeterminism or irreversibility: the intersection of two nondeterministic, irreversible relations may be deterministic and/or reversible. Indeed, it may be empty.

\ preserves determinism and reversibility in its left argument.

∪ does not preserve determinism and reversibility, but it does preserve nondeterminism and irreversibility: if the result is deterministic, both of the operands were deterministic, and similarly for reversibility.

The usual properties of set operations hold in the usual way, with the usual adaptation of DeMorgan's laws and the distributive and associative laws to set subtraction:

- A∪A = A∩A = A
- A∪B = B∪A
- A∩B = B∩A
- A∪(B∪C) = (A∪B)∪C
- A∩(B∩C) = (A∩B)∩C
- A∩(B∪C) = (A∩B)∪(A∩C)
- (A\B)∪(A\C) = A\(B∩C)
- (A\B)∩(A\C) = A\(B∪C)
- (A∪B)\C = (A\C)∪(B\C)

- $(A \cap B) \setminus C = A \cap (B \setminus C) = (A \setminus C) \cap B$
- $A \cup \varnothing = A$
- $A \cap \varnothing = A \setminus A = \varnothing$

We can also derive some laws for how they combine with the primitive relation constructors:

- $a \to b \cap c \to d = \varnothing$ unless $a = c \wedge b = d$
- $a \to b \cap \mathrm{K}(c) = \varnothing$ unless $b = c$, in which case it's $a \to b$
- $\mathrm{K}(a) \cap \mathrm{K}(b) = \varnothing$ unless $a = b$

(All of these, like the other laws presented below, can be proved in terms of the pointwise definition of the operators above.)

Composing relations with ∘

As a generalization of function composition, $A \circ B$ maps some element a to some element c precisely when B maps a to some (possibly different) element b, and A then maps that element b to c. So, for example, $(1 \to 3) \circ (0 \to 1)$ is $(0 \to 3)$. The reversals of direction are most unfortunate, XXX maybe I should switch to using | or . and the other direction of composition.

∘ preserves finiteness in the sense that the composition of two finite relations is also finite. The composition of a finite relation with an infinite one, or vice versa, or the composition of two infinite relations, may be either finite or infinite; but in particular $A \circ B$ has B's domain, or some subset thereof, and A's range, or some subset thereof, so if B's domain and A's range are both finite, then $A \circ B$ is finite, even if neither A nor B is finite.

∘ preserves determinism and reversibility in that the composition of two deterministic relations is deterministic, and the composition of two reversible relations is reversible. Nothing in particular can be said about the other combinations.

∘ has a number of useful algebraic properties: XXX some of these are probably wrong

- $\mathrm{K}(a) \circ (b \to c) = b \to a$
- $(a \to b) \circ \mathrm{K}(a) = \mathrm{K}(b)$
- $A \circ (B \circ C) = (A \circ B) \circ C$
- $A \circ (B \cup C) = (A \circ B) \cup (A \circ C)$???
- $A \circ (B \cap C) = (A \circ B) \cap (A \circ C)$???
- $A \circ (B \setminus C) = (A \circ B) \setminus (A \circ C)$??? this one seems really dubious

Inverting relations with $^{-1}$

A^{-1} is a relation that interchanges the domain and range of A, a generalization of taking the inverse of a function; so, for example, $(p \to q \cup p \to r \cup q \to r)^{-1} = (q \to p \cup r \to p \cup r \to q)$. Unlike the case with functions, every relation has an inverse, which is also a relation, and A^{-1-1} is always precisely A.

A^{-1} is finite if A is finite, and it's reversible if A is deterministic, and conversely deterministic if A is reversible.

We can write down some laws for how $^{-1}$ combines with other operators:

- $A^{-1-1} = A$, as mentioned above
- $(x \to y)^{-1} = (y \to x)$
- $A^{-1} \cup B^{-1} = (A \cup B)^{-1}$
- $A^{-1} \cap B^{-1} = (A \cap B)^{-1}$
- $A^{-1} \setminus B^{-1} = (A \setminus B)^{-1}$
- $A^{-1} \circ B^{-1} = (B \circ A)^{-1}$
- $\mathrm{K}(a) \circ \mathrm{K}(b)^{-1} = b \to a$

From these we can derive an unending farrago of worthless variants like $B \circ A^{-1} = (A \circ B^{-1})^{-1}$, $(a \to b \cup c \to d)^{-1} = b \to a \cup d \to c$, $A \cap B \cap C = (A^{-1} \cap B^{-1} \cap C^{-1})^{-1}$, and so on.

Constructing and deconstructing aggregates with car, cdr, and ×

The × "relational product" operator, if applied to two relations A and B over the same domain, produces a new relation A×B with the same domain, but whose range is the Cartesian product of the ranges of A and B — it consists of pairs constructed from an item from A's range and an item from B's range. This allows you to take two relations from the same range and combine their results into a single relation. So, for example, $(x \to y) \times (x \to z)$ is $x \to (y, z)$.

What × has joined together, the car and cdr relations put asunder; car∘(A×B) is some subset of A, all of A if B didn't lack any of A's domain values, and cdr∘(A×B) is analogously some subset of B, all of B if A didn't lack any of B's domain values. For example, $(y, z) \to y \in$ car, while $(y, z) \to z$ is in cdr.

car and cdr are unapologetically infinite and polymorphic, even Protean, but they have the property that composing them *on the left* (car∘R, cdr∘R) with a finite relation R produces another finite relation.

You might complain that car, cdr, and × reintroduce the asymmetry between domain and range that we dispensed with when we moved from functions to more general relations. This is merely a matter of convenience; you can use $(A^{-1} \times B^{-1})^{-1}$ or $R \circ car^{-1}$ nearly as easily as you can use A×B or car∘R, so there is no need to provide them as separate operations.

× preserves finiteness in the sense that the product of two finite relations is finite, but the product of two infinite relations, or a finite relatin with an infinite relation, may be either finite or infinite. More precisely, if either A or B has a finite domain, A×B will have a finite domain, and if both A and B have finite ranges, A×B will have a finite range. × does not preserve infinity in any of these cases, in the sense that A×B may be finite even if both A and B have infinite domains and infinite ranges.

× preserves determinism in the sense that if A and B are *both* deterministic, then so is A×B. In this case, A×B will also have no more pairs than either A or B — precisely the same number if and only if they are deterministic and their domains are equal. It preserves reversibility in the stronger sense that if *either* A or B is reversible, then so is A×B.

× has some interesting algebraic properties:

- $(a \to b) \times (c \to d) = \varnothing$ unless $a = c$
- $A \times (B \cup C) = (A \times B) \cup (A \times C)$
- $A \times (B \cap C) = (A \times B) \cap (A \times C)$
- $A \times (B \setminus C) = (A \times B) \setminus (A \times C)$
- $car \circ (A \times B) \cap A = car \circ (A \times B)$
- $cdr \circ (A \times B) \cap B = cdr \circ (A \times B)$

XXX this is simpler than the kind of relational product I eventually ended up using in Binate, but I imagine I'm going to end up with a bunch of opaque definitions saying things like "carry = car∘car∘cdr" in particular circuits. The Binate approach avoids that, and also sort of avoided the problem of infinities associated with car

and cdr. The idea there was that you would write {x: Xexpr, y: Yexpr, z: Zexpr}, and that would be a relation from the (intersected) domains of Xexpr, Yexpr, and Zexpr to records, one for each triple of range elements; then, new field-selection relations x, y, and z related those records to their individual fields. This avoids the problems of infinities and polymorphism mentioned above, but it's definitely conceptually more complicated, and I think the algebraic laws described above are a lot harder to state in the named-field case. On the other hand, I'm not sure how useful such algebraic identities will really be.

Transitive closure: *

XXX maybe this should be +? Because the * suggests that maybe zero times through the relation is fine too, in which case the identity should be included, and it isn't. It would also help avoid the semantic collision with Kleene closure once we get to regular expressions.

A* is a superset of A; it's the smallest relation such that A*∘A = A*. If A tells you where you can get in one move, A* tells you where you can get to in one or more moves. A* = A∪(A∘A)∪(A∘A∘A)∪.... So, for example, $(p \to q \cup q \to r)^* = (p \to q \cup p \to r \cup q \to r)$.

, surprisingly, preserves finiteness: a finite relation A necessarily has finite domain and finite range, and A has precisely the same domain and range as A. This provides a convenient way of enumerating A* in finite time for finite relations A, despite its recursive definition.

If A's domain and range are disjoint, A* = A.

* does not preserve determinism or reversibility, as can be seen by the example above in which the transitive closure of a deterministic, reversible relation is neither deterministic nor reversible. Indeed, I think A* can only be deterministic or reversible if A* = A.

* has relatively few useful algebraic properties:

- $A^{**} = A^*$
- $A^{-1*} = A^{*-1}$
- $K(a)^* = K(a)$
- $A \cap A^* = A$
- $A \cup A^* = A^*$
- $(A \circ A) \cup A^* = A^*$, *etc.*
- $(a \to b)^* = \varnothing$ unless $a = b$, in which case it's $(a \to b)$

Regular expressions

Concurrency

Nondeterminism

State trees

Example: XOR from NAND

Example: J-K flip-flop

Example: divide by 2 counter

Example: 3-bit magnitude comparator

Example: ring counter

Example: quadrature decoding

Example: quadrature step counting

Example: bit-serial addition

Example: bit-parallel addition

Example: simple ALU

Example: Single-port RAM

Example: Dual-port RAM

Example: 16-bit multiplier

Notes from the paper notebook

I started writing this file in 2017, and then never finished it; I just found it again now in 2019. But I have some notes from a paper notebook, in Spanish, which I am attempting to transcribe here. I appear to have been working from them when I originally wrote this file.

The expression notation for algebraically composing binary relations is fairly similar to Binate; you have relational composition, transitive closure (which, as a note later in this file points out, is necessarily terminating on finite sets), Cartesian product, union, intersection, set subtraction, relational inverse, relations car and cdr for dissecting Cartesian products, and notations for constant functions and primitive relations consisting of pairs of elements.

It's not a particularly tiny algebra — it has eleven primitives, of which two are primitive relations, one constructs a relation from an element, one constructs a relation from two elements, two more construct a relation from another relation, and the remaining five construct a relation from two relations. And, implicitly, there's the tuple-creation operator, which constructs an element of $A \times B$ from an element of A and an element of B. But, at least for finite relations — which are all we care about for specifying combinational logic! — it is complete (which really only requires $\mapsto$ and $\cup$) and I think may fulfill the desiderata I laid out below of being concise and flexible, and is certainly tractable, simulable, and synthesizable.

Then I use the union and primitive pairs to build up boolean NOT, and then without mentioning elements again, build up bitwise identity, the universal relation on bits and from pairs of bits to bits, restrictions of car and cdr to relations from pairs of bits to bits (called L_2 and R_2), their bitwise inversions, and finally a relation called NI_2, which I think is $(1, 1) \to 1 \cup (0, 0) \to 0$. I'm pretty sure this is what I was thinking about in the "Example: XOR from NAND" section below: build up all the primitive gates from just a NAND relation

(rather than from NOT and AND as I was trying on paper).

I think I was trying to derive AND from just NOT and the primitive relations, so that I would have an algebraic derivation of a whole Boolean algebra from just a definition of NOT. I think doing this isn't possible, because NOT is symmetric under exchange of 0 and 1, while AND is not. I think it also requires the use of some form of negation, which I hadn't tried using yet, and although I had included set subtraction in my set of logical definitions of operations, I had omitted it from the list at the end!

My comments about which relations are finite and which are infinite suggest that I was worried about infinite sets popping up and making algorithms fail to terminate, as does my use of set-subtraction \ rather than simple negation, and the aside (on the facing page, which I set aside for asides) about providing only $A \div B$ instead of B^{-1}, which would keep infinite sets that occur in the range confined to the range, preventing them from escaping into the domain and making it infinite too. This would have preserved the ability to finitely enumerate the domain of any relation except car and cdr (which I suggested demoting in the same aside) although if ranges were infinite anyway I'm not sure why that would be especially useful.

I think the idea then was that you can consider a binary relation to be a specification of a combinational logic function, and you can consider that combinational logic function to be a specification of a sequential logic behavior (if you can identify which parts are state and which are inputs, anyway); but, that under other circumstances, other forms of behavior specification, such as the sequence/alternation/repetition operations of regular expressions, are more convenient; and that other ways of combining finite state machines, such as (some unspecified form of) concurrency, (some unspecified form of) nondeterminism, and (some unspecified form of) state trees, are sometimes more useful.

You could conceivably treat an FSM (S_0, F) as a relation from input strings to output strings — a function, if deterministic, but in any case a relation — and attempt to handle them with the same algebra of relations. But I'm not sure I had that in mind, and it may not be a useful way to compose finite state machines. I suspect I just had in mind that you would construct primitive finite state machines using binary relations to specify their transition function, then use the other mechanisms to combine these primitive state machines in series, in parallel (note that tconcurrency was a heading not in the paper notebook), nondeterministically, and in trees of state (perhaps as in Harel statecharts, although I didn't know about those then).

Componer FSMs

- Intro [link]
- Relaciones [link]
- Expresiones regulares
- No determinismo
- Arboles de estado

Intro

Un circuito digital sincrónico es una máquina de estados finitos, FSM por sus siglas en inglés. En cada momento tiene un estado $S_i \in$

Σ, y en cada transición del reloj entre un nuevo estado $S_{i+1} \in \Sigma = F(S_i, I_i)$ donde I_i es su entrada en momento i, y F es una función $\in \Sigma \times \Upsilon \to \Sigma$. Así genera una palabra en Σ^* a partir de una palabra en Υ^* y un estado inicial S_0.

```
π λ α τ ο ν ␣ κ α i
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
P→l→a→t→o→n→␣→K→a→i
```

Cabe destacar que, ya que Σ y Υ son (ilegible) así también $\Sigma \times \Upsilon$, así (ilegible) hacer un listado de F.

No incluidos:

- H Z (Hi-Z?)
- open-collector
- asyncronia
- variables (registros o pseudo-registros con nombres)

En los registros del circuito almacenamos S_i, representando como un conjunto de bits, interpretamos el conjunto de entradas como I_i, y representamos F como el RTL del circuito, que se puede realizar con LUTs, lógica combinacional de DAGs de compuertas (un "circuito" de teoría de computación), un EPROM, lo que sea.

No es necesario especifcar los registros ni Σ — se pueden inferir de F y S_0. La representación de Υ puede ser especificado o libre para optimizar.

Así que podemos diseñar un circuito digital secuencial sincrónico especificando su función de transición nomás, y ya que es finita, lo podemos hacer enumerando los casos. Pero para la mayoría de circuitos, nos conviene usar un lenguaje más poderoso para poder componer la función deseada de sub-funciones separadamente entendibles, verificables, y reutilizables.

Cabe repetir que ésos no son las "funciones" de JS o C, es decir, subrutinas, que pueden ejecutarse y así tomar tiempo.

Relaciones

- $(a, b) \in A \land (b, c) \in B \Rightarrow (a, c) \in A \circ B$
- $(a, b) \in A \Rightarrow (a, b) \in A^*$
- $(a, b) \in A \land (b, c) \in A^* \Rightarrow (a, c) \in A^*$
- $(a, b) \in A \land (a, c) \in B \Rightarrow (a, (b, c)) \in A \times B$
- $(a, b) \in A \Rightarrow (a, b) \in A \cup B$
- $(a, b) \in B \Rightarrow (a, b) \in A \cup B$
- $(a, b) \in A \land (a, b) \in B \Rightarrow (a, b) \in A \cap B$
- $(a, b) \in A \Rightarrow (b, a) \in A^{-1}$
- $(a, (b, c)) \in A \Rightarrow (a, b) \in \leftarrow A$
- $(a, (b, c)) \in A \Rightarrow (a, c) \in \rightarrow A$
- $((b, c), b) \in \text{car}$
- $((b, c), c) \in \text{cdr}$
- $(a, b) \in A \land (a, b) \notin B \Rightarrow (a, b) \in A \setminus B$
- $(a, b) \in K(b)$
- $(a, b) \in a \mapsto b$

Así con $\mapsto$, K, car, cdr, $^{-1}$, $\cap$, $\cup$, $\times$, *, y $\circ$, podemos combinar relaciones para formar otras relaciones, y podemos construir cualquier relación finita.

(ilegible) va al (intentar?) (evitar?) dominio inf(inito?) es brindar una operacion $A \div B = A \circ B^{-1}$. Pero no ayuda con car y cdr (pero pueden ser sus propias

operaciones.)

De éstas operaciones, car, cdr, y K(x) son relaciones infinitas, y los otros solo general relaciones infinitas a partir de relaciones ya infinitas.

NOT es $1 \mapsto 0 \cup 0 \mapsto 1$.

AND es $(0, 0) \mapsto 0 \cup (0, 1)$ (ilegible) $(1, 0) \mapsto 0$ (ilegible) a partir de NOT sin $\mapsto$.

- $\mathrm{NOT} \circ \mathrm{NOT} = I_2$
- $\mathrm{NOT} \cup I_2 = U_2$
- $U_{22} \cap \mathrm{car} = L_2$
- $U_{22} \cap \mathrm{cdr} = R_2$
- $(U_2 \times U_2)^{-1} = U_{22}$
- $\mathrm{NOT} \circ L_2 = \overline{L}_2$
- $\mathrm{NOT} \circ R_2 = \overline{R}_2$
- $(L_2 \cap R_2) = NI_2$

Topics

English diphones

Kragen Javier Sitaker, 2019-12-03 (5 minutes)

How much speech would you need to sample to make a diphone synthesis engine out of it?

How many diphones are there in English? The following set of words more or less covers the phoneme set:

Ack, ache, mother, father, bet, cite, done, leash, gum, hedge, sip, rot, sought, room, foot, azote, few, valley, which, yet, think, pleasure.

`espeak --ipa` renders this as follows:

```
 ˈak
 ˈeɪk
mˈʌðə
fˈɑːðə
bˈɛt
sˈaɪt
dˈʌn
lˈiːʃ
ɡˈʌm
hˈɛdʒ
sˈɪp
ɹˈɒt
sˈɔːt
ɹˈuːm
fˈʊt
 ˈazəʊt
fjˈuː
vˈalɪ
wˈɪtʃ
jˈɛt
θˈɪŋk
plˈɛʒə
```

To this we need to add at least æ, which I'm not sure how to do with espeak without -v en-us.

This reduces to this set of phonemes, as I understand them:

```
ʌ
ɑː
a
aɪ
æ
b
d
ð
dʒ
ʒ
eɪ
ə
əʊ
f
h
```

iː
j
uː
k
l
m
n
ɔː
p
s
t
tʃ
ʃ
uː
ʊ
v
w
z
θ

That's 34 phonemes. That means that English can't have more than $34^2 = 1156$ diphones, although in practice English phonotactics exclude most of these.

English speech is typically on the order of 200 wpm, and each word might average 5 phonemes, so this is about 1000 phonemes (and thus 999 diphones) per minute of speech, which is to say that you could in theory capture all the English diphones from roughly one carefully-constructed minute of speech. If instead the diphones are uniformly randomly distributed, the number of uncovered diphones will drop by a factor of 2.7195 (about e) per 1156 phonemes uttered, an exponential progression 1156, 425.1, 156.3, 57.5, 21.1, 7.8, 2.9, 1.1, 0.39, 0.14, so complete coverage would usually require some 7 or 8 minutes of recorded speech.

But in fact some phonemes and thus some diphones are much less frequent than average: the order is very crudely something like, first, n, ə, d, and t; a factor of 2 lower, a, f, z, s, m, l, and iː; a factor of two lower, v, p, k, ð, w, uː, j, b, and ɔː; and so on. A standard Zipfian guess is that the 34th phoneme is about 3% as common as the most frequent one. This would tend to lengthen the time required for complete coverage, just as phonotactic restrictions would tend to shorten it.

Crudely, though, I think it's reasonable to guess that sampling a few minutes of speech should be enough to produce a comprehensible approximation of a person's speech, without requiring previous knowledge like statistics of other people's speech in the language.

An interesting question is how to formulate a prepared speech that you could read in a minute or so that would provide a richer-than-average set of diphones. Given, say, the pronunciation of each word in /usr/share/dict/words, you can formulate this as a combinatorial optimization problem: the shortest set of words containing all the diphones contained in all the words. Analyzing a recording of such a prepared speech would be easier than analyzing arbitrary speech because the alignment problem would be much easier.

As a quick experiment, I recorded myself reading the above list of 22 words, about 72 phonemes, slowly, in 21 seconds, under relatively poor recording conditions, with `rec words.wav`. Then I encoded it with LPC-10 with `sox words.wav words.lpc10`. The resulting LPC-10 file was 6447 bytes (293 bytes per word, 90 bytes per phoneme) and was pretty comprehensible; it would surely contain enough information for phoneme alignment.

How exactly do phoneme alignment?

Maybe the matrix profile would help, for example, over a time series of mel-frequency cepstral coefficient (MFCC) vectors, for example, or of LPC coefficient vectors. Maybe for a sufficiently small dataset the whole correlation matrix that the matrix profile is a "profile" of would be more useful; the peaks in that correlation matrix might tell you at which time points the same phoneme occurred previously and later, which you could compare against the temporal distribution patterns of phonemes in the supposed phoneme sequence.

Maybe you could use something like the Viterbi algorithm, in which the "symbols being transmitted" are positions in the supposed phoneme sequence rather than phonemes, each with a transition to itself (that is, the phoneme continued from one time window to the next) and to the following phoneme in the sequence --- although this requires some a priori knowledge of what each phoneme sounds like. But once you have even the most basic guess about alignment, you can use that to get an improved guess about what each phoneme sounds like, then repeat the procedure.

This is surely a problem that is well explored in the research literature.

An interesting possibility is to try to use some kind of self-organizing maps to map the space of observed sounds, perhaps using three or four dimensions conjectured to correspond to tongue position and nasality; the idea is that the continuous changes in sound harmonic content provide you with information about which sounds are physically adjacent to which other sounds. (Voice pitch and sibilance are probably independent dimensions there.)

Topics

Can artificially-lit vertical farming compete with greenhouses?

Kragen Javier Sitaker, 2019-09-08 (12 minutes)

Comment at https://news.ycombinator.com/edit?id=20908476:

the point of sun light is that it has a very attractive cost of exactly 0$. It costs absolutely nothing. So it doesn't matter that solar panels are not very efficient. They actually are getting better and certainly 16% efficiency is nowhere near the best there is.

For most purposes I agree with you! However, in this case, the alternatives I am considering are:

• Build a 10000 m² greenhouse full of, say, lettuce, perhaps on multiple shelves ("vertical farming").
• Build a 10000 m² solar park full of solar cells, then use the energy produced by the solar cells to illuminate lettuce being grown inside an opaque concrete box, of some arbitrarily variable size.

In this comparison, the efficiency of solar cells and of LEDs matters very much indeed! Because of their inefficiency, you get 30 times as much lettuce in case #1. That's the reason I think this scheme is uneconomic except in unusual cases. In another part of the comment thread, I agreed that abundant wind energy is a case where it might make sense.

It's true that there are solar cells in commercial production that are 30+% efficient instead of 16%. However, those are specialty solar cells designed for use in things like spacecraft (we used them on our satellites at Satellogic, for example.) Consequently, they are eye-wateringly expensive and not getting cheaper, and so nobody is building solar parks with them, particularly since non-arable land is abundant and will remain so for a couple of decades, while the prices of low-cost 16%-efficient solar cells are dropping like a hafnium pellet.

The whole point of LEDs is that they are supposedly very efficient; around 40-50%.

No, LEDs are nowhere close to 50% efficient. LEDs have many wonderful attributes, including tunable color spectra, directionality, the possibility of being scaled down to submillimeter scales (hard to do with an incandescent bulb!) and, indeed, very good efficiency ---*compared to other light sources*, that is. The problem is, all light sources are shitty when it comes to efficiency; LEDs are just less shitty. That's why we charge our cellphones wirelessly with induction coils, not LEDs and expensive multijunction photovoltaic cells.

It's hard to get your hands on good efficiency numbers, because LED vendors don't quote any kind of absolute energy efficiency number in the datasheets, because they only publish luminous efficacy (because that's what people normally care about). In theory, we can derive the absolute energy efficiency using a luminous-efficiency curve: https://en.wikipedia.org/wiki/Luminosity_function. I'll see if I can do that in a separate comment.

heat is not actually energy loss in a vertical farm.

This argument turns out to be wrong; I've explained why in detail in https://news.ycombinator.com/item?id=20906210, but in brief, 87% of the energy we're talking about gets lost in the solar park, not

the hothouse, and artificial illumination to crop-growing levels produces so much heat that you need to air-condition the hothouse rather than heating it. Moreover, produced heat is *always* energy loss, because you can always reduce it further even by adequate insulation.

Meaning that the reason vertical farming is getting a lot of attention is that the cost of energy has been dropping by rather a lot and is projected to continue to drop. Effectively this dominates variable cost in a vertical farm.

It's true that the cost of energy dropping, and to levels that would make people in the Space Age gasp. I still don't see how that justifies building a 30-hectare solar park to grow the same lettuce you could grow in a one-hectare greenhouse. I mean, how big is your armored vault hothouse going to be?

You seem to be arguing this cost is too high. That seems to be countered by the many people actually growing stuff in greenhouses for decades this and making plenty of money.

No, man, that's not what I'm *saying*, man. I'm saying that if you're going to build a hothouse, make it a greenhouse. Daylight it, with skylights and/or lightpipes. Maybe supplement with artificial lighting some of the time. Lighting it with a solar farm that's thirty times as big is going to be more expensive, unless solar cells are thirty times cheaper than glass per square meter, and lighting it with fossil fuels is more expensive still. Fuck, thirty times cheaper than plexiglass. Thirty times cheaper than the shitty transparent plastic wrap we used to make greenhouses in Ecovillage Velatropa. If you're right and, against all odds, LEDs are now 50% efficient, exceeding the theoretical ideal luminous efficacy maximum Wikipedia gives, the threshold becomes fifteen times cheaper instead of thirty --- still improbable!

While I'm calculating the efficiency of LEDs for you, would you mind undoing your downvote, please?

There are some example calculations in https://en.wikipedia.org/wiki/Luminous_efficacy#Examples_2 of the overall luminous efficiency of different kinds of light sources. Candles are around 0.04% efficient, while incandescent bulbs range from 1.2% efficient (though there's really no lower limit) to 5.1%. The illumination LEDs listed are in the range of 15%–25%. None comes close to 40% efficiency, much less 50%. Low-pressure sodium lights, the kind you occasionally still see in streetlights, head the pack with luminous efficiency up to 29%, which is why they're so popular with clandestine indoor growing operations.

Here's the deal with luminous efficacy: to calculate the absolute energy-efficiency (η) of an electrical light source, we need to know the input power (watts: $P = EI$) and the output power, which is in the form of radiant flux (also watts). The ratio between these two is the efficiency; it tells us how much of the electrical energy that goes into the luminaire comes out as light, or inversely, just gets wasted as heat.

But nobody publishes radiant flux numbers for their light sources because what people care about is mostly the *brightness* and, sometimes only secondarily, the temperature and the color rendering index. Brightness --- luminous flux --- is measured in lumens, not watts. But because the humans' eyes are not equally sensitive at all wavelengths, converting between radiant flux and luminous flux is complicated. A hundred watts of radiant flux at a 900-nanometer wavelength counts as zero luminous flux, because the humans' weak

little eyes can't see it at all. Similarly, a hundred watts of radiant flux at 350 nanometers is also zero luminous flux, although that will give a human a sunburn pretty quickly. A hundred watts at 555 nanometers, where the humans' cone cells are most sensitive to light, looks twice as bright as a hundred watts of radiant flux at around 520 nanometers or 630 nanometers.

So, to convert between radiant flux and luminous flux, we use a weighting function called the luminous efficiency function, which reflects this variation. At 555 nanometers the weight is 683 lumens per watt, a number arbitrarily chosen to make the SI candela (a lumen per steradian) approximate the Victorian-era candlepower as closely as possible. At every other wavelength, it's lower, according to a standardized approximation of the photopic luminosity function of the humans' eyes, which can be downloaded from the image links at the top of https://web.archive.org/web/20081228083025/http://www.cvrl.org/database/text/lum/vljv.htm (I recommend https://web.archive.org/web/20070518010940/http://www.cvrl.org/database/data/lum/vme_1.txt, specifically).

Now let's consider a modern illumination LED I happen to have the datasheet handy for, the Cree XLamp CXA2530 family. On p. 6 of its datasheet, it displays its relative spectral power distribution at 800 mA and 85 degrees, from 380 nm to 780 nm. Let's take the "warm white" variety, because its distribution is flatter, so the calculation errors will be smaller. Unfortunately, it's normalized to percentage of maximum brightness. From 380 nm to 430 nm, it's between 0 and 10%; from 430 nm to 480 nm, it's between 10% and 50%; from 480 nm to 530 nm, it's between 20% and 55%; from 530 nm to 580 nm, it's between 55% and 90%; from 580 nm to 630 nm, it's between 85% and 100%; from 630 nm to 680 nm, it's between 30% and 85%; from 680 nm to 730 nm, it's between 5% and 30%; and from 730 nm to 780 nm, it's between 0 and 5%.

On p. 7 we find that these LEDs --- LED arrays, really --- hit this 800-mA current level around 37 volts. Also, they're a little more efficient if you run them at a lower power level, because the light output is fairly closely proportional to the current, while the current increases with voltage. So if you run them at 400 mA instead of 800 mA, you get half the light output, but the voltage drops to 34 volts, about a 10% efficiency improvement (so, if we end up computing that they were 22% efficient, that could be improved to 24% this way). Also, if they're sufficiently cooled, we can get another 10% improvement. I'm going to use the nominal 800 mA and 85 degrees, but keep those possible improvements in mind.

This works out to 29.6 watts of electrical power.

On p. 9 we find that the "brightness" performance groups range from 2100 lumens up to 3950 lumens, with the top bin, U2, having a minimum luminous flux of 3680 lumens at 85 degrees and 800 mA. (This also shows up on p. 3 of the datasheet.) I'm going to assume that Cree isn't just being optimistic and some of their LEDs in this family actually do test into this bin.

Now, if you were trying to find out the maximum luminous flux an LED could put out at a given radiant flux, you would calculate with as much of the energy at the highest-luminous-efficacy wavelengths as possible. Ideally, all of the energy would be at 555 nm,

although here we know that some of it is outside the 530–590 nm range, and some of it is even out past 680 nm one way and 480 nm the other. If someone had managed to produce LEDs that had the same 3680 lumens at purely 555 nm and 29.6 W, those LEDs would be producing 5.39 watts of green light (and be 18.2% efficient.)

But in this case we have the inverse problem: we're trying to find out the *maximum* radiant flux these LEDs could possibly be emitting, given their published luminous flux rating. That is, because they're emitting light at other wavelengths as well, they can emit *more radiant flux* at the same luminous flux. And to put bounds on how much radiant flux they could be emitting, we need to do the opposite: put as much of the power as possible at the least-visible-possible wavelengths.

Given the numbers above, the total area under the curve of the LED's spectral power is between (0 + 10% + 20% + 55% + 85% + 30% + 5% + 0)50 nm = 205% * 50 nm and (10% + 50% + 55% + 90% + 100% + 85% + 30% + 5%)50 nm = 425% * 50 nm.

Topics

Progressive revealment crypto

Kragen Javier Sitaker, 2019-04-10 (2 minutes)

If I encrypt a document with a symmetric key to which I give you all but the last 40 bits, and the decrypted document contains a way to verify that it's the correct decrypted document (lower entropy, say, or a checksum), then you can decrypt it in about 2^{39} work. If this document contains all but the last 40 bits of the encryption key to another document, then once you've decrypted it, you can decrypt the second document in about another 2^{39} operations, or 2^{40} operations in total. Such a chain of documents forms a sort of "time lock crypto"; it is difficult and perhaps infeasible to decrypt the documents further down the chain without first decrypting the earlier documents.

It's not a very good timelock, in that I (the puzzle-maker) must encrypt the documents serially as well, and my only advantage over you (the puzzle-solver) is that I only had to encrypt each document once, while you had to decrypt it 2^{39} times. But those decryptions can be carried out in parallel; in the limit of unbounded parallelism, decryption is as fast as encryption plus a bit of communication overhead.

But, in the case where the puzzle-solver has limited parallelism, the puzzle can be made arbitrarily hard. Suppose, for example, that the puzzle-solver only has a million-node cluster available to try keys with, each node being as powerful as my laptop. Then, using the 40-bit example above, a chain of documents that I needed one day to encrypt on my laptop will take them a million days — 3000 years — to decrypt.

The downside of this is that, if they are decrypting it on their laptop instead of a million-node cluster, they will need 3 billion years instead.

Topics

Another candidate lightweight frequency tracking algorithm

Kragen Javier Sitaker, 2017-08-18 (4 minutes)

Okay, I can't find my previous notes on frequency identification, but here's what I came up with tonight, which I'm pretty sure is good. You get two booleans by comparing the current sample to 0 and to the simple moving average (or maybe double simple moving average) of the last N samples, getting a kind of estimate of its derivative. These two booleans give you a quadrature rotation signal: 00, 01, 11, 10, repeat. The timing of the (corresponding forward) transitions of this signal is ¼ the period of the input signal. You get the transition immediately at the sample, with no codec or processing latency the way an STFT has, and fairly little computation per sample:

```
// Advance ring buffer sample pointer
xj = &x[j++];
if (j == n) j = 0;
// Update ring buffer x and moving average numerator m
m -= *xj;
*xj = xi;
m += xi;
// Compute new quadrature state.
s = (xi > 0) << 1 | (xi * n > m);
if (s != os) transition(t[os << 2 | s]);
os = s;
```

So in the usual case, that works out to an increment, an indexed address computation, four comparisons, a memory fetch, a subtraction, a memory store, an addition, a multiplication by constant N, a bit shift, a bitwise or, and a state variable update which could be eliminated by unrolling the loop once: 14 operations per sample, all very simple except possibly the constant multiplication. In the case of a detected transition, you get a bitshift by two, an or, and a table lookup.

All of this is LTI up to the last bit where we take the signs and look stuff up in a transition table.

The FIR frontend is a very chintzy bandpass filter: the (implicit) subtraction from the current sample attenuates low frequencies, while the moving average itself attenuates high frequencies, but can only manage about 3dB of attenuation on the overall transfer function because of the impulse at lag 0. An additional moving-average pass (6 more operations, including the memory references and wrap) cleans up the high-frequency part of the response. We could make it a somewhat better bandpass filter, at the expense of fractional cycles of response latency, by using something else than the current sample: another simple moving average, which of course suggests that maybe we should use two moving averages of the same size, abutting but not overlapping, thus removing the multiplication as well.

Perhaps more interesting, though, is to use several moving-average

filters of different sizes, which, if they are single-stage, can use just one single buffer of input data. If their sizes are powers of 2, you can rescale the sums for the subtraction just by a bit shift of 1. In the limit, this takes the same 14 operations per sample per octave, but gets you much better low-pass filtering. Each filter will detect a separate frequency, which may or may not be the dominant frequency in its octave.

You could try to track the phase of the signal more closely than just within 90°, and this may be useful. However, the moving-average filter that provides the "derivative" signal imposes a somewhat arbitrary attenuation, which means that your phase velocity will vary throughout a cycle, perhaps wildly. However, the lag between crossing a given phase angle on successive cycles should be consistent.

Median filtering, PLLs, blah. Everybody uses variants of autocorrelation (ASDF and AMDF).

Topics

- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Magic kazoo (p. 3557) (3 notes)
- Proramming

Ultraslow radio

Kragen Javier Sitaker, 2013-05-17 (updated 2013-05-20) (26 minutes)
(Published previously on kragen-tol.)

Long-distance telecommunication is really important, and currently crucially dependent not only on the state of the global economy, but also on operationally centralized telecommunications companies, large telecommunications structures that are easy to bomb, and undersea cables. What's the minimal digital telecommunications infrastructure that would usefully enable intercontinental communication?

Possible communications modes

Shortwave radio

Shortwave radio (1.5MHz–30MHz) reflects off the ionosphere and can be received intercontinentally. This includes a couple of ISM bands in which unlicensed operation is permitted internationally: 13553–13567 kHz (22 meters) and 26957–27283 kHz (11 meters). Wikipedia's "High frequency" article says, "The maximum usable frequency regularly drops below 10MHz in darkness during the winter months, while in summer during daylight it can easily surpass 30MHz." Wikipedia's "Skywave" article says, "Signals of only a few watts can sometimes be received many thousands of miles away as a result." One hop off the ionosphere gets your signal up to 3500km.

This means that, in theory, you could set up a single unlicensed radio station providing communications services to you in most of a 7000-kilometer-wide circle. Something over a dozen such receivers could provide you with global coverage. Of course, you'd have to deal with interference: both decoding the received signals in the face of interference, and not generating so much interference to other users of the spectrum that they start to interfere with you in ways other than radio communication.

(Transmitting in the 22-meter band, especially using time-domain radio with low-power code division multiplexing and a low duty cycle, would also make it practically difficult to locate the radio transmitter; and of course the receiver would produce even less evidence of its presence.)

Possible bit rates for shortwave

Shannon's Theorem, aka the noisy-channel coding theorem, has no minimum signal strength below which communication bandwidth is zero; the Shannon-Hartley limit is B log (1 + S/N), where B is bandwidth in hertz, and S and N are the power of the signal and noise. In the "power-limited regime", when S/N is very small, this is linear in S/N. A somewhat counterintuitive result here is that, practically speaking, your bit rate doesn't depend on frequency selectivity much at all --- if you use twice as much of the spectrum, you double B, but you also probably double N. This means you can choose your bandwidth to satisfy other considerations, such as minimizing interference with other users of the band, who are mostly still using frequency-division multiplexing, or minimizing device cost.

So, supposing you can send one bit per second in the HF band using one watt over a few thousand kilometers, you can probably send about one bit per 1000 seconds using one milliwatt, or one bit per day using 12 microwatts, and this regardless of whether your signal is 0.1kHz wide, 14 kHz wide, or 28.5 MHz wide.

Is one bit per second about right? Digital Radio Mondiale in the shortwave bands uses at least 6 kilobits per second, but that doesn't tell us much, since this could be slammed through by using hundreds of kilowatts. What we really need to know is how much background noise there is in these frequency bands, in an absolute sense, or how many words per minute people can send and receive with CW Morse code over what distance and what power. One bit per second would be about 1.2 "words per minute".

It seems like the keyword for this kind of thing in current amateur radio operation is "QRP operation", which means around 5 or 10 watts or less. QRPers consider anything below 1 watt as "extremely low power", or QRPp, and they consider a thousand miles per watt as a difficult benchmark to meet. PSK31 is the digital code most commonly used at present for this kind of thing, and it operates at 31 baud and a bandwidth of 31.25Hz; it does a 180-degree phase shift during an amplitude null to represent a zero bit after encoding the digital signal into Varicode, a huffman code for ASCII. There are 10-baud and 5-baud variants in use. PSK31 is commonly used down to the 160-meter band.

If we figure that talking a thousand miles (1609 km) on a watt already puts you into the power-limited regime, and guess that you can get some 10 bits per second out of it, then using skywave bounce, you should be able to talk N thousand miles at 10/N bits per second. For example, you should be able to talk 16000 km, about a third of the way around the world at one bit per second.

Lasermoon moonbounce communication

You could illuminate the moon with a rapidly modulated laser at night. The moon's albedo is about 12%, it covers about half a degree of arc, and you can get cheap 5mW semiconductor laser pointers with divergence of less than that, indeed closer to a tenth of a degree, so that all of their power falls on the moon. Similarly, an inexpensive lens focusing the moon's image onto a photodiode (which is faster than a phototransistor) can easily give you better than an arcminute of selectivity. A square arcminute is about 85 nanosteradians, so this corresponds to about 80 dBi of "antenna gain" on the receiving end, but focusing more tightly than the original laser pointer spot isn't useful, so you only get about 60 dBi in practice.

Against this you must weigh the fact that most of the light falling on the moon is reflected off into space. You can probably reasonably use a one-meter-diameter plastic Fresnel lens, but you're about 380 megameters from the moon. This means that the light signal you get is about 176 dB weaker than the light signal sent from the moon, which is presumably most of the 500µW or so of light emitted from a 5mW laser pointer, minus about 9dB for the low albedo. This means you're trying to detect a signal of some -188dBm, which is difficult but feasible. It's actually considerably more feasible than doing the same trick with radio waves, due to the higher "antenna gain".

I think common laser diodes have a Q factor around 1000 --- their

light is mostly contained in a range of wavelengths of less than a nanometer --- which means you could improve your SNR by up to about 30dB by filtering the moon's image through a dichroic filter.

This leaves the question of how much noise you're dealing with in the first place, particularly important if you might have significant "crosstalk" from brightly-illuminated parts of the moon, say because your camera is imperfect. Sadly, I have very little idea how bright earthshine is, which is kind of the question.

Earth-mode communications

Ultra-low frequency radio --- 300 Hz to 3000 Hz --- can propagate through the earth itself; there are apparently lots of NATO papers from the 1960s about using this for military purposes. As far as I can tell, though, this involves running antenna cables a distance comparable to half the wavelength (100 km or more) or to the distance you want to communicate. So I think this may be impractical.

Balloon radio

Getting a balloon to 90km altitude is straightforward, at which point it has a line-of-sight radio path to anything within 9.57 degrees of arc (acos(r/(r + 90 km)), where r is 6371 km, the radius of the Earth) on the Earth's surface, about 1000 km; at this point it could quite reasonably communicate at gigabits per second using laser or highly-directional ultrawideband microwave to any of those points, if it can figure out how to align its transmitter and receiver. Furthermore, two such balloons will have a line-of-sight if they're less than twice the angle apart, or 2000 km. Twenty such balloons could ring the earth along a great circle, so a frequency-four geodesic sphere would suffice to provide global coverage with a bit under 200 balloons.

On the other hand, getting balloons to stay up there would be quite a challenging project. In fact, just getting them to work at all while they're up there is no walk in the park.

Air mail

In 1998, Laima, an Aerosonde, flew successfully across the Atlantic on 5.7 liters of fuel in 27 hours. Aerosondes weigh about 13 kg, about the same as a large swan. One micro-SD card weighs about 500mg and can currently hold 64 GB of data, so a 2-kg payload would hold 256 TB. This would amount to a 21 Gbps transatlantic data connection.

In 2003, the 5kg Spirit of Butts Farm made more or less the same 3000km flight on under a liter of fuel, in just under 39 hours.

On one hand, these bandwidths are orders of magnitude higher than the bandwidths reasonably achievable by the other technologies mentioned above. On the other hand, the latency is also some five orders of magnitude higher, and also, this is a very unreliable way of transmitting data --- it is likely to result in the loss of the aircraft and its payload on a regular basis.

A lower-tech approach for north-south communication would be to attach high-capacity store-and-forward radio-communication nodes to long-lived migratory birds. When they came into communication range of other nodes, they would exchange messages via short-range, high-bandwidth radio.

Neutrino beams

Particle accelerators can generate large quantities of neutrinos in a beam, by accelerating large quantities of pions or kaons to relativistic speeds and letting them decay; if the neutrino beam is directed at the earth, most of them will penetrate to the other side. Unfortunately, detecting a change in neutrino flux is very difficult, because they can easily penetrate things like the earth. It's feasible using thousands of photomultiplier tubes and tens of thousands of tonnes of water or other transparent liquid deep underground, like the Super-KamiokaNDE and the Sudbury Neutron Observatory; such observatories see on the order of one natural neutrino decay per few days per thousand tons of matter included.

This is clearly a feasible technique for communication, but it costs tens to hundreds of millions of dollars (which is to say, hundreds to thousands of person-years of work) and the result should be communication with bandwidth measured in fractional bits per day. If it were possible to increase the number of neutrinos by several orders of magnitude, you could increase the bit rate substantially, and in the limit, you could have lower latencies for long-distance communication than communication methods for which the earth is an obstacle.

Sound in air or water

Low-frequency sound can propagate for hundreds of kilometers in air, which is the principle behind the West African talking drums, which were used for long-distance digital communication for several centuries, up to the mid-20th century. I think you can get up to tens of bits per second this way. The talking drums could transmit several kilometers (8 km, says Wikipedia), but not tens or hundreds of kilometers. Whistled speech can cover one to two km, and has in theory a higher bit rate.

High-frequency sound is unusably attenuated by air in a few kilometers, but can cover thousands of kilometers in water, where whales use it; it also travels above 1km/sec. There's the additional advantage that sound in the ocean tends to remain a few hundred meters from the surface, in a zone called the SOFAR zone or Deep Sound Channel, so its intensity falls off as $1/d$ instead of $1/d^2$ over long distances.

Telegraph towers and cloud-bounced signal lamps

A 1.6-km-tall mountain, such as Sandia Peak in Albuquerque, has a line of sight to some 140 km distance on a smooth-spheroid earth, and perhaps 100 km in real life. Even a 100-meter-tall tower can see for 36 km. This principle was used in semaphore telegraphs throughout Europe in the decades before the development of the electric telegraph: a line of fortified stone towers at 30-km intervals provided about one bit per second. If you put a cheap laser pointer on the tower, you can transmit over this line of sight at megabit speeds or better.

Of course, you can also use a reflecting mirror to reflect sunlight instead of a laser; such a "heliograph" can transmit up to 300km under ideal conditions or 50km under normal conditions, using small, hand-sized mirrors. Bit rates depend on how fast you can open and close some kind of "shutter", but at least hundreds if not thousands of

bits per second should be attainable with inexpensive LCDs.

As suggested in the Lasermoon section, you can do this same kind of long-distance communication even without going up on the mountain yourself; it's sufficient to illuminate it from afar. Naval communication with Aldis lamps uses this trick to communicate by illuminating the bases of cloud formations, which enables communication at much greater distances; in theory, you could use cirrocumulus or cirrostratus clouds at their altitude of some 5–12 km to communicate to a station 500–800 km away, or perhaps even the 23-km-high tip of a cumulonimbus cloud to communicate with a station 1000 km away. This is more practical than using the moon, since you only have to detect light diffusing from the cloud 500 km from you instead of 380 000 km from you, giving you 60 dB less path loss in the cloud-to-you path than in the moon-to-you path. (As before, I think you can focus light onto the cloud well enough that effectively all the light you transmit reaches the cloud, so you effectively don't have any path loss.)

Useful bit rates

Why would it matter if you could transmit a millibit per second? Well, one bit can still be a crucially useful thing to be able to transmit. "I'm alive," for example, or "The killers weren't al Qaeda." But even a Tweet is only 1120 bits, so a bit per 1000 seconds would let you you transmit a Tweet every two weeks. The Annals of Spring and Autumn, 春秋, covers 541 years in 16000 characters; if each were 16 bits, that would be just over one bit per day on average. The Chappe semaphore line system transmitted about one "symbol" per minute out of 92 possible symbols, giving a bandwidth of about 0.11 bits per second; it was credited with giving Napoleon a decisive military advantage.

One bit per second

Consider the higher bandwidth of one bit per second, which I've hypothesized above requires about a watt of power over transcontinental shortwave. Two bits is roughly one letter of compressed ASCII text [0], so this is about 7000 words of text per day; one to ten Wikipedia articles per day, for example, or a fairly in-depth summary of daily international news, or a couple of hours of writing email per day, or a book per month. Furthermore, it's fast enough to permit someone to write a line of text in a chat system, transmit it, and get back a response from someone else within a couple of minutes.

[0] bible-pg10.txt is 824 146 words and 4 452 069 bytes; its .bz2 is 999 906 bytes, 1.80 bits per uncompressed byte; compressors other than bzip2 may be more efficient; but extremely high compression ratios may only be achievable with the transmission of large amounts of data. For example, if I split the same document into 64KiB chunks and bzip2 each one independently, the result is 1 214 055 bytes, 2.18 bits per uncompressed byte.

One bit per second, then, is a sufficiently rapid connection to permit real participation in the intellectual life of an invisible college; but it's less information than a person can read. Furthermore, with a broadcast medium such as radio, it's reasonable to transmit the same information to many recipients at a time, Usenet-style.

Commercially speaking, even one bit per second, or for that matter a thousandth of a bit per second, at low latency is enough to make you a hell of a lot of money in commodity arbitrage, if your competitors don't have access to equally fast communication. A price to three significant figures is only ten bits, and it's straightforward to encode names for widely-traded commodities in 30 bits or less. (This is why the stock ticker was such big business in the late 19th and early 20th century.)

Ten bits per second

Ten bits per second would give you some 70 000 words of text per day, which is almost as much as you could read in real time; and it would be fast enough to permit real-time hypertext navigation with a stateful two-way protocol. (If you're transmitting document contents containing the IDs of other documents in one direction, while transmitting the IDs of requested documents or document parts in the other direction, each ID will typically be within the last thousand IDs transmitted, so it can be encoded in ten or twenty bits.) Ten bits per second at 2.5 bits per character works out to 240 characters per minute, or 48 words per minute, so it's at least theoretically possible for one person to type faster than ten bits per second on a sustained basis; but it's fast enough that one-to-one text chat would be more or less real time.

A hundred bits per second

A hundred bits per second would give you some 700 000 words per day, an entire book. This is enough that one person can no longer read all of it, if it's text, and eliminates the need for a stateful protocol for real-time hypertext --- although full HTTP might still be excessive. It's about 480 words per minute.

Acquiring one new book per day was an unachievable dream for nearly all individual scholars and the majority of libraries in most of the age of print. If the books were new cheap paperbacks, it cost about US$1800 per year toward the end of the 20th century, within the means of most of the population of the developed world, but new hardbacks pushed the cost up by an order of magnitude, to US$18000 per year. So a hundred-bit-per-second connection to a large data store could beat access to printing presses for all but the wealthiest.

If one person were trying to consume a hundred bits per second, they would probably want to do much of it in the form of still images. A high-quality color JPEG file typically uses about four bits per pixel, and black-and-white needs about half that. So a hundred bits per second is on the order of 100 low-quality pixels or 25 high-quality pixels per second. A low-quality 320×200 color image might download in 21 minutes.

As of early 2012, the English Wikipedia articles without pictures total 10 399 030 471 bytes in a compressed Zim file. At a hundred bits per second, this file would take 26 years to transmit.

For much of the twentieth century, most Telex systems ran at 45.45 bits per second.

A thousand bits per second

A thousand bits per second is 7 000 000 words per day, some ten books. At this speed it's possible to receive text you don't intend to read yourself, just in case you want to read it later; but it's also fast

enough that you don't need to if the connection is reliable. You could download the Wikipedia archive I mentioned above in 2.6 years.

It's also at the minimal lower limit where you can do real-time voice transmission. Many voice codecs (from early LPC work in the 1970s to HVXC and Speex today) have modes between 2 and 3 kilobits per second. I think transmitting intelligible speech at 1kbps is feasible, but only barely, and I don't know if it's been done.

At a thousand bits per second, you can completely kick the ass of printing-press-based communication systems, both providing instant hypertext access to faraway data stores and accumulating the equivalent of thousands of locally-stored books per year, plus, of course, email.

Ten thousand bits per second

This is the speed at which much of Fidonet operated.

Possible communications hardware and algorithms

The very low power levels involved suggest that it might be possible to do these ultraslow transmissions with extremely steampunk equipment, or not even that. For example, for radio communication, perhaps you could trigger a static-electrical spark from a Wimshurst Influence Machine at intervals driven by holes in a rotating glass sphere, firing an electrical impulse into a dipole antenna. But a Wimshurst machine can produce more than a watt of power; a simpler generator, such as your hair and a balloon, might be sufficient. You just have to time your sparks correctly to make them interpretable by your (presumably much more capable) receiver, and successfully couple them into your antenna, maybe with a low-Q resonator containing a coil and a Leyden jar.

(In theory, maybe you could decode such a low-bandwidth signal by hand, too, but it's not obvious to me how you'd do it.)

(The earliest non-spark-gap radio, the Poulsen arc, used a carbon arc as a negative-resistance device to produce a dynatron oscillator; I think you could also use just about any gas discharge tube, such as a fluorescent lamp tube or even a fluorescent lamp starter.)

Of course, multiple ultraslow radios could form a very slow packet-switching network.

Moving further into the steampunk and cargo-cult direction, you could imagine a shorter-range purely passive radio transmitter, like a sort of switchable analog RFID tag. The "receiver" would wait for the transmitter to be energized by electromagnetic pulses produced by nearby lightning strikes; then it would measure a variable set of RF resonances in the device. A Leyden jar might have a nanofarad of capacitance, and a variable inductor with up to about a henry of inductance can be fairly easily constructed by hand, the two forming a tank circuit with a resonant frequency of as little as 5kHz and a Q of perhaps 100; a large number of such resonant circuits connected to an antenna will "ring" for tens or hundreds of oscillations when subjected to a radio impulse. If their frequencies are chosen in accordance with some kind of error-correcting code, a distant detector will eventually be able to determine what those frequencies are, after detecting enough impulse responses. Multiple receiving antennas could provide

enough phase diversity to do beamforming.

Such a device is fairly inefficient, since most of the energizing pulse energy will not coincide with a resonant frequency of the transmitter and will thus simply reflect from the antenna. If you could store the pulse in some kind of device that allowed you to use it to continuously energize the resonators until it dissipated, you could use more of it. For example, with a semiconductor diode, you could store much of the pulse in a capacitor, and use that to power active oscillator circuits; or if you could circulate it through some kind of delay line, such as a piezoelectric acoustic delay line, you could use any nonlinear but nondissipative circuit element to shift some of the remaining pulse energy into resonable harmonics or subharmonics on each circulation through the system.

Pulse radio would allow for relatively reasonable decoding electronics and algorithms: while your average power might be, say, 100 milliwatts, if you're only transmitting for 100 microseconds out of every minute, that transmission would be at 60 kilowatts. (If it's a single 100-microsecond pulse, it would be a six-joule pulse; into a 63-ohm antenna impedance, it would be a momentary current of 31 amps, driven by 2000 volts, which could be stored in 3 microfarads, so this wouldn't require any particularly exotic circuit design.) Detecting the pulse at even a very substantial distance should not require particularly sensitive or highly linear analog electronics; the remaining task, then, is to figure out which subset of the detected pulses represent a decodable message.

Topics

- Physics (p. 3632) (119 notes)
- Independence (p. 3520) (63 notes)
- Communication (p. 3382) (19 notes)
- Information theory (p. 3524) (9 notes)
- Aerosonde

Cassette tape capacity

Kragen Javier Sitaker, 2018-04-27 (1 minute)

David Rowe's Codec 2 packs relatively comprehensible voice into 700 bits per second, for use in the FreeDV amateur radio low-bandwidth voice mode. But what if you record it on an audio cassette tape instead? You should be able to get higher bandwidth than we got out of analog telephone lines — two channels with an ENOB of about 5 with frequencies up to about 15kHz, implying 30ksps. This works out to a channel capacity of about 300 kilobits per second, given the appropriate modulation. (And doing the appropriate modulation should be pretty feasible.)

This implies that each second of audiocassette data can hold about 7 minutes of Codec-2-encoded voice; a 60-minute audiocassette can hold some 400–500 hours of recorded voice.

Fortunately or unfortunately, this is not actually a useful thing to do.

Topics

Binary translation register maps

Kragen Javier Sitaker, 2017-07-19 (1 minute)

Reading Sorav Bansal's dissertation, I was struck by the fact that in the middle of the binary-translation section, he tackles the register allocation problem using the Viterbi algorithm, although he seems not to have realized that he was solving register allocation (conventionally considered NP-hard) or that his solution was the Viterbi algorithm.

The context is not conventional compiler code generation but rather binary translation from PowerPC code to 386 code, so both the input and output of his system are sequences of instructions. He is faced with the question of how to assign registers for the output instructions.

His solution is to compute the cost of all possible register maps, one input instruction at a time, as he walks through the input code, adding an extra "switching cost" when the predecessor state used a different register map; but he retains only the lowest-cost few maps at any given point in order to keep the cost reasonable, up to about 8 maps, though he only gets a significant performance advantage up to about 3 maps. (I wonder if this is due to having several maps that are essentially equivalent.)

This seems like a remarkably simple approach to a remarkably difficult problem, so simple that I am led to wonder whether it actually works. Some other aspects of the scholarship in the dissertation are shaky.

Topics

- Algorithms (p. 3310) (123 notes)
- Compilers (p. 3383) (16 notes)
- Viterbi

Atmospheric pressure harvesting phoenix egg

Kragen Javier Sitaker, 2018-11-23 (14 minutes)

I think I've finally found the problem to how to refresh the memory in the Egg of the Phoenix: use daily tidal atmospheric pressure swings to power programs to periodically rewrite NAND Flash.

The memory retention problem

The problem is that the device, by definition, needs to be able to retain its memory when buried for a century or more; that's its entire raison d'être. But modern Flash is not designed for such a requirement; instead, it's designed for memory retention of 20 or even 10 years. And it's very likely that Flash memory will gradually lose its memory as charge tunnels away from its floating gate, at a rate which is IIRC exponential in temperature.

It doesn't take very much energy to refresh Flash; Low-power microcontrollers for a low-power computer (p. 2602) says the Adesto AT25SF041-SSHD-T NAND chip requires 100 mA for 500 ms to erase 32 kB ≈ 500 nJ per byte, and presumably some smaller amount to write the erased memory; Keyboard-powered computers (p. 2220) says Flash uses 2 µJ per 32-bit write, which is exactly equal. FeRAM also exists and is about 2000 times more energy-efficient than Flash, and MRAM is apparently about 500 times more energy-efficient than Flash, but both are dramatically more expensive.

(Other possibilities besides Flash, FeRAM, and MRAM exist, but nothing that we can confidently predict will survive a century.)

The power needed is thus proportional to the amount of data archived.

Energy requirements; why not just use a battery?

This means that preserving, say, 100 gigabytes of Flash requires about 50 kJ every 10 years or so, which works out to an average of 160 µW. If we take that over 100 years, it's 500 kJ; converted to milliamp hours at the 3.7 V of a lithium-ion battery, it's 37000 milliamp hours. This would be a very reasonable-sized lithium-ion battery, one you could hold in your hand. So why not just use that?

Well, all batteries have a self-discharge rate, and typically it's enough to discharge them completely within five years. Lead-acid batteries have a somewhat longer shelf life, perhaps 10 or 20 years. But no commercially made battery has a shelf life of a century.

One possible alternative is a Zamboni pile such as, it is believed, the Clarendon Dry Pile that powers the Oxford Electric Bell, which has been ringing almost continuously since 1840, on about 1 nA and 2000 V (2 µW). Unfortunately, although long-lived Zamboni piles do exist, they are not very well characterized; we don't know very much about the ways they can fail, particularly after decades. We don't

even know for sure that the Clarendon Dry Pile is in fact a Zamboni pile. And, as far as I know, nobody has ever built a Zamboni pile that yields tens of microwatts, let alone hundreds.

Atmospheric pressure variation

Earth's atmospheric pressure varies tidally by a few millibar (a few hundred pascals) around its average of 101.325 kPa at sea level, varying from 87.0 kPa (during Typhoon Tip in 1979) to 108.48 kPa (during the Siberian High in Mongolia in 2001). (The vertical variation is larger than this, at about 11.3 Pa/m, so reaching the record low typhoon sea-level pressure only requires going a bit over a kilometer up.)

As I write this, the pressure is some 103 kPa, and tomorrow it's forecast to fall to 102 kPa, then as low as 100 kPa the next day as a rainstorm comes through. For most of the next week, the trend is almost perfectly flat, staying between 101 and 102 kPa. There are tidal-looking wiggles in the forecast of about 100 Pa above and below the trend line. The NWS says Boston currently has 101.51 kPa, but earlier today it's been as low as 101.05 kPa and as high as 101.62 kPa; yesterday was from 100.91 kPa to 101.56, with a bit of rain. So, although the dependable tidal variations are only around 100–200 Pa, working out to a total of 400–800 Pa per day absolute change, it's common to have a bit more than that, like 1 kPa or more per day of change in one direction or the other.

Common soils and even rocks are not sufficiently rigid and impermeable to prevent this pressure variation from reaching underground, even to a depth of tens of meters, a phenomenon known in some contexts as "barometrically induced variability". So an energy-harvesting device that harnesses these tidal pressure variations will work even if it's buried, as long as it doesn't get all crudded up with sand or something. Even if it's filled with water, it should continue to work.

Such energy-harvesting computational devices date back to the seventeenth century; one of the greatest engineers of all time, Cornelis Drebbel, built several clocks powered by such pressure changes at that time, starting in 1610. Cox's Timepiece, built in the 1760s, ran exclusively on atmospheric pressure until being acquired by the Victoria & Albert museum in 1961, and the Beverly Clock, built in 1864, is still running today, 154 years later, though it also harvests thermal energy.

Jaeger-LeCoultre currently sells the Atmos clock, which harvests energy from temperature and pressure variations to run without winding; some half a million have been produced so far since their 1929 début. Although, like the Beverly Clock, it primarily harvests thermal energy, a pressure variation of 3 mmHg = 0.4 kPa is sufficient to energize it for two days; although typical tidal pressure variation is slightly smaller than this, it shows that pressure variations in this range are suitable for harvesting even with a tabletop-sized device with no electrical parts, by pressing an ethyl chloride bellows against a spring. Michael P. Murray, a specialist in Atmos clocks, claims that they use about 250 nW; Adam Sacks claims they should have a service life of about 600 years.

Underground pressure penetration

Aside from the few hundred pascals of tidal variation, when it rains, the soil may saturate with water and go to higher pressures. If the Egg is buried 2 meters deep, for example, it will experience an extra 20 kPa of hydrostatic pressure if the water table rises past it to the surface. This is some two orders of magnitude larger than daily tidal variations, but only somewhat larger than the 15 or so kPa of difference between world records. So it's important to design the Egg to withstand such pressures, even if it can't harvest the energy they present.

The surface tension of water in the soil can also produce soil suctions up to some 30 MPa, which I interpret as pressure below atmospheric pressure and indeed below zero pressure, despite cautions that that's not exactly what is meant. Apparently typical soil suction for agricultural soils is 25 kPa or less.

"Barometrically induced variability" is a term used for variability in subsurface gas concentrations, of concern in monitoring of toxic waste. One paper reported pressure variation over a range of some 25 mb (2.5 kPa) during a few days, tens of meters below the soil surface, which was inversely correlated with CCl_4 concentrations.

Even higher-frequency pressure variations, such as those caused by wind vortices, can penetrate deep underground; Takle and colleagues in 2004 measured about 6 dB of attenuation of 2 Hz pressure variation by 600 mm of soil, if I'm reading the paper right, which suggests that the ≈20 μHz tidal pressure swings should be able to penetrate many, many meters of soil with no real attenuation.

Power

How much power is available from these air pressure variations? It seems like the limit is somewhere around 100 μW/ℓ, but it may be hard to approach that limit.

Suppose we have a cubic decimeter of air (a liter) with one flexible or bellowed face, to which we couple some kind of harvesting device, piezoelectric or mechanical or whatever. Suppose the air pressure changes by 200 Pa. If that device provides no resistance to the pressure variations, then the volume of the cube will increase and decrease by 0.2%, which is some 200 μm. But this is doing no work on the device, because there is no force. At the other extreme, suppose that the harvesting device is absolutely rigid. Then the force on it will vary from +1 N to -1 N, but the volume of the cube will not vary, and so it will do no work on the device, because there is no movement.

In between these extremes, we can harvest energy. For example, suppose that the harvesting device will rigidly resist a maximum of 0.5 N and thereafter move, winding a spring or something at constant force. Then the flexible face will move by only 100 μm, but it will be exerting 0.5 N over that distance, providing 50 microjoules, about every 6 hours. This works out to about 2 nW, which is two orders of magnitude smaller than what Murray claims the Atmos clock runs on. And the Atmos clock bellows does not appear to occupy an entire liter. We can perhaps conclude that using air, as the Beverly clock does, is quite disadvantageous.

Dividing the cube into slices to get more faces does not help, because the slices will expand and contract by correspondingly less distance. Making the cube larger does help, as the energy available is

proportional to the expansion distance and piston area.

What's the maximum work we could potentially extract from air pressure on our liter? Suppose it's still a cube, but filled with vacuum and with four bellows sides, with a constant-force spring pulling on it producing 1013.2 N. Now, whenever the air pressure falls below 101.32 kPa, the spring will expand the cube to its full 1ℓ volume, and whenever the air pressure rises higher, the spring will contract the cube down to a little square pancake, which we can suppose has insignificant volume compared to 1ℓ. The work that can potentially be extracted is a function of the height of the peak at the point that we allow the harvesting device to move; supposing that, as before, it's 200 Pa from trough to peak, we have 2 N over 100 mm, which of course yields 200 mJ. This is a quite respectable 9.3 μW, though still considerably smaller than the 160 μW we need. And, of course, if we somehow know that the pressure will fall from 103 kPa to 102 kPa tomorrow, then we can leave the cube fully extended until the pressure reaches its minimum, and then extract 10 N over 100 mm, a total of 1 J, some 100 μW over 24 hours.

(And of course we can use multiple liters to multiply the power; use a longer, thinner cylinder to get a longer stroke at proportionally less force; or use a fatter, flatter cylinder to get a shorter stroke at proportionally more force.)

Note that 9.3 μW is some 4000 times larger than the 2 nW we'd get from the air cube.

How close could we come to this ideal in practice? Presumably the Atmos uses ethyl chloride because the pressure range is enough to cause much of it to condense and evaporate at room temperature, enabling much larger volume variations than an air tank could manage. At a given temperature, the equilibrium pressure within such a mixed-phase, single-material container is absolutely constant at the vapor pressure of the liquid at that temperature, so a liquid whose vapor pressure is 101.32 kPa at whatever your temperature is would actually behave precisely like the vacuum-and-constant-force-spring gedankenexperiment above, expanding to an almost arbitrarily large volume when the pressure drops, and contracting to almost none when the pressure rises. (For water, the liquid to vapor density ratio is a few thousand to one; it's a bit lower for ethyl chloride.) Unfortunately, that experiment assumes that the tidal swings center precisely on that known pressure, which, as described above, is far from true; day-to-day shifts are commonly several times the amplitude of the dependable tidal pressure swings.

It isn't yet obvious to me how, but I suspect that something close to the ideal behavior described above is feasible. So only a few liters (perhaps less than the 28 used by the Beverly Clock) should be necessary to provide the refresh power. I think the Atmos compromises its ability to harvest energy at any given pressure/temperature combination by using its "counterweight" Hookean spring to spread its response curve over a wide range of pressures and temperatures in a purely static fashion. That is, at a given pressure and temperature, I think the Atmos's expansion chamber will always have the same volume, plus or minus a small constant bit of hysteresis.

Efficiently converting the mechanical power to electrical power is, I think, the easiest part of the problem; piezoelectric,

electromagnetic, and electrostatic techniques are all applicable, though it seems likely that the fairly rigid piezoelectrics are the best fit for the likely small displacements involved.

Topics

Observable transaction possibilities

Kragen Javier Sitaker, 2019-06-15 (10 minutes)

I was reading about Observablehq last night. They've layered a dependency and auto-recalculation system on top of JS, so that any cell of your code is re-executed when its dependencies change, without requiring you to explicitly reinvoke it.

I should probably dig into what Observable can do, or actually I should work on some DSP code I've been procrastinating on, but instead I am going to go off and natter about vaguely related stuff.

By executing the sequence of code in a cell, such a system can discover which inputs it's currently reading from and which outputs it's currently writing to, although both of these could change depending on the values of those inputs. Observable in particular determines the outputs statically and doesn't let different cells write to the same variable.

Suppose in such a system you have a cell that computes

```
r = (x**2 + y**2)**0.5
```

You can interpret this in the standard way, as a "statement", which is to say an instruction, which, when executed, reads single values from `x` and `y`, do a computation, and write the single result to `r`. Numpy extends this, following APL, by allowing `x`, `y`, or both, to contain *values* which are arrays of *numbers*. For example:

```
>>> x
array([-1.  , -0.75, -0.5 , -0.25,  0.  ,  0.25,  0.5 ,  0.75,  1.  ])
>>> y
array([ 1.73205081,  1.85404962,  1.93649167,  1.98431348,  2.        ,
        1.98431348,  1.93649167,  1.85404962,  1.73205081])
>>> r = (x**2 + y**2)**0.5
>>> r
array([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
>>> y = 1
>>> r = (x**2 + y**2)**0.5
>>> r
array([ 1.41421356,  1.25      ,  1.11803399,  1.03077641,  1.        ,
        1.03077641,  1.11803399,  1.25      ,  1.41421356])
```

In addition to allowing `x` and `y` to either not vary or to vary *together*, we can also have them vary *independently*, producing a *matrix* of results for `r`:

```
>>> y = x.reshape((9, 1))
>>> y
array([[-1.  ],
       [-0.75],
       [-0.5 ],
       [-0.25],
       [ 0.  ],
       [ 0.25],
       [ 0.5 ],
```

```
       [ 0.75],
       [ 1.  ]])
>>> r = (x**2 + y**2)**0.5
>>> r
array([[ 1.41421356,  1.25      ,  1.11803399,  1.03077641,  1.        ,
         1.03077641,  1.11803399,  1.25      ,  1.41421356],
       [ 1.25      ,  1.06066017,  0.90138782,  0.79056942,  0.75      ,
         0.79056942,  0.90138782,  1.06066017,  1.25      ],
       [ 1.11803399,  0.90138782,  0.70710678,  0.55901699,  0.5       ,
         0.55901699,  0.70710678,  0.90138782,  1.11803399],
       [ 1.03077641,  0.79056942,  0.55901699,  0.35355339,  0.25      ,
         0.35355339,  0.55901699,  0.79056942,  1.03077641],
       [ 1.        ,  0.75      ,  0.5       ,  0.25      ,  0.        ,
         0.25      ,  0.5       ,  0.75      ,  1.        ],
       [ 1.03077641,  0.79056942,  0.55901699,  0.35355339,  0.25      ,
         0.35355339,  0.55901699,  0.79056942,  1.03077641],
       [ 1.11803399,  0.90138782,  0.70710678,  0.55901699,  0.5       ,
         0.55901699,  0.70710678,  0.90138782,  1.11803399],
       [ 1.25      ,  1.06066017,  0.90138782,  0.79056942,  0.75      ,
         0.79056942,  0.90138782,  1.06066017,  1.25      ],
       [ 1.41421356,  1.25      ,  1.11803399,  1.03077641,  1.        ,
         1.03077641,  1.11803399,  1.25      ,  1.41421356]])
```

As I wrote in A principled rethinking of array languages like APL (p. 1995) and Index set inference or domain inference for programming with indexed families (p. 1434), you could think of the broadcasting rule for these simple elementwise operators as a logical interpretation: *if* x = -0.75 *and* y = -1, *in that case* r = 1.25; but *if* y = 0, *in that case* r = 0.75; and so on. And given a reactive observable dataflow transaction system like Observablehq, you could in fact evaluate it that way — if you change the value of `y` from -1 to 0 without changing `x`, `r` will react by changing its value from 1.25 to 0.75.

A FIR filter example

How about a FIR filter?

```
y = sum(w[i] * x[-i] for i in range(len(w)))
```

This is a valid Python statement, applying the time-domain FIR kernel `w` to the last few values of the list or array `x`. If you run it every time you append a new value to `x`, it will put successive samples of the FIR filter result into `y`. You have to be careful to not run it before `x` has enough values in it, and you probably want to do something with the result in `y` before it gets overwritten by the next execution, because the result is a function of a time-varying state of the system.

Suppose instead that you want to compute the entire FIR-filtered signal rather than just a point on it. You can do such a computation in Numpy in at least four different ways. The way you would actually use in practice is the `convolve` function:

```
y = numpy.convolve(w, x, 'valid')
```

Unfortunately this doesn't throw any light on how to solve the problem; it just delegates it to a library function, which, as it turns

out, delegates to the `multiarray.correlate` function, which is written in C and presumably does a nested loop.

The other two straightforward ways involve an interpreted loop in Python (which is why you wouldn't want to use them in practice) which invokes an inner loop via a Numpy SIMD operation. You could have the implicit inner loop compute a single point of `y`:

```
y = [(w[::-1] * x[i:i+len(w)]).sum() for i in range(len(x) - len(w) + 1)]
```

Or you could have the implicit inner loop calculate the contribution of a single sample of `x` to all the samples of `y`:

```
y = numpy.zeros(len(x) + len(w))
for i in range(len(x)):
    y[i:i+len(w)] += x[i] * w
y = y[len(w)-1:1-len(w)]
```

The fourth approach is to transform both `x` and `w` into the complex Fourier domain, multiply the complex phasors there elementwise, and transform back into the spatial domain. This involves some subtle issues of numerical precision, and it's profoundly nonobvious, but for large signal vectors, it is by far the fastest method. Implicitly in the above, `w` is not longer than `x`, and in that case it looks like this:

```
wf = numpy.fft.fft(numpy.concatenate((w, numpy.zeros(len(x) - len(w)))))
y = numpy.fft.ifft(wf * numpy.fft.fft(x))[len(w)-1:]
```

How about in the "logical" view? `w * x` is a perfectly reasonable expression; if `w` has 5 possible values and `x` has 11, then it `w * x` has 55 possibilities, reasonably represented as a matrix (the outer product of `w` and `x`, considered as vectors). It's indexed by the Cartesian product of `w`'s index set and `x`'s index set. The `y` computed above is just a sum over some of those values; the summation introduces a dummy variable that ranges over the valid indices of `w`.

The conventional mathematical notation for this is

$$y_n = \Sigma_i w_i x_{n-i}$$

where the dummy variable i implicitly takes all the values that it would be coherent for it to take. This is somewhat ambiguous and often disambiguated by contextual information: if x_i only exists when $i \geq 0$, for example, does y_0 exist, being simply $w_0 x_0$? And resolving that ambiguity is what the `'valid'` in the above Numpy expression is for.

Typically we think of x and y here as being indexed by (discretized) time, but of course nothing in the math requires that; for math, the time is just another meaningless variable.

In the "logical" view, this introduction of the dummy variable i means that *any* value of y depends on *every* value of w. In the sort of ambient-indexing environment I was thinking of there, Σ considers a multiplicity of possible worlds with different values of w; in this case, we also want to use the *index* into w to reindex *time itself* (ominous music with minor chords!) so we can access values of x from other points in time. A reasonable way to write this in ASCII might look something like this:

```
y = sum(w.i, w * x[t = t-w.i])
```

Here `sum` takes two arguments: the index to sum over and the expression to evaluate in a possible world for *all* of the possible values of the index. `w.i` is the index set of `w`; as `sum`'s first argument, it's being used as *a reference to an index* that needs to have its index set inspected and iterated over, while within the expression, it's being used as *a reference to a particular value of that index*, a mere rvalue. `x` is being indexed by the specification `[t = t-w.i]`; this indexing creates another possible world in which to evaluate `x`, in which the index `t` (which, perhaps, might be `x.i` or a part of `x.i`) has a different value from its value in the outer environment. The expression `t-w.i` is evaluated in that outer environment, taking the ambient value of `t` and subtracting the current value of `w.i` from it. The indexing expression hides the outer ambient value of `t` from `x`, replacing it with the fake value.

This implies that `t` and `w.i` are of some types such that it makes sense to subtract them; they are not, for example, merely categorical or ordinal. They could, however, be vectors of the same dimensionality, such as 2-vectors, in which case the same expression above serves for convolving images as well as time-series signals.

However, the fact that we're iterating over all the possible values of `w.i` means that it cannot be a *continuous* dimension, although its values could be values of a continuous type. It must be discrete so that it can have a finite number of possible values!

The same ambiguity about out-of-range values is present. If one of the `x[t = t-w.i]` values happens to not exist, for a given ambient value of `t`, does that mean the entire sum fails to exist, and thus `y` has no value at that point? Or does it merely mean we sum over a smaller number of values?

Topics

- Digital signal processing (DSP) (p. 3419) (60 notes)
- Programming languages (p. 3656) (47 notes)
- Python (p. 3671) (27 notes)
- Arrays (p. 3326) (17 notes)
- SIMD instructions (p. 3711) (10 notes)
- APL (p. 3320) (9 notes)

Karatsuba

Kragen Javier Sitaker, 2019-04-20 (2 minutes)

$(aX + b)(cX + d) = acX^2 + (ad + bc)X + bd$

Here X is some power of the base of your number system, and this is the conventional algorithm for multiple-precision multiplication. This divides the problem of multiplying two numbers "ab" and "cd" into the problem of multiplying four pairs of numbers, each half as long; so it's a sort of recursive divide-and-conquer algorithm which, in the end, takes $O(N^2)$ time: for 2^i digits, you do i levels of divide-and-conquer, producing 4^i bottom-level multiplications, which is just the square of the number of digits. These multiplications are then combined in a smaller number of shifted addition operations.

Karatsuba came up with a different way to do this, computing (a + b)(c + d) = ac + bc + ad + bd. This contains the ad + bc sum we need as a couple of subterms. If we compute ac and bd, we can subtract them off to get ad + bc.

For example, 93 × 24: ac = 9×2 = 18; bc = 3×4 = 12; (a + b)(c +d) = (9+3)(2+4) = 12 × 6 = 72; 72 - 18 - 12 = 42. So our final result is 1800 + 420 + 12 = 2232, which is correct.

This has the advantage that, although the operations per internal node are slightly more complicated, instead of 4^i bottom-level multiplications you have 3^i. So, for example, if you have a 1,048,576-digit number, you need 1,099,511,627,776 bottom-level multiplications with the conventional algorithm, but only 3,486,784,401 with Karatsuba's algorithm,

Topics

- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Multiplication (p. 3590) (3 notes)

Commentaries on reading Engelbart's "Augmenting Human Intellect"

Kragen Javier Sitaker, 2018-12-24 (updated 2018-12-25) (25 minutes)

My commentaries on reading Engelbart's 1962 "Augmenting Human Intellect: A Conceptual Framework" for the first time, rather belatedly in 2018.

The date 1962 is rather crucial in understanding this work, because he talks a lot about computers, and computers were changing rapidly around this time. 1962 is the same year Sutherland (mentioned on p. 72) wrote Sketchpad, published in 1963, in which he invented graphical user interfaces, object-oriented programming, computer-aided design, and constraint-based programming, for example; and 1962 is the year IBM published their book on Project Stretch, edited by Werner Buchholz, which ran from 1954 to 1961. The first integrated circuits were fabricated around this time, the first real compilers were written in the years 1958–9, and the Lisp paper was published in 1959, based on an implementation on a vacuum-tube IBM computer. DEC had just been founded, and Spacewar! had just been written. The IBM 360 project, with its ambition to unify "business" and "scientific" computing on a single general-purpose computer, had not yet begun. Timesharing operating systems, like compilers, was an avant-garde technique — Stretch was designed for timesharing operating systems, but most computers did not use them, or indeed any operating system. Nowadays instead of "timesharing" we call it "multitasking".

(I am not sure when Expensive Typewriter, the first interactive word processor, was written.)

In short, this report was published immediately after the creation of the compiler, the high-level programming language ("problem-oriented language (e.g., ALGOL or COBOL)", p. 17), the computer game, the operating system, and multitasking ("what is known as 'time sharing'", p. 70), and immediately before the publication of the graphical user interface and object-oriented programming.

Moreover, this was three years after Sputnik, and airlines had just started flying jet airplanes ten years before, adding a new increment in human velocity to some 300 m/s (600 mph in archaic units), up from the 8 m/s that had reigned from the domestication of the horse until the introduction of the automobile seven decades earlier; supersonic jets were being flown by the military, but the Concorde wouldn't offer passenger supersonic service until 1976. In retrospect, the continuous, spectacular increase in human velocity Engelbart could look back on ended almost precisely in 1962, with the exception of a few dozen astronauts.

This was also the time period when full automation was being touted as an imminent end to work as such, before the difficulties of robotics were appreciated. Engelbart doesn't mention this directly, but his report is part of the same intellectual ferment that sprang up to

try to make sense of the possibilities of the new technologies.

Given this atmosphere of radical ferment, it's interesting that a significant part of Engelbart's paper (pp. 48–56) is dedicated to discussing "As We May Think", published in 1945, 17 years earlier; Engelbart quotes its description of the Memex in its entirety. I'm not sure when Engelbart came upon the paper (and I missed my opportunity to ask him) but that 17-year gap is notable. He cites Licklider's "Man-Computer Symbiosis", various list-processing things (including IPL-V and LISP 1.5, on pp. 65–66), and a couple of items in passing from Ross Ashby, but in nothing like the depth he dedicates to the Memex.

Engelbart's famous "Mother of all Demos" in 1968 was the result of six years of work at SRI attempting to realize the vision he set out in this 1962 report.

Are we augmented yet?

On p. 3, Engelbart explicitly cites the velocity progress I mention above as his antecedent:

> there is no particular reason not to expect gains in personal intellectual effectiveness from a concerted system-oriented approach that compare to those made in personal geographic mobility since horseback and sailboat days.

This leads me to ask, 56 years later, have we achieved those gains? Can I do intellectual things now that would have taken 150 people to do in 1962, or can I do things now in a day that would have taken me four to six months then?

In a few domains, I think I can. In purely calculational terms, I have balanced on my paunch a laptop capable of about a hundred billion multiplies per second; its RAM is 4 gibibytes with a memcpy speed of about 2 gigabytes per second (4 gigabytes/second or 32 gigabits/second unidirectional), and its SSD holds 100 gigabytes of data and can be read at 75 megabytes per second (600 megabits per second). Stretch's disk was 8 megabits per second, or 125 000 64-bit words per second, with a total size of 2 megawords (16 megabytes) and a seek time of 150 milliseconds (p. 20 of Buchholz), and it needed about 2.5 microseconds per instruction (p. 32 of Buchholz), some of which could be multiplies; its memory cycle time was 2.1 microseconds per 64-bit word (p. 17 of Buchholz), giving 32 megabits per second, with a maximum of 2^{18} words (1 mebibyte). Moreover, Stretch was shared by all of Los Alamos National Labs, which I think was several hundred people at the time, though perhaps only a few dozen of them had access to the computer.

(On pp. 66–67 Engelbart describes the capabilities of a typical computer of his day: 100,000 6-bit bytes of RAM costing 60¢–US$1.50 per byte, 2–10μs per machine instruction, with a megabyte-sized magnetic drum costing 5¢ per byte, or a hundred-megabyte-sized disk costing 0.14¢ per byte.)

So, roughly speaking, my laptop is 2.5 million times faster than Stretch at multiplying (and arithmetic in general), 1000 times faster than Stretch at accessing its RAM, of which it has 4096 times as much, and 75 times faster than Stretch at accessing bulk storage, of which it has 6000 times as much; and this computing power is distributed among about 1000 times fewer people.

As a consequence of this extreme arithmetic power, it can do things like real-time raytracing, software-defined radio, and speech

recognition. Mostly, however, this arithmetic is devoted to decompressing porn and other video. I don't have any speech-recognition software on here.

I *can* and occasionally do use it for tasks like 3-D design, and designing and testing signal-processing algorithms in IPython, and I have a library of many technical papers, including the Engelbart report I'm currently reading. Due largely to format incompatibilities, I don't have an easy way to search through the library for key words, although searching through an individual book takes only a few seconds. Quoting one document in another, as I did above, often requires manual retyping — evince's user interface sometimes doesn't respond to mouse drags, and it doesn't offer an interface to easily display an excerpt from a PDF or DjVu file, even though those file formats contain internal indices to permit random access.

I can easily prepare a document plotting a bunch of equations and simulations, complete with nicely-typeset equations, using IPython; TeX and HTML provide me with document-preparation capabilities Engelbart's whole team couldn't pull together in 1962, even leaving aside the ability to distribute the results immediately to the world.

I wrote a Tetris game a couple of days ago. It took me three or four hours, using the C programming language from the 1970s, slightly augmented with type-checking from the 1980s. Using a more modern programming language would have simplified it only slightly. It doesn't handle key-repeat in the way I would like, which is substantially more difficult due to decisions baked into the X11 protocol in the 1980s to make terminal emulators easier to write; fixing that will probably take several more hours.

When checking one of the assertions in Engelbart's report, I quickly calculated how many words you'd need in a lexicon for it to cover half of the words in a sample corpus (95, for the British National Corpus). This took me 4 minutes.

Wikipedia, Stack Exchange, arXiv, the Internet Archive, and other sites with books and academic papers are perhaps the things that most closely approximate Engelbart's vision of jet-like "gains in personal intellectual effectiveness", since there I can easily get information about a wide range of topics. Unfortunately, though, they can only provide answers to questions that someone else has already learned the answers to; they are immensely useful for describing known consequences, but they are not useful for working out the consequences of situations I just dreamed up.

Consequently, I do most of my noodling in the same way I would have 56 years ago, with typed notes and a pencil and paper, but with quicker access to the library of knowledge represented by Wikipedia and the like, and the easier revision Engelbart suggests on p. 13 (though Emacs is considerably more fluid than the OCR stylus he suggests, more like the text editing process he suggests on p. 16, pp. 77–80, and p. 84). I have instant access to the worldwide community of knowledge workers, but this is useless; they largely waste their time on trivialities and infighting.

Nothing approaching Engelbart's vision of the augmented architect has been realized, despite the extensive computerization of architectural drafting (p. 5):

> He checks to make sure that sun glare from the windows will not blind a driver on the roadway, and the "clerk" computes the information that one window will

reflect strongly onto the roadway between 6 and 6:30 on midsummer mornings. … Finally he has the "clerk" combine all of these sequences of activity to indicate spots where traffic is heavy in the building, or where congestion might occur, and to determine what the severest drain on the utilities is likely to be.

I conclude that, so far, Engelbart's unfinished revolution is still unfinished.

The failure of diffusion

Remarks on pp. 16–17 express Engelbart's forlorn hope that a centrally-planned research effort could "greatly accelerate" the evolutionary process of diffusion of innovations:

Normally the necessary equipment would enter the market slowly; changes from the expected would be small, people would change their ways of doing things a little at a time, and only gradually would their accumulated changes create markets for more radical versions of the equipment. Such an evolutionary process has been typical of the way our repertoire hierarchies have grown and formed.

But an active research effort, aimed at exploring and evaluating possible integrated changes throughout the repertoire hierarchy, could greatly accelerate this evolutionary process. The research effort could guide the product development of new artifacts toward taking long-range meaningful steps; simultaneously, competitively- minded individuals who would respond to demonstrated methods for achieving greater personal effectiveness would create a market for the more radical equipment innovations. The guided evolutionary process could be expected to be considerably more rapid than the traditional one.

Perhaps if he had studied the accumulating literature on, for example, diffusion of agricultural innovations in poor countries, he might have had a better plan. In retrospect, it took 20 years for even the beginnings of his innovations to be adopted even by office workers, and 40 years to be broadly adopted by society.

How did Engelbart conceive training would happen?

Engelbart put a lot of emphasis on the importance of "training", which is an important part of the process of diffusion of innovations, but he seems to have lacked an anthropological or ethnographic conception of how this training would come about; or at least he is silent on the topic.

It's interesting that Engelbart takes as his starting point "a man", that is to say, an individual, rather than an institution, grappling with problems. (The unexamined sexism in the terminology was *de rigueur* in formal contexts in the US in 1962.) To my mind, much of the underlying zeitgeist of the work is collectivist, in the sense that Engelbart imagines institutions supporting the individual ("pursuit by an enlightened society"); he speaks of "diplomats, executives, social scientists, life scientists, physical scientists, attorneys, designers," perhaps with the implicit presumption that the users of his system will be supported not only with a powerful computer system but also with secretaries, administrative staff, a purchasing department, and so on. His "H-LAM/T" framework (p. 9) cites the necessity of "training", but never discusses the social context of the training — is it provided to new employees by a company, individually undertaken by problem-solvers who want to get augmented, or required of the whole population by a government?

In the 1980s, when some of Engelbart's ideas finally gained broad adoption, training was precisely the point on which he remained

outside the mainstream, and which made his continuing research progressively less relevant to that mainstream; CHI or HCI (already purged of the implicit sexism of the term "man-machine interface" term, or "man-artifact interface" as on p. 20) began to take advantage of findings from cognitive science with an eye to market competitiveness, while Engelbart did not.

On training, pp. 9–10:

> [W]hile an untrained aborigine cannot drive a car through traffic, because he cannot leap the gap between his cultural background and the kind of world that contains cars and traffic, it is possible to move step by step through an organized training program that will enable him to drive effectively and safely.

Of course, part of this is that the first part of the report is purely descriptive; it is not prescribing courses of action so much as it is describing the human world as Engelbart sees it. On p. 30 he talks a bit more about the circumstances of training in the context of possible research programs:

> For instance, some research situations might have to disallow changes which require extensive retraining, or which require undignified behavior by the human. Other situations might admit changes requiring years of training, very expensive equipment, or the use of special drugs.

(In this context it is worth pointing out that the Tuskegee Experiment was still ongoing at the time, no such thing as an IRB existed, LSD was still legal, and the MKUltra research program was in full swing, although it wouldn't be revealed to the public until 1975.)

This is unfortunate in part because the advent of computer games showed that computers were capable of producing fairly extreme learning performance by setting up a sort of Skinner box. But Engelbart was writing too early, and on the wrong coast, to have observed Spacewar!, and in any case computer games didn't really have a significant number of players until the 1970s. Consequently his focus, in practical terms, was mostly on designing new artifacts (in his H-LAM/T breakdown) to perform work for humans, then training humans to use them, rather than on designing new forms of training.

The lack of emphasis on communication

As I said above, Engelbart's focus is primarily individualist, focusing on the individual in their efforts to solve problems, but implicitly collectivist; he treats communication and community as secondary to individual thought and action, e.g., p. 22:

> Humans made another great step forward when they learned to represent particular concepts in their minds with specific symbols. Here we temporarily disregard communicative speech and writing, and consider only the direct value to the *individual* of being able to do his [sic] heavy thinking by mentally manipulating symbols instead of the more unwieldly [sic] concepts which they represent.

In a sense, this is the same lacuna represented by the mysterious absence of discussion of the social context of training.

He says, "temporarily," but so far I have not encountered where he analyzes communication in such a way.

On pp. 90–91, he does mention the utility of rich hypertext structures with typed links for communication, and even talks a bit about pedagogy:

> Well, when you ever get handy at roaming over the type of symbol structure which we have been showing here, and you turn for this purpose to another

person's work that is structured in this way, you will find a terrific difference there in the ease of gaining comprehension as to what he has done and why he has done it, and of isolating what you want to use and making sure of the conditions under which you can use it. This is true even if you find his structure left in the condition in which he has been working on it--that is, with no special provisions for helping an outsider find his way around. But we have learned quite a few simple tricks… Some of these techniques are quite closely related to those used in automated-instruction programming--perhaps you know about 'teaching machines?'

The power of the unpredictable and of irrationality

Engelbart touts of unpredictable processes, claiming that the power to use them is one of the largest benefits of the computerized augmentation he proposes (p. 45):

When the course of action must respond to new comprehension, new insights and new intuitive flashes of possible explanations or solutions, it will not be an orderly process. Existing means of composing and working with symbol structures penalize disorderly processes very heavily, and it is part of the real promise in the automated H-LAM/T systems of tomorrow that the human can have the freedom and power of disorderly processes.

Simultaneously, though, he perceives the subconscious mind as mostly an enemy due to its irrationality:

Clinical psychology seems to provide clear evidence that a large proportion of a human's everyday activity is significantly mediated or basically prompted by unconscious mental processes that, although "natural" in a functional sense, are not rational. … It may be that the first stages of research on augmenting the human intellect will have to proceed without being able to do anything about **this problem** [emphasis mine] except accommodate it as well as possible.

On reasoning and argument structures

On p. 62 Engelbart is discussing the process of revising a research design:

…when several consideration statements bore upon a given product statement, and when that product statement came to be modified through some other consideration, it was not always easy to remember why it had been established as it had. Being able to fish out the other considerations linked to that statement would have helped considerably.

Modern automated software testing supplies this in a somewhat restricted form: if you change the code and some tests start failing, you can go back and read those tests to find out what other considerations led to the code behaving as it had; but of course not everything can be described in code yet. Proof assistants perform a similar task in a more rigorous fashion for mathematical proofs. Requirements-management systems like DOORS also attempt to do this for more general structures of non-machine-readable statements.

He goes into more detail about his proposed antecedent-consequent links on pp. 84–88.

Chording keysets and premature commitment

On pp. 74–79, Engelbart describes the chording keysets that his system used into the 1980s:

He could hit a great many combinations of keys on his keyset--i.e., any one stroke of his hand could depress a number of keys, which gave him over a thousand

unique single-stroke signals to the computer with either hand.

He was quite optimistic about the speed of this system, with a quite reasonable information-theoretic basis:

It seems that, for instance, the 150 most commonly used words in a natural language made up about half of any normal text in that language. Joe said that it was quite feasible to learn and use the single-stroke abbreviations for about half of the words he used, but beyond that each added percent began to require him to have too many abbreviations under his command. …

(As it turns out, the 95 most commonly used words, without lemmatization or even stemming, make up half of the text in the British National Corpus; the 150 most common words make up 54.43% — though this disregards the tail of words that occurred less than 5 times, which might indeed amount to the 9% of the total that would make his statement correct. Thanks to augmentation, I could calculate this in about four minutes by writing an 8-line Python script. Doing the same exercise on words extracted from an earlier draft of this note, I needed 179 words, drawn from the 223 most common words in the BNC, to reach 50% of the words in this note; some common words, such as "she", "her", and "know", did not appear. 6.3% of the total words in this note, such as "1968", "lemmmatization", and "keysets", don't appear in the BNC at all. Incidentally, the point in the curve at which a given word makes up 0.1% of the corpus — so it could conceivably be efficient to enter it with a 10-bit abbreviation code — is just about that 50th percentile (95 words) in the BNC; this note is too short to give good statistics at that level. This greater rigor brought the total to almost 40 minutes rather than 4.)

A whole word so abbreviated saved typing all the letters as well as the spaces at either side of the word, and a word-ending abbreviated by a single stroke saved typing the letters and the end-of-word space. He claimed that he could comfortably rattle off about 180 words a minute--faster than he could reasonably talk. … He made some brief references to statistical predictions that the computer could make regarding what you were going to type next, and that if you got reasonably skillful you could "steer through the extrapolated prediction field" as you entered your information…

In practice, though, the chording keysets constructed by his team didn't really reach above about 50 words per minute, according to anecdotes. Part of this may have been that they ended up using only one key per finger (the "over a thousand" quoted above suggests two bits per finger, e.g., four keys per finger or two keys per finger that can be concurrently depressed) but other parts may have been the lack of the extensive abbreviation dictionary described.

This leads one to ask: why didn't Engelbart give up on his own keyset design at some point and use a Stenotype keyboard? Stenotype operators regularly reach well over the 180 words per minute he was hoping for. Instead, he apparently continued to hope that the design he came up with in 1962, before doing any experiments, would eventually pan out. That was a terrible idea; it prevented him from taking advantage of the 20+ years of experience he gained in the meantime.

Similarly, the outliner structure of NLS and Augment already appears on p. 84.

Topics

Bike charger

Kragen Javier Sitaker, 2014-04-24 (2 minutes)

What would it take to recharge your laptop and other electronic devices by riding your bike? There are highly efficient and reliable dynohubs these days. You could mount a small lead-acid gel cell battery on your bike and charge it by riding.

A regular car or motorcyle battery holds in the tens of kJ up to MJ. The Wikipedia article for "Car battery" says 30–40 Wh/kg, around 100 kJ/kg. My netbook battery says:

```
$ acpi -i
Battery 0: design capacity 5246 mAh, last full capacity 4518 mAh = 86%
```

It's, I think, 11 volts at the moment, which means it had about 200 kJ when full. So you'd need about 2kg of lead-acid battery to hold a full recharge for it, plus some small, lightweight electronics like a buck-boost converter to supply the laptop with the 19 volts it wants for recharging.

Amazon has a £17 lead-acid battery with 10Ah at 12V, or 400kJ. At the moment £17 is US$27. (Typical prices seem a bit higher.) It weighs 3.3kg. If everything scaled linearly, a 200kJ lead-acid battery would weigh 1.7kg and cost US$14.

At a substantial cost in reliability, safety, and money, you could use the laptop battery directly. I think it weighs more like 500 g, but it costs more like US$100.

A bike dynamo hub like the Schmidt SON28 consumes some 2–6 W of mechanical power, and probably has efficiency in the 80%–99% range. Getting 200 kJ out of 4 W would require almost 14 hours of riding, which I think of as two to four days' worth.

A modular display strategy would be very useful: several cellphone-style displays would both be easier to protect against breakage and allow scaling power usage up and down as needed.

Topics

- Independence (p. 3520) (63 notes)
- Energy (p. 3438) (63 notes)
- Batteries (p. 3340) (7 notes)
- Bicycles

Karplus–Strong PLLs

Kragen Javier Sitaker, 2017-06-09 (1 minute)

A PLL is great for tracking the frequency of a sine wave. But lots of sounds we want to track the pitch of aren't sine waves; they have substantial energy in their harmonics. The usual chopper PLL approach detects phase by multiplying the input signal with a square wave, thus convolving its spectrum with a spectrum containing significant third and fifth harmonics, but no even harmonics, then low-pass filters the result to try to separate out the dc component.

As a possible alternative for software PLLs, maybe you could try using the Karplus-Strong string model for your phase detection, using two delay lines that differ in length by one sample.

Topics

Forth with named stacks

Kragen Javier Sitaker, 2014-02-24 (7 minutes)

Something occurred to me on the way back. I was thinking about the lack of good notation for state machines, as opposed to pure functions; we don't have anything equivalent to Haskell for state machines, unless you count Haskell itself. But when you're using the state monad in Haskell to write state machines, lots of stuff goes away: you can no longer reasonably write it in point-free style, for example. I think.

The closest we have is Forth, which lets you write state machines in point-free style, but only up to a certain point: once you have more than two or three live values, you start to get confused about stack depth and stuff.

It reminded me of a thing I saw someone (Jeff Fox?) say a few years back about how stacks and registers are sort of duals at the level of CPU design: a stack lets you retrieve each value only once, but store many values, while a register lets you retrieve each value many times, but store only one value at a time. And it occurred to me that a way out of this is to make variables in Forth work a different way: rather than simply pushing a memory address onto the stack, instead each variable could name a separate stack out of a potentially infinite variety of stacks, and uttering its name could switch future evaluation onto that stack (until future notice). You'd need @ and ! to store and fetch values from some special magic place that wasn't on a stack: a register. But you could still say N @ M ! to move a value from N to M.

A couple of things that occurred to me about this notion:

• /=, +=, *=, etc., are the fundamental operations in a sense. N 3 + is sort of the equivalent of n += 3; from C.
• You get named local variables for free (naïvely, with shallow binding) as long as you're careful to store into them before you fetch.
• The return stack stops being special; it's just another variable.
• PICK replaces array indexing, as long as you don't mind indexing from the end of your array. That is, all of your variables are, in a way, aggregates (APL-style), although this competes with them being automatically local.

It occurred to me that this could maybe give you a notation for writing state machines with variables that was similar to RPN for pure math or regexps for state machines without variables.

[Some text in this note is from me typing messages to someone who could see my screen, and who I could hear, but who could not hear me. That's why it sounds like half of a conversation; it is. I may come back and edit this out at some point.]

Maybe interesting?

How about the "largest prime factor"? That's sort of cheating, maybe, since I was thinking about it...

So let's see. You have a sequence of letters, which we could push onto a stack as the argument; call it INPUT; then we want to switch between consuming whitespace and non-whitespace, incrementing a counter each time?

I guess arguments should be like a default named stack? Call it <>. What's not needed?

What does that look like if you want to, say, increment n?

OK, so some of the things on the argument stack should be arguments to the function and others should be other stacks?

You mean, put the state variable on the same stack as the input?

Do you want to type that out? Because I don't understand still.

So you use @ and ! to pop and push from the other stacks? Then you have something almost exactly like Forth with shallow-bound local variables and the need to dup values on the stack instead of in variables, which could be interesting; let's see if that's fruitful in a bit.

Hmm, { } from StoneKnifeForth is probably the wrong way to go, because most loops are better as while loops rather than do-while loops. So I think Knuth's/Forth's BEGIN/WHILE/REPEAT is better; but I want to spell it { | }.

```
: wc-w
  n 0                           \ initialize counter to 0
  { <> empty? ~ |
    <> { dup isblank? | pop }      \ drop characters until we find a blank
    n 1+                          \ here we switch to the n stack
    <> { dup isblank? ~ | pop }    \ again, but until we find a nonblank
  }
  n @ <> ! ;                      \ return n

: isblank? dup bl = [ pop #t exit ]     \ [ ] are IF THEN, with no ELSE.
           dup \n = [ pop #t exit ]
           dup \t = [ pop #t exit ]
           pop #f ;
```

<> switches to the <> stack. dup dups on the current stack, whatever it is, and immediately after executing <> it is the argument stack.

It does seem kind of promising, no? Somewhere I guess we have to declare n. Oh, and it's buggy because it always underflows the stack in the inner loops.

Reformatted horizontally:

```
: wc-w  n 0  { <> empty? ~ | <> { dup isblank? | pop }  n 1+
                            <> { dup isblank? ~ | pop } }
  n @ <> ! ;
```

Suppose my text is in memory instead of on the stack? It seems like we need a lot of dups in this notation, so I'm going to use "," for dup. This is looking suddenly a lot harder... how do I transliterate `while (n && isblank(*t)) t++, n--;`?

`n , [ ... ]`, so far so good; if we press into service | for "else" as well as "while", then we can end up with `n , [ ... | 0 ]`. But then what are you doing inside? You want to

That seems very promising! It could even be a deterministic regexp, perhaps, so that you don't have to handle backtracking.

Backtracking is tricky in the presence of add1.

Python itertools and family (not to mention Unix shell) show that you can do quite a lot by plugging together state machines that

generate and consume sequences.

```
: wc-w  @ t !  <> @ n !  words 0
  { n , | { n , [ t , c@ isblank? | t ++  n -- }
```

Should exiting from a loop or conditional restore you to whatever stack you were working on when you entered it?

What does this look like in Real Forth? Is it simpler?

Topics

- Programming (p. 3658) (286 notes)
- Programming languages (p. 3656) (47 notes)
- Syntax (p. 3738) (28 notes)
- Stacks (p. 3730) (21 notes)
- Forth (p. 3461) (19 notes)
- Stoneknifeforth

Programming paradigms for tiny microcontrollers

Kragen Javier Sitaker, 2007 to 2009 (6 minutes)

Concurrent actors in a microcontroller

Suppose you have a microcontroller with a small RAM (say, 2kiB) and you want to run bigger actor-oriented programs. Perhaps you can "page" the actors in and out to an external serial memory?

What I have in mind is that you can have many messages in flight at the same time --- enqueued in the RAM --- and some subset of actors resident. Whenever an actor sends a message to another actor, the message is enqueued, and the execution engine merely repeatedly searches for a message to deliver to a resident actor. When there are no more such messages, it "pages out" some actors to the serial memory to make space to "page in" some other actor that has messages waiting for it.

It's possible for the set of in-flight messages to get too big for RAM. There are a couple of tactics we could use in this case.

First, we could simply page out a chunk of the message queue to the serial memory. This will universally work, but it might make it hard to figure out what actors would be good choices to page in later.

Second, we could look for message deliveries that will diminish the number of in-flight messages instead of increasing it. In the classical Actors model and in Erlang, each handling of a message returns a new state for the actor, rather than mutating the actor's state during the handling of the message; if the system implemented this, then any message delivery could simply be undone by returning the message to the queue, deleting whatever outgoing messages the execution produced, and keeping the old actor state instead of using the new one. So the system could simply try each pending message, one after another, until it finds one that reduces the number of in-flight messages.

Third, we could page out actors to make more room for the message queue.

Of these three, the first strategy is apparently mandatory; the other two might be useful optimizations.

2kiB of RAM is enough to hold 1024 16-bit quantities; if a typical message contains four of these (destination actor, message name, and two arguments) then we could handle a queue of 256 messages at once. If half the RAM is dedicated to actor storage, we could handle 128. This is the level of concurrency at which the system would be most efficient; having fewer concurrently in-flight messages would worsen the choices of which actors to page in next, and possibly reduce the amount of work a particular actor could do before getting paged back out, while having more would not improve that choice, but would require time to be spent paging messages in and out.

When considering the size of each actor on its way to or from memory, it's important to remember that we have to include its code as well as its data. The data might be only 2-10 words, but the code will probably be much larger. So it's probably worthwhile to take this

into account in decisions of which actors to page in and out, and to share the code between objects when possible, just as in a traditional in-memory object system without concurrency.

There are several drawbacks to this scheme:

- the space usage of the message queue is inherently nondeterministic --- it depends deeply on the task switcher's choice of the order to run tasks in.
- the cost to "page" actors in and out may be excessive, especially since most microcontrollers don't have any support for DMA for access to external serial memories. The actors will have to be contain very much less code than the objects we are familiar with from current OO systems.

Concurrent tree-space transformation in a microcontroller

Suppose instead that we run Aardappel in the microcontroller. In place of in-flight messages, we have the trees of the tree space, which of course we page out to memory; in place of stateful actors, we have stateless rewrite rules, which rewrite one or more trees into zero or more trees. (Approximately.) In the Aardappel implementation, all the rewrite rules for a particular "type" of tree get collected by the compiler and compiled into a single function, where the "type" is the atom at the beginning of the expression for the tree.

So for a relatively simple system, we repeat the following process:

- figure out which type of tree is most abundant in the tree space;
- "page" in all the code needed for rewriting that type of tree (unless it's already paged in);
- rewrite all the trees of that type, paging them in as necessary, except for those that aren't currently rewritable (because some other tree is needed).

(This needs some refinement in case all the trees of the most common type aren't currently rewritable.)

I'm not sure this is really very different from the other proposal, but I think it is likely to work better for the following reasons:

- the code to rewrite a type of tree in Aardappel is likely to be substantially smaller than the code for a class in a Smalltalk-like language;
- all the state is in a single kind of thing, rather than being spread between messages and actor states;
- it seems straightforward in the source language to separate out rewrite paths that increase the number of trees from those that leave them constant or decrease them.

Topics

- Programming (p. 3658) (286 notes)
- Microcontrollers (p. 3580) (29 notes)
- Actors (p. 3305) (2 notes)

Building a resilient network out of litter

Kragen Javier Sitaker, 2014-04-24 (4 minutes)

Buenos Aires is about 300 km from Rosario. There's a train that runs twice a day, taking seven hours. The Flutter embedded wireless device, due to ship in 2014 for US$20 with a few kilobits per second and about a 1km radius of communication, could conceivably be thrown out the window of the train at regular intervals to provide a low-bandwidth, low-power communications relay line between the two cities; you would need perhaps 500 of the devices, for a total cost of US$10 000. This sounds like a lot of money but actually I think it's orders of magnitude cheaper than running a conventional communication line over such a distance; and it ought to decrease by another factor of two or three in the near future.

The remaining problems, then, would be to power the devices and to protect them from sabotage.

First, power. Nonrechargeable batteries will only last a year or two, maybe less in the sun. Small nickel-cadmium or nickel-metal-hydride cells might last five or ten years; but the problem remains to harvest sufficient power from the environment to supply the communication line. The device might use 2W in operation, so 2mW of photovoltaic power per device, with adequately efficient power harvesting, might be sufficient.

Second, protection. It's probably not practical to harden 500 such devices to make them impractical to break, but you could probably make them hard to find by concealing them inside of pieces of garbage. This also suggests a way to deploy them from the train without arousing suspicion: every 50 seconds, on average, one member of the planting team throws a piece of garbage (say, a food package) containing a wireless transceiver out the window. The train ticket costs US$2.50, so you could buy 4000 trips (2000 round trips) for the cost of the needed devices; so one person could do this in a practical fashion over some 50 to 100 trips, so two or three months. Ten people could do it in five or ten trips each.

You still need to prevent the devices from being detected by their radio emissions and destroyed, which is difficult if they're basically stationary. The best defense against this, other than spread-spectrum and using legal frequencies (the Flutter devices use a 915MHz frequency that's legal for unlicensed use in the US only) is very low duty cycle and very long periods of time between transmissions — days to weeks. This suggests that all the devices should wake up their receivers periodically at a synchronized time to see if they're being activated, say a few times a day, but not transmit anything unless a circuit is opened.

(Alternatively you could use store-and-forward transmission, but this would result in message latency of months.)

The desire to support solar power, combined with the desire to look like garbage, poses the problem of how to have solar cells that are not exposed to sunlight. This seems impossible, but keep in mind that we only need about 2mW. At the 5% efficiency typical of cheap

thin-film cells used for solar calculators and the like, this is 1 cm^2; it could be 4 or 5 cm^2 that's not directly exposed to sun, for example behind a layer of paper.

A perhaps more useful project would connect an off-the-grid location rather than a major city, and could be deployed by bicycle rather than train. There are locations in the Tigre delta, for example, that have electrical power but no broadband internet access.

Topics

Hacking a buck converter into a class-D amplifier?

Kragen Javier Sitaker, 2018-06-30 (4 minutes)

I was thinking about this 5V-output buck converter I dumpster-dived. A plausible-looking datasheet says the input is 8–48 volts and the output is 5V at up to 2.1 amps. It has a feedback pin which it uses to adjust the duty cycle of the 150kHz PWM signal it uses to generate the output; you're supposed to hook that up to the output side of the inductor so it can regulate the output voltage properly.

It occurred to me that this sort of implies that if, instead of hooking up the feedback pin directly to the output side, you hook it up to the output side through a voltage divider to ground, you should be able to get it to act as a 6V or 8V or 10V regulated power supply instead of a 5V one. Maybe it won't compensate for load changes quite as quickly, but I don't think it should oscillate.

But what if, instead of using a voltage divider to ground, you use a voltage divider to a *variable* reference voltage? Maybe you could make a high-power class D audio amplifier! Say you use a high-impedance 4:1 divider, so that a 1-V shift in the "reference" voltage (really the input) requires a 4-V shift in the output in the opposite direction to compensate. It'll be superimposed on a 20VDC level that you probably want to block with a capacitor, but 4 volts RMS across 8Ω should give you 500 mA RMS, which is 2 watts, which is pretty loud. Filtering out the 150kHz (and harmonics) carrier should be pretty easy.

Also, you might be able to use this as an AM radio transmitter. AM radio is in the range 540 kHz to 1610 kHz, so there ought to be some harmonics of the 150kHz PWM signal in there (at least the fourth through ninth or so), and their power ought to vary with its duty cycle. Some experiments reveal that the second harmonic of a PWM wave is zero at 0% power, 50% power, or 100% power (which is the same as 0%), but maximum at 25% power; the third harmonic is zero at 0% and I guess 33⅓% and 66⅔%; the fourth harmonic is zero at the same places as the second harmonic and also at 25% and 50% power; the fifth harmonic at multiples of 20%; and so on. So lower harmonics should modulate more reliably. If you're trying to modulate the fourth harmonic, you'd probably want to use a duty cycle on the steepest part of this slope, like in between the maximum at 12½% and the minimum at 25%, around 18.75%, which means you want to regulate the nominal output voltage to be around 18.75% of the input voltage, and vary that percentage according to the signal you're trying to modulate. Then you "just" need to filter out the other harmonics (a bit tricky since the actual frequency could be anywhere from 125kHz to 175kHz, so the desired fourth harmonic could in theory be anywhere from 500kHz to 700kHz, while the undesired fifth harmonic could be anywhere from 700kHz to 875kHz), hook it up to an antenna, and tune a radio to it.

The thing isn't designed for an output duty cycle of over 62½% (⅝), but maybe using the region between the maximum at 37½% and

the minimum at 50%, like around 43.75%. For its nominal 5V output, this would mean an input voltage of around 11.43 volts, which is more manageable than the input voltage needed to get 18.75%.

Topics

Clay fabrication objectives

Kragen Javier Sitaker, 2017-01-16 (updated 2017-01-17) (3 minutes)

I'm doing some stuff in the ceramics lab, with the objective being to construct a self-replicating micromachine. I have four key objectives to achieve this:

• Full recursion: do the entire fabrication process, from separation to firing and assembly, using entirely components that can be fabricated with that process. Alternatively, if this objective is misguided (perhaps using metals, glasses, plaster, or plastics would be more reasonable), I want to discover that.
• Material properties characterization, simulation, and optimization: I want to have quantitative measurements of the main physical properties of the material in both green and fired states. In particular:

• Tensile strength and modulus.
• Shear strength and modulus.
• Density.
• Thermal expansion.
• Separately, fracture resistance, even though that can in theory be predicted from the other properties.
• Abrasion resistance.
• Dependences of these on relevant process parameters.
• Heat resistance (e.g. maximum service temperature, softening point if any, sintering point, melting point, perhaps variation of other properties with temperature below the softening point).
• Creep, plastic deformation, and related complications; although I expect these will be very low for fired clay ware, plastic deformation is an enormously important process for shaping wet clay, which can have almost zero elastic deformation and creep but enormous plastic elongation.

Given an adequate numerical characterization of the materials' properties, simulation of its behavior under different circumstances should become possible; by running a series of such simulations with different designs and performing error and differentiation analyses on the simulations, a substantial degree of automation in design should become possible. If the simulation performs adequately, it should be possible to run such optimization processes during the fabrication process in order to automatically improvise responses to newly available information.

With the first two of these items, I can do FEA analyses and optimizations for static loading; with the third, I can do dynamic loading as well. The others are useful for particular cases.
• Generation-time reduction by process intensification: I predict that the primary figure of quality for self-replicating machinery will be its mass growth rate, which needs to exceed the IRR of available investments in order to be economic, and which is an exponential function of the generation time.
• Miniaturization: somewhat secondary to the above three considerations, smaller is better. This is largely a means to an end: thinner walls will dry faster and fire faster (up to a point), linearly

smaller machinery will have a proportionally shorter generation time, and the demands its mass places on its mechanical properties will be proportionally smaller, both permitting more geometrical freedom and simplifying calculations. Miniaturization also helps with safety and cost of materials and energy. However, miniaturization has limits: it requires more precise manipulation and measurement, surface effects like stiction and sliding wear become more serious concerns, and in particular thermal processes like firing pottery become more difficult to complete at smaller scales.

Topics

The Dontmove archival virtual machine

Kragen Javier Sitaker, 2014-06-29 (5 minutes)

(In this version of the dontmove archival virtual machine, Aa input and output a byte, Bb read the subtractor or subtract a number from it, Cc read and write the word in memory in the Dd register, and Ee read and write the program counter, the last of which leaves a return address in the accumulator. Somewhere I had a sort of "signum" register but I forget where it is.)

I was thinking about how to do an assembler for dontmove, and I think that the right thing to do is probably really a simple Forth compiler, one that pastes together chunks of code with known stack effects. If we put the top of stack in Y, the down-counting stack pointer in X, the down-counting return stack pointer in W, and the return address (top of return stack) in V, assume Z is zero, and use U as a scratch register, then DUP is "BbXbBbbZ1bBdxYc", for example, and DROP is "BbZ1bBbbBuXdbBbbUbBxCy" or "BbXdbZ1bBbbBxCy". Function calls are similarly obnoxious; the actual function call itself is simply something like "Z1234e", where 1234 is the address of the function; and function entry is simply "v", and function exit is simply "Ve"; but before the call it is necessary to save V onto W, which is "WdVc", and decrement W, which is the painful "ZbWbBbbZ1bBw"; and after the return, restoring the previous V from W is also necessary. (Perhaps leaf functions could avoid that, and in other functions it could be moved into the prologue and epilogue.)

(Note, there's lots of confusion in the code both above and below about whether the pointers point to the last item pushed or the space for the next one.)

If we had a way to simply set the contents of the subtractor, say "f", rather than subtracting from it, then these would simplify substantially, replacing lots of cases of "Bb...bBbb" with "...f". Also let's suppose we have a premade -1 in, say, T, which we can set up at some point with "Zf1bBt".

- DUP: "BbXbBbbZ1bBdxYc" -> "XfZ1bBdxYc"
- DROP: "BbZ1bBbbBuXdbBbbUbBxCy" -> "XfdTbBxCy"
- save V onto W and decrement W: "WdVcZbWbBbbZ1bBw" -> "WdfZ1bBwVc"
- restoring V from incremented W: "WfTbBwdCv"

(On second thought, it might make more sense not to keep top-of-return-stack in a register; then non-leaf function entry is something like "vWdfVcZ1bBw" and non-leaf function exit is "WfTbBwdCe", needing no scratch register. Without "f", non-leaf entry is, for example, "vBbWdbBbbVcZ1bBw". Leaf entry and exit can still be "v" and "Ve".)

At some point you might get to questioning whether it's worth maybe having increment and decrement instructions, distinct modes for reading memory (add to accumulator, subtract from accumulator, XOR with accumulator) but it doesn't take much to get to an abstract

machine that's more complicated than a GreenArrays core.

Forth "@" (memory fetch) is then "YdCy". Store "!" is complicated by needing to pop the stack twice: "XfdTbBxCsYdScXfdCyTbBx", or something like that. SWAP is easier due to the lack of arithmetic: "YuXdCyUd". OVER, which pushes onto the stack, is something like "XfdCuZ1bBxdYcUy".

If we had an increment instruction "+" that would increment the accumulator instead of having "f", then store would become "XdCsYdScX+d+xCy". DUP, DROP, and function entry and exit would similarly benefit from increment and decrement: "X-dxYc", "Xd+xCy", "vW-dwVc", and "Wd+Ce" respectively. This + and - completely eliminate the need for "f" for these basic operations.

Forth "+" then would be "BbYbX-dxCbBbbBy" and Forth "-" is "BbX-dxCbBbbYbBy".

Bitwise AND is going to be a real bear, involving a loop that shifts bits to the left by adding a number to itself (something like "BbYbbBbb") and then somehow getting the most significant bit.

This implies that we can probably do orders of magnitude better by adding a NAND operation. Say it was Gg, behaving like Bb: g NANDs the input with what's already there, and you can read the result with G. Now "Gg" NANDs the current result with itself, bitwise inverting it, and "Zg" NANDs the current result with 0, setting it to all-1, which also means that it will bit-invert the next thing we "g" into it. Then we can AND registers Y and U with "ZgYgGgUgGgG": first putting the bit-inversion of Y into G, then inverting that with Gg, then NANDing U with Ug, inverting the result with anothe Gg, and then getting it with G. Similarly OR of U and Y becomes "ZgUgGuZgYgUgG", I think.

The more I think about this, the more I wonder if something like Calculus Vaporis is a better design.

Topics

- Programming (p. 3658) (286 notes)
- Instruction sets (p. 3526) (40 notes)
- Archival (p. 3322) (34 notes)
- Assembly language (p. 3328) (25 notes)
- Dontmove (p. 3415) (2 notes)

Designing a drawing editor for well-factored drawings

Kragen Javier Sitaker, 2019-05-07 (9 minutes)

As described in Dercuano drawings (p. 64), I want to add illustrations to Dercuano. These include things like sketches of screen layouts, box-and-arrow diagrams for things like data flows, cutaways of mechanical devices, pictures of three-dimensional solids to refer to in the text, plots of physical properties, decorative orders, timelines, diagrams of records in memory, and so on. But I want the illustrations not to be extremely large files, so as to keep Dercuano easily downloadable.

In Some musings on applying Fitts's Law to user interface design and data compression (p. 1164), I've talked a bit (perhaps to the point of beating a dead horse) about achieving this kind of efficiency at the primitive level: freehand drawings are going to involve a lot of coordinates of points, lines, and curves, placed individually with a stylus, mouse, or finger, and continuously displaced until they look right. And it's probably beneficial both to the quality of the drawing and to the download size to include intentional, considered placements and exclude random measurement noise.

But the primitive level is just one aspect of a visual language, albeit the only one that is accessible to purely freehand drawings, which have much to recommend them. And on top of this, we can layer things like blur and sharpen filters, or cutting and bending existing primitives, or using them as reference points for more primitives. However, it's also often useful to make drawings by combining existing drawings, rather than starting entirely from scratch. The simplest way of doing this is just putting one drawing next to another in a larger drawing, and a better way of doing merely this was sufficient for Gutenberg to revolutionize the world. But several other means of combination are possible!

In particular, let's talk about curves ("polylines") and how they can be combined.

Curves have many attributes

A curve on the page has a shape, by which we mean that each point along the curve has a position, an orientation, and a curvature. If we remember which direction the curve was drawn, it has a direction, which is the orientation plus one more bit of data. If we remember the speed with which the curve was drawn, each point also has a time and a speed. These are the intrinsic data of the curve; it also has a variety of presentation attributes or aesthetics, typically including fill color, stroke color, z-order, and stroke thickness, which we can change without changing (in the view we adopt here) the curve itself.

Most of these aesthetics can also vary continuously along the length of the curve, just like its curvature and position. Letterform curve thicknesses typically vary according to the local orientation of the stroke, and we could imagine applying such a relationship to a curve or many curves, perhaps interactively adjusting anchors on an auxiliary X–Y function curve like the luminance curves in the

GIMP. Or we could adjust the curve thickness as a function of time, or of the distance from the start of the curve, or, if we make it a function of speed, orientation, and direction, we can perhaps achieve a quill-pen calligraphic effect.

A curve can be approximated by a series of directed line segments, each of which is associated with a local frame of reference, one which distorts (perhaps via perspective) to follow the curve. These distorted frames of reference, if used to transform the unit square, will produce a sequence of trapezoids that approximate the curve with unit thickness. But the frames of reference can be orthogonalized, normalized, and even derotated, so that they become nothing more than a sequence of translations to the midpoints of line segments along the curve; or only some subset of these neutering operations can be applied. The line segments themselves can be placed according to fixed length, fixed time, fixed maximum angle, fixed approximation error, or some other criterion. By using fixed-length segments and using them to transform a tileable graphic segment, we can transform the curve into a rope, a braid, a dotted line, a (rather poor) dashed line, or a curly border.

Given two curves, we can produce a variable number of additional curves interpolating between them (both in intrinsics and in aesthetics), or extrapolating beyond them. Moreover, we can adjust the interpolation; the interpolated curves can be anchored along some other curve, at the midpoints of the approximating segments described earlier, and further auxiliary curves can adjust the nature of the interpolation.

Interpolating an array of lines along a line gives you a fence, a brick wall, or half of a grille. Interpolating an array of lines along a curve can give you a ribbon, a suspension bridge, or a sunburst. Etc.

From a curve with speed you can extract a "normalized" curve that has lost its speed information, making it equivalent to the distance along the curve.

Any of these nonscalar attributes, intrinsic or aesthetic, can be pseudorandomized by adding deterministic noise, which can have varying frequencies and amplitudes.

By making these attributes potentially time-dependent, we have animations.

Stroking a curve produces another curve, one we typically fill with black; but we can use it to clip a texture in order to get many other kinds of curves.

The ends of open curves are special; we can mark them with arrowheads.

Reducing abstraction overhead

A stencil that is merely copied wholesale into another drawing is already useful, and turning a drawing into such a stencil is conceptually trivial. Nothing about the drawing need be explicitly abstracted, and it is then subject to whatever manipulations are available in its destination environment, which may include things like rotation, scaling, clipping, colorspace transformation, convolution, alpha-blending, skewing, and cutting into pieces. (An image that has already been made so concrete that it consists only of pixels may not rotate as well, though, and one that has been merged with a transparent background may need some extra flying-matte

work to re-abstract it to use in a new situation.)

But often these leave something to be desired. Scaling can change stroke widths, for example, and sometimes the original stroke width is desired; in other cases, bringing in stroke widths and colors from the new environment is necessary to get it to "look right". And you may want to change the text, adjust a rectangle to meet up with another graphic element, and so on.

However, often the biggest issue is that it's just very fiddly to convert a concrete graphical object into something that can be reused. The overhead of "abstraction", even when no detail is actually being removed, eliminates the majority of opportunities in mainstream programs.

Enabling individual drawn strokes in a drawing to be used as "stencils" in context, providing immediate feedback as they are edited, and making the combining facilities more flexible may help with this problem.

Constraint drawing and optimization

Sutherland's SKETCHPAD, SolidWorks, SolveSpace, and FreeCAD are four well-known constraint-drawing systems; their drawings contain graphic objects whose concrete parts are held in relation to one another by non-unidirectional constraints. (See Relational modeling (p. 1102) for some more thoughts about this, generalized beyond purely graphical applications.)

Constraint-based drawing has a potentially very flexible interface for combining drawings, even if only points can be constrained to be equal (this can enable one object to fit with another).

SKETCHPAD solved its constraints iteratively by a sort of optimization approach. It's common for drawings to have some kind of desiderata they would like to optimize — it's best if arrows don't pass through boxes in box-and-arrow diagrams, for example, or kink too much or cross each other at shallow angles, but any of these can be acceptable.

Particle systems

A seed leaps along a trajectory, curving under gravity, depositing branch segments behind it until it flames out; periodically a leaf sprouts. Your mouse draws an arch, from which explode a thousand sparks, each of which seeks a consistent distance from its neighbors before sprouting a tapered, curved eyebrow hair. Randomly sprayed points sprout into randomly rotated copies of a mother sand-grain polygon, each with its vertices perturbed slightly differently, and they repel until all their surfaces are a minimal distance apart. One by one, slightly inconsistently sized bubbles appear at the nearest bubble boundary at a source, and bubbles shift and move until every bubble is in equilibrium.

How can these systems, so seemingly simple to describe, be simple to create, debug, and tweak interactively? Does the temporal description help or hinder?

Topics

Filling hollow FDM things with other materials

Kragen Javier Sitaker, 2016-09-07 (5 minutes)

You can improve some characteristics of 3-D printed FDM things by filling them with other stuff.

"Filling" with plastics usually means mixing the plastic with "filler", originally just some random solid material that was cheaper than the plastic (thus the name), but often these days something that imparts other superior properties to the plastic: higher strength and lower gas permeability in the case of bentonite clay; higher strength and density, plus reflectiveness in the case of steel or aluminum (aluminum-filled nylon is what Shapeways calls "alumide"); color; conductivity, when graphite, silver, or gold is the filler; thermal conductivity, using e.g. iron oxide; surface texture, as when sawdust is used as a 30% filler in PLA filament for 3-D printing to produce "printable wood"; and so on.

That's not what I'm talking about. I'm talking about using FDM to print a hollow shape, and then filling the hollow spaces with some other material, in order to provide different properties to the resulting object. In some cases, you might want to actually remove the FDM plastic entirely afterwards, sort of like lost-wax casting, but without the intermediate step of making the negative mold around your wax positive.

What kinds of properties might you want to get?

Mass

Mass is a big one. FDM is capable of producing impressively light and thin objects, but it's a very slow process for massive objects. And for many applications, "light and thin" equals "cheap-feeling and fragile". So simply filling a cavity with a cheap, heavy material can make a big difference.

Here are some materials you could reasonably use for adding mass:

	g/cc	max temp	
paraffin wax	.9	46°-68°	<https://en.wikipedia.org/wiki/Paraffin_wax>
water	1	0°	
sugar syrup	1-1.6	0°-186°	<https://en.wikipedia.org/wiki/Sugar>

| silicone caulk | 1.04 | >260˚ | <http://catalogue.airtech.lu/product.php?product_id=30&lang=EN> <https://en.wikipedia.org/wiki/Silicone_rubber> |

| ABS | 1.07 | 210˚ | <https://en.wikipedia.org/wiki/Acrylonitrile_butadiene_styrene> |

| epoxy resin | 1.1 | 78˚-162˚ | <http://msdssearch.dow.com/PublishedLiteratureDOWCOM/dh_004e/0901b8038004e7b6.pdf?filepath=epoxy/pdfs/noreg/296-00312.pdf&fromPage=GetDoc> |

| PLA | 1.3 | 60˚-140˚ | <https://en.wikipedia.org/wiki/Polylactic_acid> |

| salt | 2.2 | 801˚ | <https://en.wikipedia.org/wiki/Salt#Chemistry> |

| portland concrete | 2.2 | 573˚ (quartz) | |

| gypsum (plaster) | 2.3 | | <https://en.wikipedia.org/wiki/Gypsum> |

| quartz | 2.6 | 573˚ (phase transition) | <http://www.mindat.org/min-3337.html> |

| whitewash/chalk/calcite | 2.7 | 600˚ | <https://en.wikipedia.org/wiki/Calcium_carbonate> |

| aluminum | 2.7 | 660˚ | <https://en.wikipedia.org/wiki/Aluminum#Physical> <https://en.wikipedia.org/wiki/Solder> |

| magnetite | 5.2 | | <https://en.wikipedia.org/wiki/Magnetite> |

| steel | 8 | | <https://en.wikipedia.org/wiki/Steel#Material_properties> |

| 37% tin, 63% lead solder | 9.8 | 183˚ | <https://en.wikipedia.org/wiki/Solder> <http://www.metallurgy.nist.gov/solder/NIST_LeadfreeSolder_v4.pdf> |

| lead | 11.3 | 327˚ | <https://en.wikipedia.org/wiki/Solder> <http://www.metallurgy.nist.gov/solder/NIST_LeadfreeSolder_v4.pdf> |

| glass powder | 1.7 | 850° (Ts) | <http://www.expertsystemsolutions.com/biblio/HSM/sintering%20behavior%20of%20glassceramic%20frits.pdf> |

The cheapest filler material is water, but it also has the lowest density of any solid here. All the others listed here are currently pretty easy to buy in kilogram quantities, a criterion which excludes otherwise interesting materials like cyanoacrylate, gallium, Wood's metal, gelatin, and albumen.

Salt, gypsum, concrete, and whitewash have the property that you can convert a powder into a solid mass at room temperature by adding water and, in the case of whitewash, carbon dioxide. Paraffin is similar in this, in that you can divide it into grains and then sinter them, but it also melts at a low enough temperature that you may be able to pour it into an FDM-printed mold, possibly carrying other filler materials with it. (Sugar can also be extruded directly, as Jordan Miller has done: http://www.thingiverse.com/thing:26343/#instructions.)

Epoxy resins and silicone caulks, similarly, will harden at more or less room temperature, starting in a liquid state.

Flexibility

Silicone is flexible and high-temperature tolerant.

Removing FDM plastic

You should be able to melt PLA or ABS off of silicone once it's hardened; also of concrete, salt, plaster, chalk, or metals. You can also dissolve ABS with acetone or PLA with a mixture of alcohol (ideally propanol, although I've heard ethanol or even water works too) and sodium hydroxide. Weld-On #5 solvent also works to dissolve PLA, but it's nasty, nasty shit, made of methylene chloride, glacial acetic acid, and methyl methacrylate monomer. Miller-Stephenson MS-111 stripping agent can also dissolve PLA, and also epoxy (!!) https://groups.google.com/forum/#!topic/ultimaker/8s1bq_9LsRM. http://www.vinland.com/blog/?p=68. MS-111 is 50% methylene chloride, 20% phenol, 15% formic acid and will blister your skin.

Dichloroethane can supposedly also dissolve PLA.

Acetone, MEK, tetrahydrofuran, and TCE will plasticize PLA but not dissolve it. I think all three of them will dissolve ABS, and MEK and TCE may be more available here in Argentina than acetone.

Ethyl acetate can vapor-polish PLA, which suggests it can dissolve it: http://www.printedsolid.com/smoothpla/

My attempts to polish PLA with ethyl acetate, or dissolve it in a test tube, were unsuccessful.

Topics

- Materials (p. 3560) (112 notes)
- Manufacturing (p. 3558) (50 notes)
- Digital fabrication (p. 3411) (42 notes)

Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1

Kragen Javier Sitaker, 2019-07-25 (6 minutes)

While the amount of solar energy available is far greater than world marketed energy consumption, by about four orders of magnitude, solar energy mostly only arrives during the day, and sometimes even during the day most of it is blocked by clouds, while existing marketed energy is generally available 24 hours per day. The impending transition to a photovoltaic-solar-energy-based economy therefore has many of the humans worried about the "intermittency problem" or the "storage problem".

Intermittent usage, or demand response

Probably the most important result of the intermittency problem is that the humans will use energy intermittently.

In some cases this is relatively straightforward and already practiced — for example, it's common for large air-conditioning systems on commercial properties to run their chillers during the night, when electrical energy is cheap, to store up ice, and then use the melting ice during the day to chill a coolant which is circulated to chill air, and electric car owners often have a lot of latitude about when they charge up.

In other cases, it's a matter of moving demand geographically — perhaps a Google Search query needs to be done in a data center close to the requester, but a large finite-element simulation can be done in whichever data center the computation is cheapest. And in some cases long-distance power transmission can bridge a storm cloud.

In other cases, idling machinery at night is expensive because of the mere depreciation of the machinery, and it might be worth paying higher energy costs.

In still other cases, like life-support machinery and alarm systems, it's worth paying even extremely high energy costs to guarantee uninterrupted power.

Lithium battery storage

Right now, the popular approach to utility-scale energy storage to reduce solar intermittency is lithium-ion batteries, similar to the Tesla PowerWall or Jehu Garcia's DIY powerwall projects, but larger. (There are a dozen other possibilities — the Sisyphean train, pumped-water storage, flywheels, compressed-air caverns, vanadium flow batteries, hydrogen or methane fuel cells, molten-metal batteries, and so on — but lithium is the currently popular one.) You might reasonably wonder: is there enough lithium?

The Earth's crust contains about 10^{17} kg of lithium

(Updated from my comments on the orange website.)

In 2018, the USGS estimated 16 million tonnes of "worldwide identified reserves" of lithium (1.6×10^{10} kg), but about 53 million

tonnes of "resources" of lithium (5.3×10^{10} kg), which are extractable with current techniques but not necessarily economic at current prices. There's also 2.3×10^{14} kg of lithium dissolved in seawater, and the estimates of lithium abundance in the earth's crust have a low end of 20 ppm. The crust averages about 40 km thick and 3 g/cc in the 29% of the Earth that has continents, which works out to 1.5×10^{14} m², 5.9×10^{18} m³, 1.8×10^{22} kg of rock, and 3.6×10^{17} kg of lithium.

World lithium "production" (which is to say, extraction) is about 4.3×10^{7} kg per year, which would take 350 years to exhaust the "worldwide identified reserves". It seems like a safe bet that some new mining technologies will become available before even 2100, let alone 2369.

Lithium is not destroyed when it is used — you can recycle worn-out batteries into new batteries — so, like gold, we should expect that eventually the amount circulating is much greater than the amount mined in any given year or decade. But, how much would need to be circulating for the humans to start using lithium batteries to power entire continents overnight? Maybe extraction would have to speed up?

The lithium in the crust can store 130 zettajoules

Wikipedia's table of energy densities says Li-ion batteries contain 0.36–0.88 MJ/kg, or slightly higher if you only count the mass of the lithium rather than the entire battery. Conservatively taking 0.36 MJ per kg of lithium (which assumes battery technology doesn't improve — almost-nonrechargeable lithium-metal batteries get five times that much energy per kg of lithium), the 1.6×10^{10} kg of "identified reserves" would hold 5.8 petajoules; the 2.3×10^{14} kg of lithium in seawater would hold 83 exajoules; and the 3.6×10^{17} kg of lithium in the continental crust would hold 130 zettajoules.

Current world marketed energy consumption is on the order of 5.7×10^{20} joules per year, which is 18 terawatts. Incident solar power on the Earth (the Kardashev Type 1 benchmark) is 130 petawatts. So the 5.8 PJ of "identified reserves" is only five minutes of world marketed energy consumption, but the 83 EJ of seawater lithium is about 1.7 months of world marketed energy consumption. Even so, that's only 10 minutes of total terrestrial insolation. But the 130 zettajoules of continental crustal lithium is 12 days' worth of total terrestrial insolation.

So it seems likely that known concentrated lithium deposits will not be sufficient to permit the transition to solar over the next decade or two, but there is plenty of lithium in seawater and other, less-concentrated deposits to permit such a transition. New extraction technologies will be needed if lithium batteries are to bridge the intermittency gap. Alternatively, some of the other utility-scale storage technologies might be developed.

Summary table

Lithium	Energy storage	World marketed energy consumption	Terrestrial insolation
Identified reserves	1.6×10^{10} kg	5.8 PJ (1.6 million MWh)	5 minutes	45 milliseconds
"Resources"	5.3×10^{10} kg	19 PJ		

(5.3 million MWh) 18 minutes 150 milliseconds
Seawater 2.3×10^{14} kg 83 EJ
(23 billion MWh) 53 days 11 minutes
Crust 3.6×10^{17} kg 130 ZJ
(36 trillion MWh) 230 years 11 days

Topics

Leconscrip: a family of JS subsets for BubbleOS

Kragen Javier Sitaker, 2018-11-23 (2 minutes)

Leconscrip is a family of languages for implementing BubbleOS (see Speculative plans for BubbleOS (p. 2128)).

It's mostly a subset of JS, since that eliminates unnecessary syntactic obstacles. But it's implemented as a series of levels.

Leconscrip level 0: Lecon

Lecon is nearly the lowest programming level at which it makes sense to use JS syntax at all; it's barely above the assembly level. It has recursive functions with global and local `let` variables, assignment, integers, `if`, `while`, binary `+`, `-`, `*`, `%`, `&`, `|`, `^`, and arrays of integers. The arrays must be defined with the syntax `let x = Array(k)`, where `k` is a compile-time constant. Expressions are only permitted as the right operand of a variable initialization or assignment. It doesn't support closures, objects, object references, or `for` loops. Functions can return integers. Identifiers are limited to two characters. Semicolons are required. I/O is done with `read` and `write` functions on arrays of integers representing bytes.

Variables are of three types: arrays, integers, or functions. Type inference is used; only arrays can be indexed, only integers can be indexes or participate in arithmetic, and only functions can be called. Parameters can be arrays or functions.

Because Lecon doesn't directly permit runtime allocation, it is (in conjunction with a stack-depth checker) suitable for functions in which failure is not an option.

Here's the whole grammar, bottom-up, as a PEG.

```
_ <- (` ` / `\t` / `\n`)*.
LP <- `(` _.
RP <- `)` _.
LB <- `{` _.
RB <- `}` _.
LS <- `[` _.
RS <- `]` _.
EQ <- `=` _.
C <- `,` _.
S <- `;` _.

literal <- [0-9]+ _.
name <- [A-Za-z] ([A-Za-z0-9] / ) _.
atom <- name / literal.
op <- [-+*%&|^] _.

aparams <- atom (C aparams / ) / .
fparams <- name (C fparams / ) / .

call <- atom LP aparams RP.
expr <- atom op atom / atom LS atom RS / atom.
```

```
assign <- name EQ expr.
decl <- name EQ `Array` _ LP literal RP / assign.
decls <- decl (C decls / ).

return <- `return` _ atom.
let <- `let` _ decls.

block <- LB statement* RB.
if <- `if` _ LP atom RP statement.
while <- `while` _ LP atom RP statement.
statement <- block / (let / return / if / while / assign / call) S.

func <- `function` _ name LP fparams RP LB statement* RB.

prog <- func*.
```

Topics

Speculative plans for BubbleOS

Kragen Javier Sitaker, 2018-10-28 (updated 2019-02-24) (12 minutes)

BubbleOS is a tiny bubble of sanity in a universe of insane operating systems. It's self-bootstrappable — it includes all the tools needed to build it, edit its source code, and so on — and it can be used as a robust front end to a family of less sane systems. This provides a variety of benefits.

You can use BubbleOS as a standalone computing environment, run it as domo under Xen, or run it as a process under Linux (including Android), MacOS X, Microsoft Windows, or an HTML5 browser.

BubbleOS contains the following:

• A hard-real-time secure windowing system.
• A hard-real-time secure microkernel designed to protect the user from malicious applications.
• A couple of hard-real-time secure terminal emulators.
• A hard-real-time layout system built on a simplified version of the CSS box model.
• A text editor.
• A minimal TCP/IP implementation.
• Minimal cryptographic facilities, an ssh client and server, and a Bitcoin client.
• Encrypted swap and disk.
• Unicode fonts.
• Unicode text rendering including bidi and combining characters.
• A bootstrapping chain of compilers, including assembler, low-level, high-level, and very-high-level languages. The assembler is a variant of Forth, the low-level language is a subset of C, while the others are extended subsets of JS.
• A compiler for a subset of C++ sufficient to cross-compile GCC.
• A SAT solver which provides some of the horsepower for these compilers.
• A virtual CPU machine set, with simulators for i386, amd64, RISC-V, and ARM(32), which runs all the other code.
• A hardware design for this CPU.
• A library of engineering models of many different physical phenomena.
• A circuit design and synthesis system capable of synthesizing this CPU for certain Lattice FPGAs.
• An analog circuit simulation program similar to SPICE.
• A general mathematical optimization system that provides most of its horsepower.
• Emulators capable of running CP/M, MS-DOS, or Apple][programs.
• An image-format library capable of reading and writing most common image file formats, including JPEG, PNG, TIFF, and GIF.
• A minimal web browser capable of fully rendering Wikipedia and Stack Exchange, though it lacks support for most modern web standards.

• Snapshots of Wikipedia and Stack Exchange for offline viewing.
• A fast, capable, secure web server.
• The ability to checkpoint and migrate either a full machine image or an individual process.
• Its full source code as a hypertext literate program.
• A fast, secure, crashproof content-addressable filesystem that supports transactions.
• A fast key-value store similar to LevelDB.
• A minimal SQL database.
• A terser database query language than SQL, but equally powerful, better suited to interactive exploration.
• A fairly full set of data-wrangling facilities similar to Unix software tools, but implemented differently.
• An implementation of Git.

It does not provide a POSIX API.

This is 32 major components in all. I'd better get cracking!

Naming

Bikeshed-style, the thing I am currently devoting my attention to is what to name the components. Names from Consider Phlebas:

Perosteck Balveda (Juboal-Rabaroansa Perosteck Alseyn Balveda dam T'seif), Gravant (alias for Perosteck Balveda), Amahain-Frolk, Egratin, Bora Horza Gobuchul (Horza, shapeshifting traitor), Xoralundra, Farn-Idir (sect), Schar, Heibohre, Rairch (species), Kraiklyn, Zallin, Wubslin, Yalson, Gow, kee-Alsorofus, Marain (language), Tzbalik Odraye, Mipp, Rava Gamdol, Aviger, Lenipobra, Lamm (traitor), Cifetressi, Dorolow, Jandraligeli, Chicel-Horhava, Neisin, Sro Kierachell Zorant, Fwi-Song (cannibal), Twenty-seventh (redshirt), First (acolyte), Sarble the Eye, Ghalssel, Tengayet Doy-Suut, Wilgre, Neeporlax, Xoxarle, Unaha-Closp (drone), Stafl-Preonsa Fal Shilde 'Ngeestra dam Crose (Fal 'Ngeestra), Gimishin Foug.

Names from The Dispossessed: 46. 32 Pravic names along the same lines:

Bebach, Chagvol, Chapis, Chekoks, Chigvigv, Dupin, Gvegob, Komush, Kvakot, Kvushab, Lassep, Logog, Makul, Mokob, Pabar, Palab, Pechol, Pevip, Pichok, Reshub, Shishek, Shivaks, Shumin, Sikvok, Siroks, Skemun, Sotat, Suvun, Trapan, Trekvish, Vabon, Vavun.

Those are too similar.

Names from gen24.py:

Slilg, Floung, Gealgpru, Grapvirveag, Quiphdrusp, Kalgspricloost, Theagproohool, Shophslylg, Wimsplup, Houbdrid, Quyngdryg, Azpream, Greashgrystscoul, Yynpraint, Speazwhof, Scrussbres, Bealggusssprush, Kelpluf, Kasplach, Ploonchag, Prontseng, Graispclooshchuss, Kreassweaxheaph, Nyz, Yoontstaif, Doontdoup, Noosh, Clackblum, Dechspat, Plosttaix, Splez.

Those are too silly.

Names from gen24.py generated with some English probabilities, with real words removed:

```
./gen24.py 40 /usr/share/dict/words .1 | awk '{print $1}' | LANG=C sort
```

Achem, Dep, Dirness, Atafli, Essdi, Enlini, Rera, Tiomtru, Dercal,

Bii, Abhar, Difodent, Nibanspo, Lincalent, Semeed, Veskeno, Ingex, Lum, Onel, Chantcor, Iiz, Reelfin, Blocklo, Leconscrip, Ura, Satyrpo, Editdish, Igstrud, Cing, Inon, Harfa, Etordot.

Okay, I think that'll work.

Another candidate naming convention takes sequences of letters, numbers, and symbols that spell words in Spanish. For example, "5p" means "faint away" and would work well for the checkpoint-and-restart facility (as might "k+" "beds" or "ck" "dry"), "g+" means "jewels" and would work well for a packaging system, "kbo" means "headboard" or "header". And "ogo" means "glance" and would work well for the windowing system.

Oops, unfortunately "ogo" is already "an OpenFlow Network controller in Go", a handheld computer sold from 2004–2006, an abbreviation for "OpenGroupware.Org", and a mobile phone dating app similar to Tinder. So, without more detail, that name is right out. Something like "ogoak" ("ojeo acá", "a glance here") or "ogoc2o" ("silky glance") or "ogo☼o" ("ojeo solo", "just a glance") might work; none of them exist.

Wercam: a hard-real-time secure windowing system

Wercam securely multiplexes input and output between mutually untrusting applications while guaranteeing glitch-free animation and instant responsiveness to system commands 100% of the time, despite the efforts of malicious applications. Additionally, applications can upload executable bytecode to Wercam to provide glitch-free bounded-latency visual feedback to user actions without writing the whole application as a bounded-latency system.

It includes VNC and RDP clients for remote access to other graphical systems.

Wercam is about 1000 lines of real-time Leconscrip.

(The predecessor to 8½ was "a few hundred lines of source code using [Newsqueak]", and 8½ itself was about 5–15kloc, but 8½ supported 23 operations in /dev/bitblt and a bunch of font nonsense. Rio was smaller and simpler.)

See also files Scriptable windowing for Wercam (p. 1256), Window systems (p. 1335), and Real time windowing (p. 891).

Intranin: a hard-real-time secure microkernel

Intranin securely multiplexes the CPUs, memory, disk, and GPU between mutually untrusting application processes, which are isolated using objet-capability discipline, and furthermore securely enables hard-real-time control applications to meet microsecond deadlines 100% of the time.

Intranin is about 2000 lines of real-time Leconscrip, plus a few hundred lines of Abhar assembly for each platform.

Oops, "DEP" (the original name) already has a meaning in infosec, "data execution prevention": https://seclists.org/oss-sec/2018/q4/82 . Renamed to Shang.

Oops, "Shang" (the name replacing Dep) is already https://github.com/etherzhhb/Shang, free software for compiling C

to RTL. Intranin!

Atafli: hard-real-time secure terminal emulators

Atafli is a set of character-cell terminal emulators that are hardened against resource-exhaustion attacks. They run on Wercam and are useful for running older programs or providing access to remote machines, but when writing programs for BubbleOS, it's easier and provides better results to use Essdi.

Atafli is about 500 lines of real-time Leconscrip.

Essdi: a hard-real-time layout system

Essdi is a modern replacement for character-cell terminal output streams, preserving their ease of programming and guaranteed real-time performance (a particularly big contrast when compared to the janky pause-filled experience that is an HTML5 web browser) but providing enough of the CSS box model that it's easy to write things that look good. Most applications that run on Wercam use it to do their layouts.

At times, Essdi privileges responsivity over correctness.

Essdi is about 800 lines of real-time Leconscrip.

Editdish: A text editor

In the 1970s, text editors were separate application programs, but they turned out to be useful for handling the user interfaces of other application programs. Graphical user interfaces embedded tiny text editors all over other applications.

Enlini, a minimal TCP/IP implementation

Enlini implements enough of TCPv4, IPv4, and UDPv4 to support the minimal BubbleOS network services: an ssh client and server, a Bitcoin client, and an HTTP client and server. It uses a parser-driven approach reminiscent of the STEPS TCP/IP stack.

• Minimal cryptographic facilities, an ssh client and server, and a Bitcoin client, Rera.
• Encrypted swap and disk, Tiomtru.
• Unicode fonts, Dercal.
• Unicode text rendering including bidi and combining characters, Bii.
• A bootstrapping chain of compilers, including assembler, low-level, high-level, and very-high-level languages. The assembler, Abhar, is a variant of Forth; the low-level language, Difodent, is a subset of C; and the others, Leconscrip, are extended subsets of JS.
• A compiler for a subset of C++ sufficient to cross-compile GCC, Lincalent.
• A SAT solver which provides some of the horsepower for these compilers, Semeed.
• A virtual CPU machine set, with simulators for i386, amd64, RISC-V, and ARM(32), which runs all the other code, Veskeno.
• A hardware design for this CPU, Ingex.
• A library of engineering models of many different physical phenomena, Lum.

• A circuit design and synthesis system capable of synthesizing this CPU for certain Lattice FPGAs, Onel.
• An analog circuit simulation program similar to SPICE, Chantcor.
• A general mathematical optimization system that provides most of its horsepower, Reelfin.
• Emulators capable of running CP/M, MS-DOS, or Apple][programs, Blocklo.
• Nibanspo: An image-format library capable of reading and writing most common image file formats, including JPEG, PNG, TIFF, and GIF.
• Ura, A minimal web browser capable of fully rendering Wikipedia and Stack Exchange, though it lacks support for most modern web standards.
• Satyrpo: snapshots of Wikipedia and Stack Exchange for offline viewing.
• Igstrud: A fast, capable, secure web server.
• Cing: The ability to checkpoint and migrate either a full machine image or an individual process, which produces an executable.
• Inon: its full source code as a hypertext literate program.
• Dirness: A fast, secure, crashproof content-addressable filesystem that supports transactions.
• Harfa: A fast key-value store similar to LevelDB.
• Etordot: A minimal SQL database.
• Binate: A terser database query language than SQL, but equally powerful, better suited to interactive exploration.
• Atsoled: A fairly full set of data-wrangling facilities similar to Unix software tools, but implemented differently.
• Bonlar: An implementation of Git.

Bootstrapping sequence

Veskeno, the virtual machine, is, in some sense, the bedrock of the system; everything else runs within it. But Veskeno alone is unusable; you need programs for the virtual machine in order to do things with it. And for that you need at least some version of Abhar, the assembler, running.

Given a definition for Veskeno's instruction set and I/O architecture, this ought to be a matter of a few hours of programming (initially in C or Python), but coming up with the instruction set could easily take longer than that. Probably writing a few drafts is a good idea, but then I need to test each one with some kind of code generator.

In Abhar or low-level Leconscrip I can write a bootstrap interpreter for high-level Leconscrip, and then I can write a compiler for high-level Leconscrip in itself.

Topics

• Programming (p. 3658) (286 notes)
• Performance (p. 3621) (149 notes)
• Systems architecture (p. 3691) (48 notes)
• Programming languages (p. 3656) (47 notes)
• Instruction sets (p. 3526) (40 notes)

Some personal notes from February 2014

Kragen Javier Sitaker, 2014-02-13 (8 minutes)

It's the 13th of February. Today my friend Stace and her dad, who are staying at my house, took delivery of a portable air conditioner, which they'd bought on my behalf from the Coto department store around the corner. My apartment is cooling down now.

On Thursday, the 23rd of January, the official exchange rate had jumped from $6 to the dollar to $8 to the dollar, apparently as a result of a small sale of a few million pesos by Shell Oil. The official market is tightly controlled (payments for imports, for example, must be immediately liquidated in it, providing it with a large captive supply of dollars; and access to the peso side of the market is strictly limited) and, apparently as a consequence, has very little depth on the selling-dollars side. The central bank stepped in to force the closing price back down to $7.75 to the dollar, and the newspapers were full of dueling accusations the next day, Friday. The true price went up from $10 to $12, and the buy-sell spread widened to 15% or so. The government issued panicked pronouncements that the restrictions on Argentines buying dollars would be lifted.

In the midst of this, I went to check out the price on this portable air conditioner on Saturday, the 25th of January. It was priced at $4900, which is US$410 at the new true rate, or US$490 at the old true rate, or US$600 at the new official rate, or US$800 at the old official rate.

I was talking with my landlord during this time about repairing the current air conditioner or replacing it with a new one, and he was taking a while to get back to me; so on Tuesday, the 28th of January, I went back with $5000 to buy the portable unit myself, so I wouldn't have to wait for him. Within those three days, the price had been increased by 40% to $7000, so I didn't have the money.

The promised restriction-lifting turned out to be kind of a dud; it entirely excludes the majority of the population (I guess the government figures that poor Argentines don't need to be able to save money) and allows high-income people to purchase only very small quantities.

The air-conditioner repairman called to tell me that the replacement compressor the apartment's unit needs no longer costs $800 or $900 like he'd expected, but rather $3000. The landlord asked if maybe I could pay the repair cost up front and then deduct it from my rent a little bit at a time over the next few months. I declined.

In an effort to stem the inflationary spiral, the government signed agreements with the big supermarket and department-store chains to freeze their prices at the levels of January 21, or in some categories (like home appliances) to freeze them at 7.5% above the January 21st levels. I returned to Coto hoping to find the air conditioner price reduced to $5375, but it was still $7000.

So I sent Stace with my money to buy the air conditioner and arrange for delivery while I was at work. She succeeded, but delivery

wasn't scheduled for another week.

So now I have a sort of monochrome R2-D2 in my bedroom belching out hot air onto my patio, cooling off my apartment and periodically urinating into a bucket I've placed next to it.

Around town there are posters with the faces and names of the heads of major chains: Walmart, Frávega, Coto, Shell. The posters blame the inflation on these chains' choice to raise prices, which seems plainly absurd to me in an environment where unions have been regularly getting yearly raises above the rate of inflation.

Thursday morning, I took a bus home from an outlying neighborhood. The fare was $8. When I arrived in Buenos Aires in 2006, the highest fare was $0.80. But this is somewhat of an overstatement of inflation, because I was only charged the cash fare rather than the $3.90 card fare because my prepaid bus card was empty.

Some months ago, a garbage truck caught fire in the street outside the office where I work. The fire ignited six cars parked along the street. They burned until nothing was left but metal and carbon. The heat blistered the stucco and paint on the buildings on the street and melted parts of the engine, which puddled in the street. A chunk of melted engine is now sitting on my bathroom vanity.

A week or two ago, I saw the last four of the six burned-out cars finally being loaded onto a tow truck. Now only the melted plastic and blistered concrete bear witness to the conflagration.

When Darius was here, I tried to persuade him to buy a set of reading glasses from a Senegalese street vendor, though without success. A couple of weeks ago, the Senegalese street vendors of Buenos Aires suffered a sudden setback: the police raided all their houses early in the morning to confiscate the watches, cheap jewelry, and eyeglasses they sell on blankets around town --- as well as the blankets themselves. They protested by cutting off streets, but since they are immigrants, public opinion is against them.

Macri, the mayor, used his new metropolitan police force to repress the various street vendors. This has been a project of his for some time, and in at least one case they have shot street vendors with rubber bullets to drive them out of an area.

Still, early-morning house raids seem like a new extreme of repression against such a previously tolerated activity. I wonder how much longer San Telmo's famous Sunday open-air market will last. Perhaps it will be spared, since it attracts so many tourists.

Macri's party, PRO, is popular here in Buenos Aires, but fortunately has little support nationwide.

The cockroaches are back. I've set out fresh roach-bait traps, and that seems to have slowed them down, but I still have a long way to go before they're eradicated.

I still haven't reassembled the hot-water heater. There's a bag somewhere in my apartment with the absolutely crucial faceplate and spring parts of it, which I haven't been able to find in months. This is starting to become a problem again as the weather cools down from sweltering to merely sweaty.

Yesterday, we went to look at an apartment that we could possibly share. It's 110 square meters with four potential bedrooms (one of which is rather too small, and one of which lacks a door); the cost is $4400 (US$400, say) plus $1300 expenses. Gorgeous parquet floors,

huge living room, well-equipped kitchen with granite counters, noisy. I was expecting the realtor to call me back today with information about the water damage in the walls and access to the roof.

The other apartments we've been looking at have been similar in size, but in large part since they're in richer parts of town, they're two or three times as expensive.

Topics

Executable scholarship, or algorithmic scholarly communication

Kragen Javier Sitaker, 2016-08-11 (13 minutes)

We are witnessing the birth of a new phenomenon I haven't seen named before. I propose to call it "algorithmic scholarly communication".

Alan Kay describes the birth of a new genre of media as proceeding in a series of stages:

> A new genre is established. A few years later a significant improvement is made. After a few more years the improvement is perceived as not just a "better old thing" but an "almost new thing" that leads directly to the next stable genre.

This essay identifies an "almost new thing" which is beginning to transform the nature of scholarly communication. First I provide some background, including my personal view of the principles underlying scholarly communication; then, I give some thought-provoking examples of the current state of the system; and then, I project where it seems likely to take us.

Background: what are scholars, and what is scholarly communication?

A hundred and thirty generations ago, we began to understand formal mental processes when Thales and the Pythagoreans including Socrates formulated proofs in geometry in terms of logical arguments. Over the next ten generations, Mozi, the Pythagoreans, and Aristotle developed the study of logic as an object in itself.

The academy as we know it is, in name, an homage to the estate of Plato, Socrates' student; in structure it descends from medieval seminaries of the Roman Catholic Church, and indeed some of those seminaries are universities today. It has always aimed at acquiring and diffusing knowledge, both in the sense of practical skill (the arts) and knowledge of what is objectively true (the sciences), though modern epistemology has demoted Christian theology and holy writ in general to the status of myth and metaphor rather than an objectively-true cosmology, tempting us to scoff at the debates of the European medieval Scholastics; it's easy for us to forget that angels and demons seemed as real to them as states, money, and corporations do to us.

XXX etymology of "university"?

After 2600 years of work, we at last succeeded in fully formalizing logical reasoning and other formal mental processes during the 20th century; the product of this formalization was the digital computer, which executes processes called "algorithms".

What happened to mathematical tables?

We have already seen executable scholarship happen in many aspects of practical mathematics. I have on my bookshelf a 1965 book by R.S. Burington entitled *Handbook of Mathematical Tables and*

Formulas, first published 1933, LCCN 63-23531. It tells me things like this, neatly organized into tables:

$\log \cos x \approx -x^2/2 - x^4/12 - x^6/45 - 17x^8/2520 - \ldots$ when $x^2 < \pi^2/4$
roughly 53030 Americans of every 100 000 live to age 63; 1800 die that year
the cotangent of 38.5° (i.e. the reciprocal of the tangent) ≈ 1.2572
$\nabla^2 S$ is defined as $\partial^2 S/\partial x^2 + \partial^2 S/\partial y^2 + \partial^2 S/\partial z^2$
the logarithm of the cotangent of $17^\circ 21'$ is 0.50526
an endomorphism is a homomorphism of a system onto itself or part of itself
$0.15^\circ \approx 0^\circ 9' 00''$
the definition of the cumulative discrete probability as a sum $\Sigma_i f(x_i)$
formulas for the sides & angles of a triangle given 2 angles & 1 side
$\int dx/(x\sqrt{(ax + b)}) = 1/\sqrt{b} \log((\sqrt{(ax+b)}-\sqrt{b})/(\sqrt{(ax+b)}+\sqrt{b}))$ for $b > 0$

The first half of the book is a brief but complete summary of the most widely useful branches of mathematics; the second half consists of numerical tables. The first "mathematical table" of this sort was assembled by Aryabhata in the fifth century CE; it consisted of 24 first differences of the sine function. Calculating and disseminating such tables — crucial for practical manual celestial navigation, artillery warfare, trigonometric calculation in general, and statistics — was a major effort of mathematicians for centuries. It inspired Babbage's original plan to automate computation in 1822; computing mathematical tables was the justification for funding one of the three World War II projects that did successfully automate computation, the ENIAC. The ENIAC was designed to compute mathematical tables for artillery firing.

The other two projects were Konrad Zuse hacking in his Berlin apartment (and later with government support) and the secret British cryptanalytic Colossus machine; arguably the Harvard Mark I ASCC, based on Babbage's work, was a predecessor to the ENIAC. The Mark I was also used to compute mathematical tables.

After fifteen centuries, the publication of books of mathematical tables essentially ceased in about 1975, at which point the numerical tables were replaced in common use by HP scientific calculators, where they had not been replaced earlier by larger and more expensive computers. (Bowditch's American Practical Navigator is one of the few remnants that still includes mathematical tables, even in the 2002 edition, the latest as of this writing; pp.565–727 are devoted to tables, most of which are mathematically calculated rather than empirically measured.)

The kind of mathematical scholarship formerly devoted to publishing mathematical tables is nowadays devoted to publishing software instead. Pages 325 to 344 of Burington's handbook are devoted to a table of n^2, n^3, $\sqrt{n}$, $\sqrt{(10n)}$, and the cube roots of n, 10n, and 100n; given a computer that can multiply integers, the square-root columns of this table can be replaced in practice with a single line of C code, which implements the "Babylonian method" of computing square roots given by Hero of Alexandria (Ἥρων ὁ Ἀλεξανδρεύς) a hundred generations ago:

```
g,h;q(n){h=n;if(n)do g=h,h=(g+n/g)/2;while(g-h+2&~3);return h-=h*h>n;}
```

However, this line of C code, although it works and replaces two sevenths of those twenty pages of numbers, falls short of many of the

ideals of scholarship, as I will explore below.

units(1)

Several times a week, I use GNU units(1), the GNU version of the BSD (?) units(1) program, which is a simple desk calculator program which contains a comprehensive database of over two thousand units of measurement, compiled over decades by the tireless scholarship of Adrian Mariano at Cornell. The original uses were things like this:

```
You have: 60 miles per hour
You want: furlongs per fortnight
        * 161280
        / 6.2003968e-06
```

But units(1) also allows me to immediately calculate things like how many minutes of arc the sun subtends on average:

```
You have: radians (2*sunradius)/sundist
You want: arcminute
        * 31.988013
        / 0.03126171
```

Or, in case I've forgotten, that a mole of ideal gas at STP occupies 22.7ℓ:

```
You have: gasconstant
You want:
        Definition: 8.314472 J / mol K = 8.314472 kg m^2 / K mol s^2
You have: gasconstant * 1 mole * tempC(0) / 100 kPa
You want: liters
        * 22.71098
        / 0.044031565
```

Or that Israel's new Sorek reverse-osmosis desalination plant, which is reported to use 70 atmospheres of pressure for the reverse-osmosis process step, is doing 7.1kJ of mechanical work per liter, putting a lower bound on the economic cost of the freshwater obtained:

```
You have: 70 atmospheres
You want: kilojoules per liter
        * 7.09275
        / 0.14098904
```

And that that is US$200 per megaliter or US$240 per acre foot if the energy used to drive it costs US$100 per megawatt hour:

```
You have: 70 atmospheres * US$ 100/MWh
You want: US$ / megaliter
        * 197.02083
        / 0.0050756054
You have: 70 atmospheres * US$ 100/MWh
You want: US$ / acrefoot
        * 243.02308
        / 0.0041148356
```

I can calculate that 10 amperes of electrical current through a round 22-gauge wire amounts to 31 amps per square millimeter; I have to be careful here, because units's `wiregauge` unit gives me the wire diameter, but its `circlearea` unit wants the radius:

```
You have: 10 amps / circlearea(wiregauge(22)/2)
You want: amperes / mm^2
        * 30.718763
        / 0.032553394
```

If I include the resistivity of copper and the equation $P = I^2R$ in the calculation, I can get the heat production:

```
You have: (10 amps)^2 * 16.78 nanoohms * meter / circlearea(wiregauge(22)/2)
You want: watts/meter
        * 5.1546084
        / 0.19400116
```

(That's enough heat to be dangerous if used as electrical wiring, but not enough to use as a heating element, except maybe in an electrical blanket or something.)

That was actually my second try at that calculation; on the first try, I accidentally calculated the voltage drop per meter instead, which units was able to catch, but not provide a useful error message for:

```
You have: 16.78 nanoohms * meter * 10 amps / circlearea(wiregauge(22)/2)
You want: watts / meter
conformability error
        0.51546084 kg m / A s^3
        1 kg m / s^3
You have: 16.78 nanoohms * meter * 10 amps / circlearea(wiregauge(22)/2)
You want:
        Definition: 0.51546084 kg m / A s^3
```

It happens that kg m / A s^3 is equivalent to volts per meter, but that's not at all obvious from looking at the result.

You can see that as the examples get more complex, more and more of the knowledge is in my head, and less and less in the software.

Actually, I didn't have the resistivity of copper memorized in my head; I looked it up on Wikipedia and typed it in. There's a Wikidata entry for copper but it doesn't yet include electrical resistivity as a property. It does include its density, but in the format "8.94±0.01 Q15639371", which units(1) cannot parse; Q15639371 is the Wikidata name for "grams per cubic centimeter." There was an entry for copper in Freebase.com too, but I don't know if it included electrical resistivity.

Wolfram Vertical Line Alpha

Stephen Wolfram also has a broad vision of part of what I am calling executable scholarship. He calls it a "computational knowledge engine", and being Stephen Wolfram, he plans to own it, and his version doesn't cite any sources.

The Wolfram version is called "Wolfram|Alpha", but I decline to

use marketing-provided punctuation in the middle of proper nouns like "Yahoo!" and "Re/Code", because I don't hate my readers enough. Instead I will spell out the term as "Wolfram Vertical Line Alpha" when necessary.

Wolfram Vertical Line Alpha *does* include the electrical resistivity of copper, and given the input "resistivity of copper * (10 amps)^2 / 0.326 mm^2", it correctly computes 5.2 watts per meter. It also claims that the size of 22AWG wire is 0.324 mm², which differs by about 0.5% from the value given by units(1), and will return it given the input "22awg", but I haven't found a way to get Alpha to combine these pieces of knowledge in a single formula.

The aims of scholarship

What is scholarship?

valid trustworthy well-known

reproducibility self-correcting interlinked honest objective stable readable public reveals preferred embodiment not ignorant creative? documentable? replicable? peer-reviewable?

[Boyer 1990] proposed expanding the definition of "scholarship" to include not only the discovery of knowledge but also teaching, application ("the use of new knowledge in solving society's problems"), and what he calls "integration" ("where new relationships among disciplines are discovered", a definition broad enough to include any research). Boyer was unable to publish his proposed redefinition in a peer-reviewed venue, publishing it instead through a phony "Carnegie Foundation" whose declared purpose is to promote the interests of teachers, which is to say, university faculty who do not engage in scholarship; not satisfied with this, however, he additionally proposed to expand "scholarship" to include practitioners of a field in general, except those who use only old knowledge.

[Boyer 1990]: E. Boyer, *Scholarship reconsidered: Priorities for the professoriate*, published by the "Carnegie Foundation for the Advancement of Teaching."

Square roots

The line of code I gave above to calculate the square root of an integer falls short of the aims of scholarship for several reasons. It's hard to read (made unnecessarily so by a)

On the computer I'm typing this on, square roots are generally calculated by proprietary Intel hardware designs that I am not able to see, because using specialized hardware makes it much faster, and

tested using a function called sqrt_test.

. But the free software I run the computer on has an option to calculate square roots using addition, subtraction, comparison, bit shifts, and iteration, called _FP_SQRT_MEAT_E.

http://www.uclibc-ng.org/browser/uclibc-ng/test/math/libm-test.inc?rev=fdebbe2044653c5c84172524ed6e036d38716d88#L4446

Automatic testing

IPython notebooks

Algorithms published in CRAN

Research in emulation

Retracted results due to spreadsheet errors

Stellarium

SPICE

What comes next?

Consider my

Intertextuality, transclusion, and plagiarism

Topics

Hypothesis evolution

Kragen Javier Sitaker, 2019-12-17 (4 minutes)

David R. MacIver's generative testing system Hypothesis is an example of a new breed of "lightweight formal methods" software that I think represents a significant advance in the state of the programming art. Hypothesis works as follows: you state a hypothesis about your system's behavior; Hypothesis searches for a counterexample by, essentially, fuzzing the system; and if it finds a counterexample, it "shrinks" it by testing simpler and simpler examples until it finds one that it cannot simplify to a simpler counterexample. (That is, any simplification it tries results in an example that isn't a counterexample.) It tells you the simplest counterexample and records it in an example database and then always or nearly always retries it in the future so that you'll know when you've fixed the bug.

Contrary to 1980s QA dogma, random testing turns out to be an excellent way to find bugs, and Hypothesis's innovative approach to shrinking counterexamples is very helpful in getting useful bug reports. But you know what would be even more helpful than a single minimal reproducing test case? A thorough characterization of the bug, telling you not only one case where it does happen, but a formal model of when it does and doesn't happen.

Evolution of a hypothesis population with adversarial experiment design

Suppose that your initial hypothesis is that no input sequence for your calculator program will produce a result that is incorrect by more than 0.01%, and random testing finds such a result by comparing against some kind of formal model of arithmetic --- perhaps another calculator program, one that doesn't have to run fast or use little memory, or perhaps some kind of sparer model. The testing has probably found a large set of cases that produce the correct result and a smaller set of cases that don't.

The initial hypothesis was that this second set would be empty. Now we could generate a set of new hypotheses --- for example, that all input sequences containing division will produce an incorrect result --- and test them against the experimental data. Hypotheses that do a better job of discriminating the two sets (per bit!) should be preferred, since the objective is to find a concise hypothesis that discriminates them perfectly. Once we have several hypotheses that discriminate perfectly, or nearly equally well, we can do more experiments: generate more examples and test them to see what the result is. We should prefer to spend our CPU budget on examples that do a good job of discriminating between our existing hypotheses. Once we have more evidence, we can generate more hypotheses, and vice versa.

The approach is very similar to generative adversarial networks: hypotheses attempt to evolve to predict the behavior of the system under test, while experiment design (test-case generation, example generation) attempts to confuse the existing hypotheses.

What form should the hypotheses take? There are many possibilities appropriate to different kinds of examples: predicate-logic expressions, regular expressions, neural networks, and context-free grammars, for example. If an example can be described by some kind of abstract syntax tree, tree regular expressions can capture interesting properties of that tree, especially if augmented by predicate-logic expressions that can examine node properties. The whole field of genetic programming can be applied to hypothesis generation.

This may provide a useful way to take advantage of increased CPU power (see The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306)).

Genesis

This was inspired by a bug I'm investigating tonight which Hypothesis is happily finding seven hundred times a day without providing much new insight into why it happens. Of course, conventional debugging approaches based on inspecting the code are necessary and sufficient, and I probably should spend some time on them in this case rather than continuing with black-box testing, but I thought maybe I could write a simple model of the bug as I did with another recent bug to exclude it from the testing Hypothesis is doing. This turned out to be more difficult than I anticipated, and I got to thinking about what kind of software would make the process easier.

Topics

- Programming (p. 3658) (286 notes)
- Hypothesis

Storing dry bulk foods in used Coke bottles

Kragen Javier Sitaker, 2012-10-15 (updated 2012-10-21) (5 minutes)

I've been putting dry bulk foods into used Coke bottles for storage. The place where I'm staying for another week now has bottles of soybeans, polenta, flaxseeds, whole wheat flour, white flour, rice, and lentils. This makes me happy. I feel like I'm creating order out of chaos. The bottles look a lot prettier than half-full bags of lentils, and they also exclude cockroaches and beetles better. (Sometimes beetles can get into even sealed-shut plastic bags.) They also seal hermetically, preventing oxygen and moisture from the air from attacking the contents, and for foods that come in paper bags rather than plastic, there's the additional advantage that they protect the food from possible roof leakage --- not a merely theoretical advantage in the places where I've been living!

One of the bottles I have to work with is not a Coke bottle, but a bottle for Terma, a sort of bittersweet herbal infusion that's popular here. The plastic is green, so a lot of foods look horrible inside of it, despite its graceful form, which looks rather like a thigh modified to be rotationally symmetric. So rather than using it for storage, I scissored off the top of the Terma bottle to use it as a funnel for filling the other bottles with, then heated up the neck with a stove flame in order to squish it down a bit so it fits inside other bottle necks --- not actually necessary since it's not a big deal if I spill a few drops of flour or a lentil or two, but helpful.

(Another Terma bottle got used for yerba mate instead, which looks fine when tinted green.)

Washing the bottles out is kind of a pain because they don't dry easily. I've taken to dripping a few drops of 96% ethyl alcohol into them and shaking it around to take up the water. Then I can just pour it out, because it has a lot less surface tension than water. Also, it's reassuring that it's a mild disinfectant.

The more time-consuming step --- which I'm not doing this time around --- is removing the glue that keeps the labels on. Terma labels come off with just water, leaving only a little residue, but Coke and Pepsi label glue (at least currently in Argentina) won't come off even with alcohol; turpentine works, but also penetrates the bottle and can leave the food tasting like turpentine. Filling the bottles with cold water before turpentining the labels off reduces this problem.

A friend suggested freezing foods like flour for a while before storing them, in order to kill any beetles that might be hiding in them, which seems like an eminently sound idea to me. After all, even if the bottles keep the beetles or moths contained, you still lose a bottlefull of food, which could have been avoided if you just had to discard a single beetle.

In order to limit the lifetimes of foods stored this way, I label the bottles with the name of the contents and the date it was bottled. So far I haven't found a good way to do this. Masking tape is workable, but the adhesive decays over time, and two years later when I use the bottle for something else, I'm left with stinky dried adhesive on the

outside of the bottle. Also, the masking tape itself is kind of ugly. Permanent marker works, but you can't take it off, and I fear that the solvent will flavor the food. Maybe paper labels with some kind of water-based adhesive, such as wheat paste, would be best, but first I'm going to try tying bows around the necks of the bottles using a wide cloth ribbon I write on.

Many people prefer glass bottles for this kind of thing because they're prettier. Plastic Coke bottles have three advantages over glass bottles:

• If you knock them off the shelf, they don't break. In fact, if you run over them with a truck, they don't break. If you hit them with a baseball bat, they don't break. They're nearly bulletproof.
• They weigh much less.
• For things like flour which can entrain air, you can squirt them out of the bottles like liquids by turning the bottle upside down and squeezing.

Topics

Evaporation chimney

Kragen Javier Sitaker, 2013-05-17 (13 minutes)

Fundamental principle of operation of an evaporative chimney

A chimney is a bistable fluidic structure, sort of like a siphon: once the chimney fills with hot air, the higher pressure from outside the chimney forces the newly produced hot air up the chimney. Hot air is a lot less dense than cold air. Air at 275°C is half the density of air at 0°C, because it's twice the temperature.

Drying things out is a major problem in day-to-day life: laundry or dishes after you wash them, fruit or other food you want to preserve, parts of your building you don't want to grow Stachybotrys mold and poison you, and so on. Dryness, cold, and poison are the three major ways to inhibit life on media that could otherwise support it (e.g. mold, bacteria, insect predation), and of these, dryness is by far the preferred option in many cases.

Room-temperature evaporation is usually adequate to dry things out enough that they can't support life. Usually, the main determinant of the speed of room-temperature evaporation is air circulation. With no air circulation, relative humidity rapidly reaches 100%, and evaporation stops. Even a fairly weak draft is often sufficient to reach adequate evaporation speeds.

But evaporation, like heat, decreases the density of air. My calculations below suggest that you could construct an evaporation-driven chimney that would provide adequate air circulation even without a source of above-ambient temperature, probably using laminar rather than turbulent airflow.

In a sense, this is an evaporative downdraft cooltower, meant to make things dry rather than cold and wet. The only difference is that you saturate it with heat instead of humidity.

Overall parameters and results

The vapor pressure of water increases exponentially with temperature, which means that any such chimney device will have a critical temperature below which it will basically not work at all, and it's much harder to build one that will work at 0° (0.6 kPa vapor pressure) than 16° (1.8 kPa), 25° (3.2 kPa), or 35° (5.6 kPa) — you have an entire order of magnitude of density difference here.

It's probably a good idea to shoot for 16°, since 0° sounds too hard, and higher ambient temperatures in places where people live are often a result of humidity-induced greenhouse effect, so in practice the dryer may not work that much better.

Of course, evaporation also cools the air and the thing from which the water evaporates, but you can passively warm the cooled air back to room temperature — it's just a matter of providing sufficiently high thermal coupling. At sufficiently high airflow, this heat transfer can be the limiting factor in drying things out. For this exercise, I will simply assume that heat transfer is adequate.

With these assumptions, the device seems quite practical. It seems

that you should be able to maintain 25 cm/s airflow with a 5.4-centimeter evaporation chimney, numbers which don't really depend on the chimney's height, according to the calculations below. However, it turns out you can get even stronger airflow by operating it as an evaporative downdraft cooltower!

Fluid dynamics calculations

Water molecules weigh 18 daltons, while the other major air molecules weigh 30 and 32 daltons. So pure water vapor weighs about 41% less than dry air.

So at 16°, air at 100% relative humidity and one sea-level atmosphere (100 kPa) will consist of roughly 1.8% water, which means its density is decreased by 41% × 1.8% = 0.74%. If your incoming air already has 50% relative humidity, which is common, you're down to 0.37%; if the air density is 1.23 kg/m³, that's about 4.5 mg/ℓ, 4.5 µg/cc, or 4.5 g/m³.

Okay, how much pressure can you get out of that? At Earth surface gravity, it works out to about 45 mPa/m, which is a pretty tiny pressure!

How much laminar airflow can you get out of that? Well, it depends on your air duct size. The Darcy-Weisbach equation says your pressure loss to friction is f[D] ρLV²/(2D), where f[D] is the Darcy friction factor (64/Re in the laminar case), ρ is the density of the fluid, L is the duct length, D is the diameter, and V is the velocity.

Conveniently, or not, the chimney length here drops out: our pressure difference is L × 45 mPa/m, proportional to the chimney length, and the Darcy–Weisbach pressure drop is also proportional to the chimney length. So we have 45 mPa/m = f[D] ρV²/(2D); if we divide both sides by the density of air, we get 0.037 m/s/s = f[D] V²/(2D).

Will it be laminar? Laminar flow dominates at Reynolds numbers below about 2000 or 3000. The Reynolds number Re in a pipe is VD/ν, where V is velocity, D is diameter, and ν is kinematic viscosity, measured in e.g. m²/s. The kinematic viscosity of air is about 1.5×10^{-5} m²/s, so the laminar-turbulent transition takes place (assuming Re = 2000) at around VD = .03 m²/s. That is, if D is 3 cm, it takes place at 1 m/s; if D is 30 cm, it takes place at 10 cm/s; if D is 1 m, it takes place at 3 cm/s. So you can get it to be turbulent by using a big enough duct, even without changing the velocity.

If we figure on a Reynolds number in the middle of the laminar range, giving us a comfortable safety factor, f[D] is about 0.06. Solving for V², we get

$$V^2 = 2D \times 0.037\ \text{m/s/s} / 0.06$$

or simplified

$$V^2 = D \times 1.2\ \text{m/s/s}$$

so our actual air velocity is

$$V = \sqrt{(D \times 1.2\ \text{m})}/\text{s}$$

assuming that Reynolds number.

So, uh, that assumption gives us a duct size, no? I was assuming $VD = .015\ m^2/s$, so $.015\ m^2 = D\sqrt{(D \times 1.2\ m)}$. Square both sides (adding meaningless complex solutions) and you get $.000225\ m^4 = D^3 \times 1.44\ m$, so $D = (.000225\ m^3 / 1.44)^{\wedge}⅓$, or about 5.4 cm.

Okay, so if you have a 5.4 cm wide evaporation chimney, you get $\sqrt{(.054\ m \times 1.2\ m)/s} = 0.25\ m/s$ airflow from the tiny pressure difference from evaporation, which is pretty respectable.

You can make the duct a bit bigger (up to twice the Reynolds number, which is growing as $D\times\sqrt{D}$, so you can make it up to 59% wider: up to 8.6 cm, which would have 2½× the cross-sectional area and 1.26× the flow velocity, for a total of three times the volumetric flow) before you hit a slowdown from turbulent airflow, at which point you have to make it a lot bigger to compensate. Eventually, the total turbulent airflow will exceed the possible laminar airflow, but it's probably better to plan to stay well into either the laminar or the turbulent regime on the Moody diagram, avoiding the much lower airflow just above the transition. If you want to stay in the laminar regime but need more total airflow volume, you can have a number of small chimneys, perhaps a grid of partitions inside a larger space.

Such partitions would also help prevent the moist air coming up the chimney from mixing with dry air descending from above, which will reduce the pressure difference, but not the Darcy–Weisbach loss, thus reducing the flow rate. In the limit of an infinitely large chimney, you will have a plume of water vapor rising slowly from your moist object and gradually diffusing into the air, while nearby air remains stationary or sinks.

$0.25\ m/s \times \pi \times (.054\ m/2)^2 = 0.00056\ m^3/s$ total airflow in one of these chimneys. That's about 0.69 g/s of total air per second, of which 0.37% is the water we're extracting: 2.5 mg/s per chimney, or about 9.2 grams of water per hour per chimney. So to evaporate a liter of water in four hours — suitable for drying laundry or fruit — you might need 27 such chimneys, perhaps a 6×5 grid inside a larger 30×30 cm duct, for a total of $.015\ m^3/s$ or 32 cfm, somewhat less than a standard bathroom ventilation fan.

Startup

As I said, chimneys are bistable: once they have a draft, they keep sucking (much like standing armies), but until they're full of the lower-density fluid, they don't suck. So how do you start the chimney?

The usual approaches are using a fan and using an inverted funnel.

The fan works in the obvious way, but if you have the fan, you don't really need the chimney. The chimney is quieter and uses less energy, though.

An inverted funnel leading up into the chimney can capture individual chimneyless updrafts of moist air from below and fill the chimney with them. This will only work if the moist air updrafts are sufficient.

Materials

It barely matters at all what materials an evaporative chimney is made from. It could be sheet metal, paper (as long as it doesn't get rained on, or if it's greased to protect it), silk cloth, brick, or

polyethylene sheeting. Soft materials will need some kind of external support.

If you're venting it outside, you might consider it desirable to curve the chimney at the top, or put a roof over it, so that rain doesn't fall into it.

In many cases it might be beneficial to use a transparent material.

Heat flow

Earlier I said I assumed heat flow was adequate. Is that reasonable?

Heat flow is really critical for two reasons: first, the 0.5%/° change in air density with temperature could easily overwhelm the 0.4% change in air density with humidity, making your "chimney" work *backwards*; and second, if the thing you're trying to dry out is cold, water on it will evaporate really slowly, which will give it more time to rot before it dries out.

Evaporating 2.5 mg/s of water is about 10.4 mW of heat dissipation, so you suck up about 10.4 mW of heat per chimney. The earlier-suggested 27-chimney assembly thus sucks up about 280 milliwatts.

A glance at a psychrometric chart shows that the temperature drop of the air at 15° and 50% humidity is about 5°. Air is about 1.01 J/g/K, so that's about 5 J/g, and at 0.69 g/s, that's 3.4 watts. This differs from my calculation above by a factor of 12, so one of them must be completely wrong. Maybe both.

You can deliver the heat from the environment by convection, conduction, and radiation.

Room-temperature radiation is more than adequate to replace a few hundred milliwatts (it's on the order of tens of watts per square meter — Stefan–Boltzmann 57 nW/m²/K⁴ gives you 390 W/m² at 15°C, and perhaps more relevantly the derivative is about 5 W/m²/K), but only if the objects you're drying is mostly exposed to radiation from other objects that aren't drying. That means they have to be spread out flat, and you don't want to put them under glass.

Convection will replace the missing heat, but only at the cost of the five- or ten-degree temperature drop in the air, which will keep the chimney from sucking air up.

Conduction is likely to be the best approach in many cases. It can take several forms: metal plates maintained at room temperature by a flow of water or air on the non-wet side, say by running water through pipes or by funneling wind past them; thick metal fins connected to a large temperature reservoir; or even flat heat pipes.

Backwards

But wait! If you have to struggle to overcome a 5° temperature drop, which produces a countervailing effect that's four times as strong, why not use the temperature drop? Instead of a moist-warm-air chimney above your drying objects, you have a cold-air drainpipe below. Your drying objects don't dry as fast, since they're colder.

As a bonus, you get free cold air, which can be used by itself to cool things to retard decay, or to improve the efficiency and delta-T of a refrigerative cooler.

Sunlight

Yeah, if you have the chance, some sunlight on whatever you're trying to dry will help a lot. Then you can get not only more-than-adequate heat flow, on the order of 1000 watts per square meter instead of tens, but also enough heat to make the chimney run purely on heat.

Topics

- Physics (p. 3632) (119 notes)
- Thermodynamics (p. 3747) (49 notes)
- Household management and home economics (p. 3504) (44 notes)
- Fluid dynamics

Compression with second-order diffs

Kragen Javier Sitaker, 2014-04-24 (3 minutes)

If you're compressing a numerical sequence, such as a digitized sound wave, you only rarely see exact repetition of sample number sequences; the smallest noise or varying DC offset (i.e. low-frequency noise) will cause all the samples in the sequence to be different. As a result, schemes like LZ77 don't do well at compressing digital signals.

Even the very simplest numerical signals fail in this way. For example, if you have a series of 16-bit samples, each being 258 times the sample number, it will repeat every 131072 bytes; but `gzip -9` can only compress it by 1.4%. (bzip2 -9 does better, compressing 8:1, partly because of its bigger block size; bzip2 -1 only compresses 2:1.)

But if you take the second-order finite forward differences of this sequence, they will be 0 258 0 0 0 0... which is very highly compressible. In fact, even memoryless schemes like Elias gamma coding will work somewhat well at this point, about as well as bzip2. But applying gzip to the above sequence gives you 961-fold compression, while bzip2 gives you 10280-fold compression.

In general, we should expect this to be the case for mostly continuous functions of time: the first and second-order forward differences should be much more highly compressible than the original sequence.

From a Fourier perspective, the finite-forward-differences operation is a single-pole high-pass filter with no knee: it just keeps attenuating at 6dB per octave all the way to DC. Two iterations of it give you 12dB per octave, strongly attenuating the low-frequency content and leaving you only the high-frequency content to encode.

You could ask why only second-order rather than higher orders: if a second-order predictor is good, wouldn't a fifth-order or twentieth-order predictor be better? It might be, depending on your data. The order multiplies discontinuities: a single impulse becomes two opposite impulses in first-order differences, three impulses in second-order differences, and twenty-one in twentieth-order differences. This kind of effect might outweigh the improved compression from removing higher-order regularities from the data.

The Fourier-analysis viewpoint might suggest that this is a lossy scheme, concerned with numerical approximation, but applied to a group (such as N-bit integers under mod-2**N addition) it is perfectly lossless. If you apply it to IEEE-488 floating-point numbers it will indeed be lossy, I think, because floating-point numbers are not a group. (Right?)

This is somehow related to linear predictive coding, in the sense that the finite differences at a given timestep are a linear function of the sample value at previous timesteps, and their "predictions" at future timesteps are also linear functions of them, and the Nth-order difference somehow represents a "residual" from that "prediction".

Various lossless audio compression codecs (Shorten, AudioPaK, and FLAC, for example) use linear prediction in this way to decorrelate

the samples they are compressing.

Topics

A plotter language of 9-bit bytes

Kragen Javier Sitaker, 2017-05-29 (updated 2017-06-01) (11 minutes)

Suppose we're in an alternate history where Japan won WWII due to some grievous miscalculations by the US. Computer technology developed anyway, somewhat later, on 18-bit machines like the PDP-4, PDP-7, and PDP-9, with 9-bit bytes, accommodating potentially a 512-bit character set — not enough for single-character kanji, but plenty for hiragana, katakana, and some common kanji. The electromechanical output device of choice in 1970 is not a teletype, but a kind of pen plotter optimized for curve plotting, with a scotophor CRT memory tube (like a Skiatron) as a higher-speed and less material-consuming alternative.

Instead of having two motors driving a teletype carriage across the paper, a third motor to select a letter from a typewheel, and a solenoid to fire a type hammer, the plotter has a pen-down solenoid and X and Y motors (counting from the upper right corner of the page) servoed from, nominally, analog voltage signals with a SNR of some 26dB, providing an effective resolution on the paper of some 2000×2000. These position signals are controlled in two ways: by setting them directly from a 12-bit DAC and by being changed over time by a velocity signal, which is also nominally an analog voltage signal with a 26dB SNR; it, too, can be set directly from a 12-bit DAC, but is changed over time by a (26 dB SNR) acceleration signal, which is an analog voltage signal that can only be set directly or zeroed by a reset command. This allows the plotter to plot perfectly smooth curves by setting only the acceleration signal periodically as the motors move under servo control.

There is additionally a "form feed" command, which initiates a change of page, either through manual operator intervention or through an automatic motor mechanism. The scotophor tube implements this by erasing the image.

This gives us 10 basic commands: x←, y←, x'←, y'←, x"←, y"←, reset, form feed, pen up, and pen down. Additionally, there's an 11th command, which idles the command stream for the given amount of time to allow the pen to catch up. The basic plotter command word, then, is an 18-bit word consisting of a 4-bit order code field, a 12-bit operand field, and two extra bits.

The "reset" command sets the velocity and acceleration to 0, but doesn't change the position.

In our timeline, the 1981 CGA had 128 kilobits of framebuffer; the US$1200 1976 ADM-3A had 16 kilobits of ASCII screen contents buffer (not even inverse video!). We're supposing that this plotter thing became standardized in 1965 at a budget of about US$50000, in a timeline where Moore's Law's 18-month capacity doubling was about five years behind our own. In our timeline, the appropriate amount of memory for computer terminal equipment was about one bit per 7.5¢ (total cost of the equipment, not the cost of the memory) in 1976, which would be 1981 in the Japan plotter timeline. So 1965 would be 16 years back from that, roughly 10 capacity doublings back: US$75.00 per bit. So the plotter has a digital memory budget of about 32 of these 18-bit words, or 64 9-bit bytes.

A terrible problem for terminal equipment at the time was that speeds of 10 characters per second were demanded (this was the major upgrade in the Teletype Model 28, introduced in our timeline in 1953, and a widely used computer console until about 1970 in our timeline; the Friden Flexowriter used as the console for the LGP-30 was also 10 6-bit characters per second) but it was challenging to transmit data over long distances at more than 300 bits per second. Much of Unix's user interface design (for example, the absence of success messages, the two-letter command names, and the "?" error message from ed, The Standard Editor) derives from this. We can suppose that the Japanese would be satisfied with 5 characters per second, but instructing the plotter to draw a single character like "ア", "ラ", "マ", or "ハ" involves about four or five vertices per character, and each vertex requires typically a new x/y position, perhaps a new x'/y' velocity, and often pen up or down. This means we need three or four commands for each vertex, and maybe 16 such commands per character, totaling 288 bits — almost an entire second to transmit in full!

There are about two strokes and two non-plotting movements per character, so the plotter pen also needs to be able to move the distance of a typical character stroke 20 times per second, or 50 milliseconds. If the character box is 5 mm square, and the typical stroke length is about half of that or 3 mm, we need a 60 mm/s pen movement speed, which seems rather slow; we can probably specify 240 mm/s, giving us the physical ability to plot some 20 katakana characters per second, rather than only 5, for which we would also need to be able to execute 320 commands per second. If we presume that our paper is shiroku-ban size 4, 264 mm × 379 mm (implying ≈200μm servo resolution), it will take us an entire second to move from one side of the paper to the other, and the paper can hold 52 columns of 75 characters, which will take some 195 seconds to finish plotting, which is an entirely reasonable kind of interval for a human operator to be asked to manually change the paper. (If we analyze a hypothetical analog electronic PID control system for controlling the pen position with a mechanical system whose acceleration is low-pass filtered by its mechanics, we can derive the kinds of glitchy distortions likely to result in the characters.)

If we figure that a typical stroke length is 3 mm at 240 mm/s, the pen needs 12.5 milliseconds to execute it, so millisecond resolution is likely adequate for the "delay" command; to be safe, let's say we count cycles of a 8kHz clock, so the maximum delay with a 12-bit operand is about half a second. Of course, later plotters that manage higher speeds, and the CRT version, will virtualize these "milliseconds".

(What should the scale for the v and a components be? Like, how many virtual milliseconds should x'=1 take to increment x by 1?)

We have a relatively large discrepancy here between the hypothetical mechanical device's physical capabilities to execute 320 commands per second and the hypothetical serial link's physical capability of 300 *bits* per second. In the other direction, in our timeline, the (18-bit!) 1959 PDP-1 could execute about 93,000 instructions per second, and presumably a similar stack-based machine could execute about 200,000 instructions per second, or maybe 20,000 if it were much smaller and therefore less powerful than the PDP-1's

2700 transistors, 3000 diodes, and 4096 18-bit words of core.

Could we advantageously use this powerful memory of 64 9-bit bytes or 32 18-bit words, or maybe a few times that, and the potential of executing tens or hundreds of thousands of instructions per second, to bridge the gap between the 300-bit-per-second serial link and the 320 drawing operations per second. Maybe you could even take full advantage of the curve-drawing abilities of the hardware with a font ROM made with PCB inductance, like the HP 9100A's microcode PCB (512 64-bit words totaling 32 kibibits, 1968 in our timeline, US$5000, 23 registers of core RAM; projecting forwards to 1973, backwards 8 years to 1965 to get a factor of 6 Moore's Law reduction, and up by a factor of 10 to account for the larger budget, we get 53 kibibits), or core ropes, like the Apollo AGC (1966 in our timeline, astronomical budget).

The GreenArrays C18 core has (not counting its stacks) 128 18-bit cells in its memory address space, of which 64 18-bit cells are RAM and 64 18-bit cells are ROM; disregarding literal cells, each one normally has four five-bit instructions packed into it, with the fourth instruction having two implicit zero bits. This is only a factor of two greater than the RAM budget of 32 18-bit cells I suggested above, and although it is somewhat limited in what you can achieve on a single core, I think a smaller C18-like processor would work fine as a controller for this hypothetical plotter. Normally it would execute 18-bit instruction words received over its serial port, which could read and write its 64 cells of RAM, and also make calls to routines loaded into its RAM and routines already present in its ROM; if it can execute 20 000 instructions per second, then it has time to execute some 63 processor instructions per necessary plotter instruction or 1000 processor instructions per plotter stroke, and if our serial connection gives us a start bit, 9 data bits, and a stop bit per 9-bit byte, at 300bps we can receive 27 bytes per second and execute 733 processor instructions per serial byte. Two such bytes should be adequate to invoke a subroutine to draw a character.

This architecture would actually allow font ROMs with a significant number of kanji, since you could define and invoke subroutines for radicals with bounding-box parameters.

Of course, this approach also allows you to write characters at arbitrary non-character-cell positions, to draw arbitrary line graphics, and probably to draw stipple patterns.

(If the pen is replaced by a brush, it might make sense to use a proportional actuator like a servomotor or voice coil to control trace intensity, which of course is easy on a scotophor.)

The physical system described here visits new points at about 20 Hz, which means it probably needs to have its first vibrational eigenmode well above 200 Hz to avoid unacceptable ringing (with a simple PID control system, anyway.)

Topics

• Graphics (p. 3483) (91 notes)
• Protocols (p. 3668) (21 notes)
• Displays (p. 3414) (13 notes)

The imbalance inherent in copyright systems

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

It's 2028. Last year, you visited your mother in a nursing home and sat with her and talked for a long time, which was sometimes difficult, because one of the other residents was listening to music on the radio on the other side of a curtain. You video-recorded the whole session and stored it on your computer. You told her you'd come back to visit the next week, but trouble came up at work, and you had to postpone the visit for another week. On your next visit, she no longer recognized you. A few weeks later, she died. Now your computer has flagged your video as copyright infringement and deleted it because of the music in the background. You would like to make sure this doesn't happen again.

Your brother stored a copy of your last recording of your mother before she died on Dropbox, but Dropbox also flagged it as copyright infringement and deleted it. After he protested publicly, Dropbox reported him to the police for child pornography, and indeed the police found child porn videos were in his Dropbox folder when they seized his computer. He swears he didn't put them there, but he's not having good luck convincing his cellmates of that. You would like to make sure you don't entrust companies like Dropbox with the ability to plant incriminating files on your computer.

Ever since you watched a video on YouTube about spearfishing off the coast of Panama, every web site you visit shows you ads for ammunition and "paracord", whatever that is. Your request for a visa to visit your sister in Spain is declined; no reason is given. You realize that watching videos on YouTube is dangerous.

Topics

What are Bitcoin's uses other than sidestepping the law?

Kragen Javier Sitaker, 2019-03-11 (updated 2019-07-05) (6 minutes)

(An edited version of a comment I left on the orange website.)

Someone asked: what are the uses of Bitcoin, other than sidestepping the law?

Rather than sidestepping the law, you can be sidestepping official corruption or crime. There are lots of cases where people who are supposed to be enforcing the law instead use their law-enforcement powers for their own benefit, sometimes in ways that are illegal, officially speaking, but in an unenforced way.

It's perfectly legal for me to, for example, donate money to WikiLeaks. But Visa and Mastercard aggressively canceled every merchant account that allowed people to use Visa or Mastercard to donate to WikiLeaks, apparently because of secret pressure from US government officials who were unhappy about some of the things that WikiLeaks revealed. The Snowden revelations — which may not have been legal, so they are peripheral to the question — very likely wouldn't have happened without WikiLeaks being able to support him, which they could only do because they could receive Bitcoin.

Similarly, many Patreon accounts are getting canceled because their owners have published things that are embarrassing to Patreon, but not illegal (typically, racist remarks). If Bitcoin becomes mainstream, there won't be a centralized authority like Patreon that is in a position to end people's livelihoods because they publish politically undesirable viewpoints. (Right now, those viewpoints are viewpoints that are odious to you, but there's no guarantee that that will be the case in the future; it's easy to imagine Patreon bending to Chinese government pressure to censor people who talk about Falun Dafa persecution or Tiananmen Square, for example, even in countries where that is legal.)

As another example, many Venezuelans are having difficulty fleeing the country, even though fleeing the country is technically legal, due to both official corruption and armed gangs. You can get long-distance bus tickets, but even if the bus can get fuel, it may be robbed in transit by armed gangs ("*piratas del asfalto*", as we call them here in Argentina), who can take your cash and your cellphone but not your LocalBitcoins password or your wallet seed. You could argue that it's not clear what is and isn't legal in Venezuela right now because there are two competing governments, but neither of those governments authorizes demanding bribes from would-be émigrés.

As yet another example, it's perfectly legal in Iran and France for French companies to do business with Iranian companies. But, because much of the world financial system is controlled by the US, it can be difficult in practice; see the Washington Post article on European companies that are selling to Iran for more details. Regardless of whether US law is misguided or not, under well-established principles of international law dating back to the Peace of Westphalia, it certainly does not apply to French companies doing business in France with Iranian companies in Iran.

Also, in many countries, saving money in the local currency is a recipe for poverty, but buying foreign currency is illegal. In 2016, India invalidated most of its circulating currency without warning, so that people would have to save their money in banks instead of in their mattresses; this change significantly exacerbated India's poverty problem. Here in Argentina, the peso lost over half of its value last year. Even the US dollar inflates by a few percent per year, and it's lost 96% of its value since Bretton Woods ended in 1972. Like gold, Bitcoin's volatility makes it suboptimal as a savings vehicle (unless you like to gamble), but for many purposes its disadvantages are less serious than those of the alternatives.

So, in a sense, I think you're right that the main purpose of cryptocurrencies is to sidestep *violent coercion*. I think you're terribly naïve about how much illegal violent coercion is actually "the law", although that's understandable if you've never lived outside a rich country. Increasingly, the same kinds of problems are occurring in rich countries as well, so you may have a chance to correct your naïveté with experience. Hopefully you'll have some Bitcoin first.

There are other uses as well; it's a dramatically better way to send money overseas, for example. Last time someone sent me money via Western Union, I had to go to three different locations, stand in line for twenty minutes, hand over a massive amount of personal information (perfectly suited for identity fraud or targeted home robbery) to an unaccountable third party, sign a false statement, and walk out the door into a dangerous neighborhood with a pocket full of small bills. For this "service", WU charged me/them about 10% of the money sent, and also didn't inform me when it arrived. Bitcoin transactions require none of this nonsense and cost much less; the last Bitcoin payment I received from overseas cost 0.3%, and I received a notification in a few minutes in my Bitcoin client of the transaction.

(See also Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) for how Bitcoin and similar systems could potentially provide many of the financial functions of the current banking system with much lower risk, both individual and systemic.)

Many thanks to sbp for his comments, which greatly improved this note!

Topics

Ternary mergesort

Kragen Javier Sitaker, 2015-09-03 (2 minutes)

Ternary divide-and-conquer algorithms are usually more efficient than their classical binary cousins.

A few years back Java switched to Yaroslavskiy's ternary or dual-pivot quicksort; it turns out that quicksort is marginally more efficient if you partition into three parts in the partitioning stage rather than two, but it's a marginal difference that grows with the dataset size.

Ternary numbers

Similarly, if you represent numbers in N-ary with unary digits, such that 1 is "|," and 0 is just ",", then the optimal value for N is 3, which is marginally better than 2 or 4, which are equivalent. That's because a number M needs ceil(log(M)/log(N)) N-ary digits to represent it, but each of the digits is of size N/2 on average, so the number as a whole is O(N/log N) digits. It turns out that 2/log 2 is equal to 4/log 4, and about 5% larger than 3/log 3; N/log N has its minimum at N = e.

For example, 32808328602 is, represented with unary digits of various bases:

```
2 |,|,|,|,,|,,,,|,|,|,,,,,|,|,|,,|,,|,|,|,,|,|,,,|,|,,|,,
3 |,,,|,,||,,,|,|,,|,||,||,||,,||,||,||,||,|,||,,
4 |,|||,||,||,,|||,||,,|,|||,|,|,|||,|,||,|,||,||,
5 |,,|,||||,|,||||,||,||||,|,|||,,,|||,||||,,||,
6 ||,|||,,||,|||,|||,|,||,||,,||,|,|,,
7 ||,||,||||,|,,|,,|,||,||,|||||,||||,,
8 |||,||||||,||||,|||,||||,|,||||||,|||||,||||||,||||||,|||,||,
9 |,,|||,||||||,|,|||,|||||,||||||||,||,||||||||,|||||||,||||||,
```

And here's 8308233280877:

```
2 |,|,|,|,,,,|,|,|,,,|,|,,|,,,|,,|,,,|,,|,|,,,,|,|,|,,|,,|,|,,|,|,,|,
3 |,,,||,|,,||,,||,|,,,,,,|,||,,,|,,||,||,,|,,|,||,
4 |,|||,||,,|||,||,|,||,||,|,|,,||,|||,,|,|||,|,|,||,|||,|,
5 ||,,||||,||,|,|,,||,|||,,||,,||||,||||,||||,||,,,||,
6 ||,|||||,||||,,,||||,||,|||||,|||||,|||,||,|||||,|||||,|||||,,,|||||,
7 |,|||||,|,|||||,|,|||||,|,|||||,||,|||,||||,|||||,|,|,||,|||||,
8 |,|||||||,,|||||||,|,|||||,|,||,||,||||||,|,||||||,|||||,|||||,|||||,
9 |||,||,|||,||||||,|||||||,,,|,||||||,|,||,||||||,|||,|||||,
```

And here's 9237528032:

```
2 |,,,,|,,,|,|,,|,,,|,|,,,|,,|,|,|,|,|,,|,|,|,|,,,,,,
3 ||,|,||,||,|,|,||,|,,,,|,,,|,,|,||,||,|,||,
4 ||,,||,|,||,||,|,||,|,|,|||,|||,|,|||,||,,,
5 |,||,||,||||,,||||,|||,,|,|||,||||,||||,|,|,||,
6 ||||,|,||,||||,|||,||||,||||,|,||||,|,||,|||||,||,
7 ||||,||||,||||,||||||,||,|||||,||||,|||||,||||||,||||,|,|||||,
8 |,,||||,||||||,||||,||||||,||,|||||||,||||||,|||||||,||||,,
9 ||,|||||,|||||||,|||||,|||,,|||,|,|,||||||||,|||||,
```

You can see that the number size is generally minimum at N=3, but higher bases can have progressively greater variances.

Following this logic, there were a few ternary computers built in the 1950s (?) with three vacuum tubes per three-state "flip-flop" instead of the usual two tubes per two-state flip-flop.

Ternary mergesort

M-ary mergesort of N elements takes ceil(log N / log M) passes over the data; each pass merges M runs at a time, making (in the simple case) M-1 comparisons for each of N elements, for a total of

N (M-1) ceil(log N / log M)

Topics

What does a futuristic OS look like?

Kragen Javier Sitaker, 2017-08-18 (updated 2019-05-05) (6 minutes)

I was just looking at the screenshot of HelenOS at http://www.helenos.org/chrome/site/screenshot.png, and it occurred to me that it's a bit clichéd.

Hobbyist OS clichés that look outdated

I mean, the reasons HelenOS is supposedly interesting aren't capable of being shown in a screenshot at all. But here's a sort of checklist/bingo card of things that look "outdated":

- Fixed-width fonts everywhere.
- Big title bars on all the windows.
- A rotated rectangular window.
- Flat-shaded buttons (and everything) with no texture and no gradient. I mean, Arena had texture on its backgrounds in 1994. All our displays are TrueColor now. No physical objects have no texture and no gradient.
- ASCII-art tables (in the top display).
- A restricted color palette that nevertheless includes a diversity of saturated colors.

There's a lot of hobbyist-OS work that kind of looks like that actually. I'm looking at a TempleOS screenshot that hits #1, #2, #4, #5, and #6; it's mostly fixed-width text on a white background, even the window borders; of the I think 59 lines of text on the screen, about 5 (8%) are taken up by window borders and titlebars; the text is in white, magenta, red, green, dark blue, light blue, and yellow; the filesystem directory and the hotkey menu are both ASCII-art tables. (TempleOS additionally has many aesthetic problems: very wide borders, 640×480×16, color boundaries that butt up right next to text and impair readability, low-contrast color choices, underlining, and a profoundly ugly font, among others.)

ColorForth screenshots I find have fixed-width text in red, green, yellow, white, and blue on a plain black background, hitting #1, #4, #6, and arguably #5.

The Toledo family's Fénix operating system and Biyubi web browser is an effort that seems mostly to have escaped this, though it still suffers from #2.

Oberon suffers from none of these except #4.

Hollywood "futuristic" UI clichés

There's a separate set of clichés in Hollywood depictions of "futuristic" user interfaces, some of which are due to constraints of filming:

- Transparent screens.
- Giant flashing error messages, and more egregiously, non-error messages: "INTERCEPTING SIGNAL", ">>SEARCHING_ALL_LIBRARY_ARCHIVES", "QUERY

COMPLETE".
• Lots of "cool" slide transitions and animations, including lots of three-dimensional rotations and progressive delays. Things rarely just blink into or out of view; instead we have progressive display on even the simplest of data, such as monospace hex dumps or even individual people's names, as if we were receiving it over a 1200-baud modem.
• Bright-on-dark text, but never in monospace except to show that the user is a data vandal.
• Bright-on-dark vector drawings, including lots of wireframes.
• Radial menus. Radial and circular layouts wherever possible, really. This is sort of a callback to "high-tech" products of the 19th and 20th centuries made by lathing steel and brass.
• Touchscreens and voice interfaces.
• Lots of capital letters.
• Holographic displays.
• Textures, animation, and translucency everywhere. Incessant animation.
• Lots of 3-D interfaces and zooming. "This is Unix! I know this!"
• Rectangular grids.
• Great text size variation to focus viewer attention on the most important part, like that used in machine tool DROs.
• Windows with tabs in funny places.
• Random patterns of dots linked into a Delaunay triangulation or something similar, all bright on a dark background, maybe with some of the triangles filled with translucent bright.
• Sound effects, especially square-wave beeps.
• Constant, instant responsivity, even of partial results.
• Lots of primary colors, and things changing color over time.
• Roundrects, perhaps with a bit of rotation.
• Lots of text windows that look more like chyrons than anything else.
• Gestural interfaces.
• Lots of photos, often with strange clipping paths around them, often reminiscent of punched cards.
• Visible pixelation and glitches. Things flickering into or out of visibility.
• Hershey-ish monoline fonts. Old movies often used MCR characters.
• Dotted lines.
• Crosshatch shading (usually light on dark).
• Bright-on-dark displays, especially with cyan glows (the ZnS:Cu color of an analog oscilloscope screen; sometimes the green oscilloscope color is also seen). Fairly sparse color schemes, mostly varying in the intensity of a small number of hue-saturation settings, usually not very saturated. (ZnS:Cu cyan is not very saturated.)
• Bright-on-dark color schemes in general.
• Lots of numbers and graphs.
• Callouts, like on a mechanical drawing — with a leader connecting some text to a point in a drawing.
• Ripples expanding from things.
• Targeting crosshairs.
• Lots of non-convex polygons whose sides are lines at multiples of 45°, often looking kind of like punched cards.
• "Scientific" information: the Earth, the Periodic Table, chemical

diagrams, DNA double helices.

Some of these are actually good ideas. Some are instead chosen to make things look foreboding, scary, "futuristic", intimidating, advanced, and so on; others are simply imitations of previous films. As Christopher Noessel says, "This is the fundamental problem of the idea of sci-fi interfaces, they're not interfaces. What they are are *plot visualizations*. They're there to illustrate, or demonstrate something happening, or something that has happened." Some are not currently achievable. Some are actually packs of prepackaged effects, like the Cybertech HUD Infographic Pack template for After Effects.

A UX StackExchange question goes into more details. There's a whole category of "FUI" or "fictional user interface" videos on YouTube, too.

See also Window systems (p. 1335).

Topics

- Human–computer interaction (p. 3493) (76 notes)
- Independence (p. 3520) (63 notes)
- Graphical user interfaces (p. 3489) (23 notes)
- Operating systems (p. 3608) (18 notes)
- Terminals (p. 3743) (6 notes)
- Oberon (p. 3601) (3 notes)
- Toledo family (p. 3752) (2 notes)

Expanded mineral beads

Kragen Javier Sitaker, 2019-10-01 (12 minutes)

Material made from expanded polystyrene beads (usually referred to as "styrofoam", although that's a brand name for a slightly different material) is very common for a number of reasons.

What if we did this with minerals?

Advantages of expanded polystyrene beads; properties of foams

The foamed nature of the resulting material makes it light, very stiff for its weight, very soft for its volume, forgiving of impact (I think much of the impact energy is actually taken up by compressing the gas), and an excellent insulator, and being some 1–2 orders of magnitude less dense than the source polymer, it is also 1–2 orders of magnitude cheaper per unit volume. It doesn't have much tensile strength because the beads tend to pull apart, but its compressive strength is extreme, although it suffers from creep in both tension and compression.

Moreover, it is much more resistant to fracture propagation than pure polystyrene, which has an alarming tendency to shatter. Both cell walls and bead boundaries act as barriers to crack propagation.

One of its biggest advantages, though, is its moldability. The beads are made to expand inside a mold, in this way filling the mold. (I'm not familiar with the process but I assume it's a matter of releasing the pressure at a temperature where the thermoplastic beads are plastic but not molten.)

Another interesting feature is its amenability to shaping processes. You can cut it with a hot-wire slicer with very little energy and low to very low side loading, producing a smooth and precise surface. The smoothest surfaces with this process come when the wire is touching the foam and transmitting its heat by conduction, rather than by radiation, but this produces some side loading and consequent surface imprecision. The "swarf" from this process is absorbed into the surface of the material by virtue of densification from collapsing cells. It's also fairly easy to cut with more traditional processes like sawing and milling — much, much easier than the underlying polymer. These processes work at such low forces that they do not create significant heat, and thus do not create a heat-affected zone.

The extension at yield of the foam is somewhat larger than that of the bulk polystyrene material, which is relatively brittle, but not dramatically so.

In terms of material properties, probably the most extreme result of the foaming process is the decrease in the material's Poisson ratio: the foamed material very nearly does not expand at all laterally when compressed, even plastically, or shrink laterally when extended. This is particularly favorable to *forming* processes: drawing on a styrofoam cup with a thumbnail, for example.

Why isn't everything made from styrofoam? It's very flexible for its volume, which is an advantage in many applications but a disadvantage where it needs to resist buckling — though it is

commonly used as part of composites (see Sandwich theory (p. 2450), which also talks about buckling a lot). It isn't transparent, it doesn't resist high temperatures (because polystyrene creeps and then melts), and it isn't hard. The molding process isn't very precise, typically having surface roughness that approaches a millimeter in places, and I suspect it's considerably slower than injection molding. It's nonporous, and in a lot of situations where softness is desirable, porosity is too.

Finally, a lot of people just fucking hate it aesthetically, precisely because it's so widely used and cheap; also, it got a bad rap in the 1980s when it was blown with CFCs and thus contributed to ozone depletion.

Presumably any desired properties of a random foam could be equaled or dramatically exceeded by a metamaterial with similar pore size but precisely controlled geometry, topology, and heterogeneous materials, in the same way that woven and knit fabrics exceed felt and nonwovens, and masonry arches and reinforced concrete skyscrapers can exceed random piles of rocks. So in a sense foams may be a bootstrapping step toward self-replicating machinery or a technique to use at scales too small for your available machinery to manipulate explicitly, not a long-term material. Living tissues generally do not contain unstructured, random foams at scales larger than a white blood cell, and once the humans' fabrication technology is not so primitive, neither will their fabricated artifacts.

Why minerals?

As I've discussed in some other notes, it's very desirable for self-replicating machinery to be independent of organic feedstocks, including things like petroleum: inorganic material is much more abundant both on Earth elsewhere, it doesn't create conflicts with objectives like reterraforming, and it avoids scaring the humans. However, most materials that are strong but not too brittle at or near room temperature are organic. It might be possible to improve this situation with metamaterials — glass fiber, basalt fiber, and steel coil springs are far from the limits of what can be done. One of the simplest "metamaterials" to fabricate is foam.

Candidate minerals

In addition to styrofoam-like processes, it's worth thinking about infusion postprocessing, where after the foam is shaped its cells are filled with some other material. This probably requires the cells to be substantially open rather than closed. Also, think about painting (depositing a layer of a different material on the surface, such as a harder material, maybe a metal), use as a mold (for example, for metal), and defoaming the surface to form a harder surface layer.

Soda-lime glass

Soda-lime glass commonly requires extreme measures to prevent it from foaming: ingredients to reduce its viscosity and long kiln times to allow bubbles to escape. (I'm not sure if precalcination of some of the ingredients is also used). Bubbles are one of the most common obstacles to getting good, clear glass from raw materials.

If you want *more* bubbles instead of none, it seems like it should be easy to achieve: leave out the viscosity-reduction agents; incorporate

more gas-producing minerals such as boric acid, sodium carbonate and even bicarbonate, and calcium carbonate, and cool the melt rapidly.

Waterglass

The sodium silicate I have here dries to a glassy, mildly alkaline substance with a Mohs hardness around 3–4. Upon heating to around 200–300°, it foams up into a white foam with visible bubbles; presumably water inside of it is boiling out and the heat has plasticized the sodium silicate enough to permit bubble formation. The total volumetric expansion is maybe a factor of 10. (These are crude stovetop experiments; my temperature estimate is based on the fact that a grease spot on the same electric burner had started smoking shortly before, for example.) The resulting foamed material floats and leaves tiny silica-gel sparkles when rolled across my palm.

(I suspect that if I let the dried waterglass sit for a long enough time, it might become harder by absorbing CO_2 from the air.)

This phenomenon is behind what I hear is the common use of sodium or potassium silicate as a firestop — upon heating, it foams up to produce a non-oxidizable insulating layer that stops airflow and heat transport.

This suggests that it should be possible to use dried sodium silicate beads or perhaps silica-gel beads to mold and foam up in the same way as styrofoam.

Since the water doesn't condense again until a lower temperature at which the waterglass is no longer plastic, the foam remains foamed after cooling.

Perlite and vermiculite

These are commonly foamed minerals used in gardening and glassblowing, able to withstand much higher temperatures without softening than the materials mentioned above; vermiculite is used as an insulator in laboratory glassblowing of borosilicate glass, commonly in direct contact with the hot glass without sticking to it.

They are mostly open-cell foams, which is the reason for their use in gardening to improve soil drainage.

In both cases, the foaming action is produced by the escape of water vapor from the mineral under heating; the minerals are found in nature in their unfoamed form, having formed, I think, under pressure sufficient to prevent the water from escaping.

Pumice

This rock is naturally foamed during volcanic eruptions; like obsidian, it is a glass, but it contained a large amount of dissolved gas which bubbled out of solution during the eruption. Typically the foam is sufficiently closed to allow the rock to float.

Pumice is not common, suggesting that unusual circumstances are needed to produce it; I don't know if that's a matter of mineral content, dissolved-gas content (which might depend on the mineral content), or rapid cooling (which might not be entirely adiabatic; obsidian typically depends on water cooling). More common natural volcanic mineral foams such as scoria cool much more slowly and typically have an open-cell macrostructure and a crystalline rather than glassy amorphous microstructure.

Concrete

Of course foamed portland concrete is a thing. I ran into a pallet of stacked foamed-concrete blocks on the sidewalk the other day. This works by mixing detergent and air into the water before mixing the concrete. While the reduction in weight and consequent improvement in insulation properties is substantial, as I understand it, it's rarely even a factor of 3, let alone the 10–100 of styrofoam.

You could conceivably puff up foamed concrete into a mold before the cement starts to set, either by initially mixing it under pressure and then releasing the pressure, or by mixing it at one atmosphere and then pulling a vacuum once it's in the mold. But usually people just pour it into molds before it sets without fooling around with pressures.

Firebrick

Insulating refractory firebrick is also a common foamed mineral product. Most commonly it is made by mixing the refractory clay with a filler material which burns out during firing; any carbon or organic matter will do (I made some using used yerba mate; see file `ceramics-notes`), and I think sulfur would do as well. This material is also amenable to molding in its plastic state.

I was able to get density reduction of up to a factor of 9, but beyond a factor of 4 (3:1 clay body to yerba) the material was noticeably friable. At a factor of 4, it felt solid but could be carved with a thumbnail in fired form. These foamed ceramics were enormously easier to cut than regular fired clay, which tends to shatter whenever you try to cut it, and enormously more resistant to thermal shock, which I suspect is partly because it's softer and partly because of its insulating qualities.

To make this process amenable to execution without organic matter, you'd have to find a different filler. This could be something with a low boiling point (like zinc); something whose oxides have a low boiling point, as do carbon, hydrogen, nitrogen, and sulfur; or something that decomposes or substantially decomposes into gases at high temperatures, like nitrate or carbonate.

Alternatively, you might be able to use a styrofoam-like foaming process, where a gas such as nitrogen or CO_2 is dissolved into the clay's water under high pressure, then bubbled out with a release of pressure.

Topics

Critical defense mass

Kragen Javier Sitaker, 2013-05-17 (14 minutes)

I think our only real hope for survival and prosperity is learning to cooperate nonviolently — not just without violence and threats in day-to-day life, but without even condoning state violence such as war, arrests, and imprisonment. Historically, state violence has been far from sufficient to organize society; society's functioning rested primarily on nonviolent day-to-day interactions, what Gandhi called satyagraha, primarily between people who know each other in small communities.

Mostly starting in the 20th century, although with antecedents in historical episodes like the Roman Dominate, we've seen a dangerous set of experiments in organizing society primarily around the state, which is to say around violence. While these experiments have produced some promising results (the extermination of polio, space travel, dramatically improved agricultural yields, the internet) I hope we can move beyond them to a peer-to-peer global society. Otherwise, it seems that we are doomed either to live under a global state — which we can hope against hope will not be particularly despotic — or perish in a nuclear holocaust or similar tragedy.

Violence, Warfare, and Agriculture

Nevertheless, it remains true that through most of history, we've been under the threat of violence from other communities. Warfare and warriors have been a constant plague on humanity since prehistory, and historically, we have had no hope of living without violence, only of prevailing in it.

And, historically, this violence has been mostly two-dimensional: warfare and warriors moved around on the ground. This gives rise to a minimal stable size for an agricultural settlement, because the circumference of a two-dimensional shape increases more slowly than its area as the shape grows.

A farmer cannot defend his own field from looters, armed or otherwise. It might require 100m² or 1000m² of land to feed one farmer, depending on the climate, your cultivars, your ability to trade, and so on; but even 100m² of land has a minimal circumference of some 35 meters. One farmer might be able to defend one or two or ten meters of circumference, even while he's awake, but not 35.

But a circle with a circumference of 70 meters has an area of 400m², enough to feed, hypothetically, four farmers. That leaves each of them only needing to defend 18 meters of circumference instead of 35. Still too much, but perhaps moving in the right direction.

If each farmer can defend ten meters, your minimal size is almost twice that, or 130 meters of circumference, and 13 farmers, with their wagons or houses circled in a 20-meter circle around 1300m² of land.

If each farmer can defend two meters of circumference, you have 315 farmers on a circle 200 meters across with a total area of 31500 square meters, about three city blocks, three hectares, or eight acres. This is beyond Dunbar's Number, and so at this point you start needing institutions, formal hierarchy, and so on. Two meters is small enough that you can have night sentries who wake everybody else up

if they see or hear anything, and dense enough that you can't steal the farmers' crops by yourself — you need a raid by an armed and organized band of bandits.

100m² is a really small farm. It's not enough to support a person except in the most fertile parts of the world, with a lot of luck. 1500m² might be a more realistic estimate in most of the world. If you need 1500m² per person, but each person can defend only five meters, your minimum community size is 754 farmers, 600 meters across, with an area of a million square meters (100 hectares).

Of course, in real life, the farmers don't live on the outer borders of the farmland; they only go there when there's a raid or the threat of a raid from another tribe, and they keep what is most precious to them — their lives and those of their families — in a much smaller area that's easier to defend than their entire fields. But there still need to be enough of them to keep a watch on the border, and to repel the raiders when they raid.

The Origin of Stratification

I think this is why the birth of agriculture led to the stratification of society. It's not, as many have said, that agriculture makes it possible, for the first time, to produce enough surplus to support a richer class; observations of contemporary hunter-gatherers show that, even though they're living on the most marginal lands, they still only work for survival a small amount of the time. And it's not just that agriculture makes you store up your harvest in a granary where it can be stolen. It's that keeping hunter-gatherers from gathering what's in your field requires you to organize into groups that are bigger than Dunbar's Number, and the more organized the hunter-gatherers or pastoralists or invading agriculturalists are, the bigger your community needs to be to repel them — even at the cost of enabling parasitic warrior-kings.

To put it more plainly, in such a world, if your farmers aren't willing to die for the sake of people they don't really know, if they aren't willing to sacrifice their lives for an abstraction, if they aren't willing to be sent to their deaths by a general, then raiders will take their food and their children will starve. But the general or king who chooses who to send to their death will not send his own sons first, and if you do not pay him tribute, he can send his loyal subjects to take it from you by force. So stratification is born.

If this explanation holds water, we'd expect to see:

- Proportionally smaller states and less social stratification in areas and time periods with higher natural agricultural productivity.
- Larger states and more stratification in areas with more forceful raiders.

I think these do exist, although I'm not sure — many enormous empires with great stratification have surely existed in very fertile places.

(The explicatory power of this area-to-perimeter ratio thing is perhaps somewhat dubious: surely in most of history the greatest determinant of the size of any given state has been the size of neighboring states, no?)

What about new technology?

Vat food

People only need to eat 100–120W of food, or 2000–2500kcal/day, and the solar resource in most of the world is greater than that per square meter — in the US, it's mostly 3–6kWh/day/m², which is 120–250W/m². The reason you need hundreds or thousands of square meters per person for agriculture is that, first, natural photosynthesis is only about 3% efficient even when not resource-limited; second, plants spend most of their energy on growth and reproduction, not on feeding you; and, of course, you need micronutrients.

But what if you had a 60%-efficient electric solar collector, powering a 30%-efficient synthetic sugar-and-protein plant? Then you could turn 18% of the sunlight you caught into edible calories. If you had access to a 250W/m² solar resource, you could get by on 2⅔m². Remember how, before, the need for one defending farmer per five meters of perimeter forced the farmers to band together into groups of 754? Now you can get that same level of security with 2 "farmers" guarding a 1.3-meter-diameter circle. You don't hit Dunbar's Number until you have 135 "farmers" guarding their shared 10.7-meter-across solar plant, half a meter apart.

One person per 2⅔m² is about 370 000 people per square kilometer, which is more than ten times the density of the densest cities in the world, such as Delhi, or Manhattan, or Friendship Village, Maryland.

That is, people eating synthetic macronutrients out of vats could establish population densities that exceed the population densities of our current agriculture-based society by as huge a ratio as ours exceeded the density of the pastoralist and hunter-gatherer societies it has defeated. And, perhaps, they would have no need to establish hierarchical states with unaccountable rulers, simply in order to be able to protect their crops from raids; they could do it with tribes of hunter-gatherer scale.

None of this is quite feasible yet, neither 50%-efficient solar-energy collection (the current state of the art is about 40%, or 5% at the lowest cost per watt) nor efficient macronutrient synthesis. However, they're both clearly technically achievable.

(You might think that it would be pretty uncomfortable to have only 2⅔ m², 29 square feet, per person, like hanging out in a crowded supermarket for your entire life; but you could build buildings many stories tall, to provide each person with ample living space.)

(Another caveat: as Charlie Manson, David Koresh, and the like have amply demonstrated, living in a hunter-gatherer-sized autonomous band is no guarantee of living in peace or hunter-gatherer-like egalitarianism.)

The United States contains 9.8 million km². If it were populated at 370k people/km², it would contain 3.6 trillion people, 500 times the current population of the Earth. At a twentieth of that density, it would contain 180 billion people, all living with a material standard of living comparable to current US society. The corresponding numbers for Argentina are 2.8 million km², 1.04 trillion people, and 21 billion people. For Earth, 150 million km², 56 trillion people, and 2.8 trillion people, about 7900 or 400 times the current population.

At the current population growth rate of 1.1% per year, we'd reach those population benchmarks in the years 2561 and 2835; but if the

world population were to grow at Qatar's 4.9%, which we know is possible, we'd reach them in years 2138 and 2201. In-vitro gestation would make much higher population growth rates possible.

(Why a twentieth? Because it brings the total solar power available to each person to some 5–10kW, or 2½-5kW after conversion to electricity, putting them on par with modern US consumption of some 10.4kW per person.)

Given the extent to which modern cities are tolerant of pluralism, and the compatibility of growing vat food with city life, it's probably not realistic to imagine the eruption of such a new lifestyle in separated communities; rather, people inside of cities will buy, separately, solar energy harvesting devices and macronutrient-growing vats, probably in both cases mostly as emergency fallbacks; and vat food won't become popular until a generation comes to maturity that grew up eating it, probably due to a political and economic catastrophe that forced them to grow up in poverty, like Spam and horrible boiled vegetables in England after World War II.

Furthermore, vat-growing food won't become a popular thing to do as a hobby, like bean sprouts or yogurt-making, unless it becomes popular first as an industrial-scale product, or unless there are regulatory and political reasons people can't do it on an industrial scale.

Quite aside from the purely factual questions considered above — how vat food could become established, what population it could support, where it would likely take root, and so on — there is the normative question to consider: is a vat-food future dystopic?

Hierarchical or not, it sounds dystopic to me.

Robots

Instead of increasing agricultural productivity per square meter, consider the possibility of extending each farmer's defensive powers. Today drone pilots in the US armed forces routinely control four drones at once, each capable of surveillance and unaccountable targeted violence over a very large geographical area.

If each farmer has at his command some kind of robots, perhaps he could use the robots, instead of his neighbors, to guard his garden against raids. In a sense, this is what happened with the enclosure of the Old West: instead of robots, ranchers installed barbed-wire fences ("bob wahr") to keep their cattle from wandering off, converting their rustler-killing, Native-American-battling pastoralism into a kind of agriculturalism. It didn't stop rustlers, but it slowed them down enough, and it did stop the buffalo.

Now, it seems eminently plausible that one person with a bunch of video cameras and, say, remotely triggered Tasers, chains, salt-water squirters, mines, and so on, could police 1500m² of land, which after all has a perimeter of only 138 meters, against raids. His neighbor, or his neighbor's robots, wouldn't be able to step onto his land to dig his potatoes without his permission.

But could one person really put all of that in place? Maybe not — but it seems clear that one person with modern surveillance, computation, and weapons could defend a larger perimeter than one person without.

So far, though, we've seen a countervailing trend: the increasing

division of labor, and thus specialization, necessary to produce the computing devices is deeply bound up with the current world system of enormous states with millions to hundreds of millions of citizens or subjects. And while current machinery may not make it particularly easier to steal potatoes, it certainly makes it a lot easier to cut chains and blow up mines, enabling attackers as well as defenders — especially attackers using the power of large states with hundreds of millions of members.

Topics

Minimal imperative language

Kragen Javier Sitaker, 2018-12-10 (7 minutes)

What's the smallest we can make an imperative programming language to, for example, plot points in a framebuffer?

Well, BF is one answer to that question; StoneKnifeForth is another. But what about a language that supports subroutines (recursive, with arguments, but without closures), conditionals, loops, arrays, and arithmetic with infix syntax? Because I guess I'm not willing to go *that* minimal.

A thing you have to think about is whether arrays are valid as arguments or not. That makes a big difference in the flavor of the language.

You need some way to declare arrays, but that could be static, like subroutines are in C.

Your syntax might look like this:

```
program: _ (name '['_ int ']'_ | name '('_ (name, ','_)? ')'_ ':'_ exp)*
exp: stmt, ';'_ | '{'_ (exp '->'_)? exp? '}'_ | stmt
stmt: name ('=' | '<-' | [-+*/%^&|] '=' | '&^=')_ stmt | cond
cond: cmp '?'_ cond ':'_ cond | (cmp, '&&'_), '||'_
cmp: val ('==' | '<=' | '>=' | '<' | '>' | '!=')_ val | val
val: ((((chain, [*/%]_), [-+]_), ('<<' | '>>')_), ('&' | '&^')_), [|^]_
chain: ([-+~]_ chain | atom) ('('_ (expr, ','_)? ')'_ | '['_ exp ']'_)*
atom: '('_ exp ')'_ | name | int
_: [ \n\t]*
int: [0-9]+
name: [A-Za-z_] [A-Za-z0-9_]*
```

In this grammar, the syntax `a, b` means `a (b a)*`; , binds more tightly than |, so `a | b, c` means `a | (b, c)`, and `a, b | c` means `(a, b) | c`. This enables this grammar to get by without defining associativity much, though it does define precedence. It also is free of left recursion, enabling a straightforward PEG implementation.

Most of this is the same as C or Golang, but the `{ foo -> bar }` construct is intended to mean `while (foo) { bar }`, and the distinction between = and <- is that = declares and initializes a new variable, while <- mutates an existing variable. (Inconsistently, += and the like are not written +<-.) The intended semantics are that everything has a value, including `stmt`, but loops return just the zero value of their conditional upon exit, rather than anything useful like their last body expression or a list of their last body expressions (since we don't have lists). Sequences `a; b` likewise return the value of the last item in the sequence.

There's a bit of parsing confusion where a stray : after a function call could give you a potentially misleading error message.

So here's a program:

```
f[100]
fib(): f[0] <- f[1] <- 1; i = 2; {i < 100 -> f[i] <- f[i-1] + f[i-2]; i += 1}
```

The really lame nature of not being able to initialize data structures shows up strongly in this program.

Here's another.

```
minskytron(x, p, n): y = 0; {n -= 1 -> x += y >> p; y -= x >> p; pset(x, y)}
```

Here's a toupper function operating on ASCII codes in s.

```
s[4096]
toupper(i, end):
    {i < end ->
        (s[i] >= 97 && s[i] < 97 + 26) ? s[i] -= 64 : 0;
        i += 1}
```

This language is somewhat similar in its capabilities to BASIC or bc, though it lacks strings.

It is, however, considerably bulkier in the description of its syntax than the λ-calculus, Abadí and Cardelli's ς-calculus, or the ur-Lisp. On the other hand, an implementation of an interpreter for it might be simpler, since you don't need any memory management or type testing. (You might need subscript error handling.)

PEG syntax

(See also Tagging parsers (p. 208).)

It's perhaps worthwhile dwelling a bit on the syntax of the PEG above. It doesn't use negation, but I'm including negation here, since it's an important tool in PEGs in general.

```
grammar: (name ':'_ alts)*
alts: (seq | seq ','_ seq), '|'_
seq: ('!'* (name _ | str | class | '('_ alts ')'_) [?*+]*)*
str: "'" ('\\' char | [^\\'])* "'" _
   | '"' ('\\' char |  [^\\"])* '"' _
class: ('[^]' | '[]' | '[^') [^]]* ']'_
_: [ \n\t]*
name: [A-Za-z_] [A-Za-z0-9_]*
```

The definition of character classes omits the syntax of ranges, but that's okay as long as we don't care about the rightmost member of a range being].

A big problem with this syntax is that it doesn't provide a way to tag parts of a production so they can be referred to elsewhere. Following the proposal in Tagging parsers (p. 208), let's use the syntax `name { contents }` to tag the range of input matched by `contents` with tag `name`. To achieve this, we could just change the definition of `seq` in the above as follows:

```
seq: ('!'* (name _ | str | class | '('_ alts ')'_ | name _ '{'_ alts '}'_) [?*+]*o
o)*
```

Now we can take advantage of this to build an AST, refactoring the grammar a bit in the process:

```
grammar: _ rule {name ':'_ choice}*
choice: choice {alt {term* | item {term*} ','_ sep {term*}}, '|'_}
term: '!' negated {term} | modded { atom mods { [?*+]+ } } | atom
```

```
atom: name _ | str | class | '('_ choice ')'_ | tagged
tagged: tagged {tag {name} _ '{' _ spans {choice} '}'} _
str: "'" str {('\\' char | [^\\'])*} "'" _
   | '"' str {('\\' char | [^\\"])*} '"' _
class: '[' class {'^'? ']'? [^]]*} ']'_
_: [ \n\t]*
name: name {[A-Za-z_] [A-Za-z0-9_]*}
```

Separating nonterminals from tags allows us to avoid constructing worthless intermediate nodes in some cases; the `term` rule can generate, for example, just a `str` node or just a `class` node, rather than a `term` containing an `atom` containing a `str`. It also enables the resulting node to tag just the relevant text, omitting irrelevant delimiters.

The idea is that each AST node has a start byte position, an end byte position, and a sequence of zero or more child nodes. In token-like cases, the client program is probably more interested in the byte positions, while in other cases, it probably only cares about the child nodes. So, for example, a `choice` node in the AST will have zero or more `alt` children, none of which children include the `|` separators between the alternatives. The `alt` nodes may have a single `item` child and a single `sep` child, or they may have a sequence of the possibilities that come from `term`: `negated`, `modded`, `name`, `str`, `class`, `choice`, or `tagged`.

The `modded` node structure is an unfortunate result of PEGs' lack of left-recursion; ideally the AST for something like `x*+?` would be `optional { oneormore { zeroormore { name "x" } } }`, although of course that is a pretty stupid thing to write. However, once we've parsed the thing into a lopsided tree structure, it's pretty easy to write imperative code in your language of choice to produce the desired structure. See Tagging parsers (p. 208) for another solution to this problem.

Topics

Audio tablet

Kragen Javier Sitaker, 2019-09-28 (7 minutes)

Today I was talking with David Christensen about a project of his, and I had some ideas about tracking styluses on drawing tablets using ultrasound. In his project, which is not a drawing tablet, they're tracking a point of contact on a surface using an array of piezoelectric contact microphones on the back of the surface, using the relative intensity of the conducted sound at different microphones to estimate the location.

It occurred to me that by cross-correlating the signals at the different microphones, you can do a much better job of localizing the sound, and this could be useful for an inexpensive large-area drawing tablet. (This is related to Measuring submicron displacements by pitch bending a slide guitar (p. 905).)

A stylus scratching on a rough surface such as paper or MDF produces broad-spectrum noise, and broad-spectrum noise is wonderful at having very low autocorrelation at any shift other than zero; it's very nearly orthogonal to itself at other shifts.

Echoes from the edges of the tablet can set up Chladni-plate-like standing waves, which could complicate the situation substantially (like some stupid Hollywood action movie that ends in a hall of mirrors) so using a highly attenuating material like leather might be a good idea, or perhaps cutting the edges of the material in a sunburst-like zigzag pattern so that the high frequencies we want are strongly attenuated and their coherence destroyed as they reflect from the edge. (This is related to the Q of acoustic resonators such as music-box tines cut from the material, although I don't know if we can talk about an acoustic Q of the material itself; but clearly for this purpose MDF is dramatically superior to plastics, which are dramatically superior to metals, which are mostly somewhat superior to ceramics.)

With only two microphones you would have an ambiguity about which side of the line through them the stylus is on (whose importance could be minimized by putting them along the same edge of the tablet); three microphones would avoid this problem, and more than three microphones would help to reduce errors and latency further. Latency of under 10 milliseconds is critical for musical use and strongly desirable for drawing; anything over 1 millisecond is detectable and undesirable.

The localization precision and interaction latency are both limited by the speed of sound in the material, but unfortunately in opposite directions: a higher speed of sound means less interaction latency but lower precision. Using higher frequencies alleviates this problem. Suppose you have three microphones in an equilateral triangle one meter on a side; the center is 661 mm from the corners, and that's as far as you can get from the corners inside the triangle or indeed anywhere near it. With a speed of sound of 2000 m/s, a reasonable estimate for many solids, that works out to an intrinsic acoustic latency of 331 microseconds, not counting processing time. If there are significant 10-kHz components of the noise being tracked, they will narrow the autocorrelation peak to around 100 microseconds ---

but at 2km/s, that's 200 mm of position uncertainty! That's no good for drawing, which needs submillimeter precision. A lower speed of sound would reduce the positional uncertainty proportionally.

However, correlation and intensity aren't the only sources of information we have. Solids actually carry two different kinds of sound, longitudinal and transverse, and transverse waves are slower and have polarization. If the microphones are able to detect the direction of vibration, for example by coupling them to points on the board through taut UHMWPE or glass-fiber threads near tangent to the board, they will first detect the longitudinal waves moving the board towards and away from the point of contact, then later the transverse waves moving it in some direction normal to the vector towards the pencil.

This still depends on getting substantial phase separation of the two waves --- I haven't measured yet but I think they'll tend to be strongly correlated, though perhaps longitudinal impulses going in one direction will be strongly associated with transverse impulses propagating at right angles to it.

Raising the frequency would help a lot, but you need to raise it by a factor of 500 or so, to about 5 MHz. It may be the case that pencils scraping on paper or MDF intrinsically produce 5-MHz noise, but I doubt it. 5-MHz ultrasound doesn't travel very far in air, but it has no difficulty with most solids and liquids. You could attach a small sound transmitter to the pencil that transmits a 10Mbps LFSR signal, which would be transmitted to the board whenever the pencil was touching it. (Or touching paper taped to it.) You could transmit this signal intermittently --- a 40-bit burst, taking 4 microseconds, every 100 microseconds or more, would be adequate.

Alternatively you could couple ultrasonic vibrations into the board from a piezoelectric, magnetostrictive, or electromagnetic actuator mounted on the back of it and see how they scatter; this might be adequate but would probably work better for large, hard contact points than for a pencil point. Or, as described for the one-dimensional case in Measuring submicron displacements by pitch bending a slide guitar (p. 905), you could attach the detector to the stylus (or the person's finger) and pick up vibrations injected into the surface.

Periodicity in the sound injected would be problematic, since the autocorrelation of periodic waveforms has many peaks, creating ambiguity about the stylus position. It's easy enough to avoid with an LFSR in the electronic case, but for acoustically produced sounds there is the risk of resonances.

Vibration transducers attached to the board could also, at sub-kilohertz frequencies, provide haptic feedback like the piezoelectric click pioneered by Nokia for some of their cellphones years ago; and if the tablet is horizontal and isn't very well damped at the edges, they could also make Chladni figures to move small objects around on it, also a technique demonstrated some years ago by a research group using a single actuator to vibrate a metal plate.

Topics

Division

Kragen Javier Sitaker, 2014-06-05 (14 minutes)

(This was published previously on kragen-tol.)

My friend Santi asked me why we divide by a fraction by interchanging the numerator and the denominator and multiplying; that is, why a/(b/c) = a(c/b). I wasn't quite sure how to answer, but after thinking about it, it turns out that there are many deep and fascinating answers that involve many aspects of the universe of mathematics. Here are three different answers. Sort of.

Part of the problem is that it's difficult to say what really counts as an explanation here, because, as Feynman explained in his famous BBC video on "Fucking magnets, how do they work?", an explanation has to start with things that you already understand to be true. In cases like this, it's really easy to fool yourself into thinking that you have an explanation, when all you really have is circular logic. Here, let me demonstrate.

An answer based on group theory

A "group" is a set with an operation that have the four properties of closure, associativity, identity, and invertibility. Nonzero fractions, together with multiplication, are a group. It turns out that the divide-by-multiplying-upside-down thing isn't limited to fractions at all; it's a much more general property that applies to any group, including bizarre things like permutations under composition, three-dimensional rotations of polyhedra under composition, matrices under matrix multiplication, Gaussian integers under addition, bit strings of some fixed length under XOR, and integers under multiplication modulo a prime number!

To explain, first I will explain the meaning of the group properties. Since I'm writing this mostly in ASCII, I'm going to use "G" to mean the set, other letters to mean elements of the set, "+" to mean the operation, and two other special notations which I explain below: "0" to mean the identity element of G, and "-X" to mean the inverse of an element X of G.

Closure: if A and B are in G, then A+B is also in G. This rules out things like "numbers 1 to 10 under ordinary addition", because 9+9 is 18, which isn't in the set, and things like "integers under ordinary division", since even though 6/3 is in the set, 2/3 isn't.

Associativity: (A+B)+C = A+(B+C), always. This rules out non-associative operations like division or subtraction.

Identity: There's an element of G which I will call 0 which has the property that A+0 = 0+A = A, for all A. We call it the "identity element". This rules out operations like "return the left argument".

Invertibility: Every element A in G has a corresponding inverse element -A such that A+-A = -A+A = 0. This rules out operations like multiplication on rational numbers, since zero (not the identity element, real zero) is a rational number, the identity element for multiplication on rational numbers is 1, and there's nothing you can multiply by zero to get 1. (But multiplication on nonzero rational numbers is still fine!)

You will note that commutativity isn't one of the group properties,

even though, say, integer addition is additive; and in fact there are lots of interesting groups whose operation isn't commutative. If you can prove a property of groups without depending on commutativity, then it applies not just to commutative groups like the nonzero rationals under multiplication, but also noncommutative groups like matrices under matrix multiplication and permutations under composition.

The question we started out with was, "Why does a/(b/c) = a(c/b), when a is a rational number and b and c are nonzero integers?" Here's why that's true not just for integers but actually for any nonzero rational numbers.

Our question, restated in generic group terms

Let's start by rewriting it in the notation I have above: ab becomes A+B, and a/b becomes A+-B. So we're trying to find out if, and why, A+-(B+-C) = A+(C+-B).

Using the definition of invertibility (-), we can derive that:

```
B+-B = 0
```

From there, we can replace B with B+o, using the definition of identity (o):

```
(B+0)+-B = 0
```

Then, using the definition of invertibility, we can replace that o with -C+C:

```
(B+(-C+C))+-B = 0
```

The definition of associativity lets us move around the parentheses around + operations:

```
(B+-C)+(C+-B) = 0
```

Now, if (B+-C)+(C+-B) is o, then for any element Q, by the substitutability property of equality:

```
Q+0 = Q+(B+-C)+(C+-B)
```

In particular, if we take Q to be -(B+-C), which we know exists by the properties of invertibility and closure, we have:

```
-(B+-C)+0 = -(B+-C)+(B+-C)+(C+-B)
```

The right side now begins with a thing of the pattern -R+R, which we know from invertibility is o, so we have:

```
-(B+-C)+0 = 0+(C+-B)
```

And by the definition of identity, we now have:

```
-(B+-C) = (C+-B)
```

This is a stronger form of what we wanted to prove in the first place, since it shows that for any A:

```
A+-(B+-C) = A+(C+-B)
```

which is our original statement, but it also shows that you don't even need the A.

Applying the result to other groups

So that shows that a/(b/c) = a(c/b), as long as all three are nonzero rational numbers, and in particular if b and c are nonzero integers. It also shows that the same thing is true if, for example, a, b, and c are numbers in the range of 1 to 6, with the following tables for multiplication and division in Z/7:

```
 *  1  2  3  4  5  6     /  1  2  3  4  5  6
 1  1  2  3  4  5  6     1  1  2  3  4  5  6
 2  2  4  6  1  3  5     2  4  1  5  2  6  3
 3  3  6  2  5  1  4     3  5  3  1  6  4  2
 4  4  1  5  2  6  3     4  2  4  6  1  3  5
 5  5  3  1  6  4  2     5  3  6  2  5  1  4
 6  6  5  4  3  2  1     6  6  5  4  3  2  1
```

For example, if a=3, b=4, and c=5, then a/(b/c = 5) = 2, and a(c/b = 3) = 2 also. Try it for any three of these numbers. It will always work. (It works in Z/p for any prime p. See attached modmul.py.)

Here's a Python example of permutations using my permutations module:

```
>>> from permutations import cycle
>>> a = cycle(1, 2, 4)
>>> b = cycle(2, 5)
>>> c = cycle(2, 4)
>>> a * (b * c**-1)**-1
cycle(1, 2, 5)
>>> a * (c * b**-1)
cycle(1, 2, 5)
```

Note that, since permutation composition is not commutative, a(1/b * c) is not the same:

```
>>> a * (b**-1 * c)
cycle(1, 2) * cycle(4, 5)
```

Why this is a little bit bogus

But wait! All this is based on my initial assertion that the four group axioms apply to nonzero rationals under multiplication. How do we know that's true? Maybe all of the above is begging the question, when it comes to the nonzero rationals, since how do we know that nonzero rationals even have multiplicative inverses in the first place? I mean, if we assume that rationals including zero are a group under multiplication, then we can use the same argument to claim that rationals including zero can be divided in this way, but that turns out to be wrong. So first we have to show that nonzero rationals are a group!

Rational numbers as pairs of integers

Suppose we take as given that nonzero integers form a *commutative* group under multiplication, and we want to explore relations a:b of those integers, using the equality relation a:b = c:d iff an:bn = cm:dm for some nonzero integers m and n; and we define multiplication as (a:b)(c:d) = ac:bd. What can we find out?

First, we can prove pretty directly that ((a:b)(c:d))(d:c) = a:b, as long as neither c nor d is zero:

```
((a:b)(c:d))(d:c) =
    {definition of multiplication}
(ac:bd)(d:c) =
    {definition of multiplication}
acd:bdc =
    {commutativity of integer multiplication}
adc:bdc =
    {associativity of integer multiplication}
a(dc):b(dc) =
    {definition of rational equality}
a:b
```

(Bloviation 0)

Which is pretty close to what we started out wanting to know. But we can also show that nonzero rational numbers, defined in this way, are a group under multiplication. We need to show closure, associativity, identity, and invertibility.

Closure: (a:b)(c:d) produces ac:bd, and since the integers are closed under multiplication, ac and bd are integers.

Associativity:

```
((a:b)(c:d))(e:f) = (ac:bd)(e:f) = (ac)e:(bd)f
(a:b)((c:d)(e:f)) = (a:b)(ce:df) = a(ce):b(df)
```

which are equivalent because integer multiplication is associative.

Identity: (1:1)(a:b) = 1a:1b = a:b since 1 is an identity for integer multiplication.

Invertibility: Bloviation 0 above shows that these relations have right multiplicative inverses (and how to compute them); you can carry out the same argument for left multiplicative inverses.

So that shows us how to compute multiplicative inverses of pairs of nonzero integers with these weird definitions of multiplication and equality. But how do we know that those pairs of integers are *really* "the rational numbers"?

We can translate individual integers into this pair-of-integers form as follows: t(x) = x:1. It should be straightforward to see that multiplication on these pairs corresponds to multiplication on the integers, i.e. t(x)t(y) = t(xy). And, in math, that's really all you need: it walks like the group of fractions, quacks like the group of fractions, so it's the group of fractions! There is no more "really" than that, in math.

I guess this is the algebraic structure you get if you assume that nonzero integers must have multiplicative inverses, and then take the set of the integers and their multiplicative inverses and extend it by transitive closure of multiplication; it's the smallest set that includes

the nonzero integers and satisfies the group axioms. I'm not immediately sure how to show that, but I'm pretty sure it's true.

An interesting thing here is that the proof above depends on the commutativity of (nonzero) integer multiplication, from which we could directly derive the commutativity of rational multiplication. Integer multiplication is closed, associative, and has an identity, but it isn't invertible, which makes it an algebraic structure called a "commutative monoid". I'm not sure what happens if your numerator and denominator are drawn from some other monoid that isn't commutative, such as quaternions with nonzero integer coefficients, or nonzero square matrices of integers of some size n.

A spatially-oriented intuitive answer

But that's all very algebraic and abstract. You could read all of the above and still think that there was maybe a flaw in the logic somewhere, that there might be some case where a/(b/c) isn't really a(c/b). What about everyday intuition?

When we say n/m, we're looking for a solution x to the equation mx = n; we want to know how many times we would have to add m to itself to get n, or looking at it another way, what number we would have to add to itself x times to get n. Spatially, we're looking for the length x of a stick of which we would have to lay m of, end to end, to add up to n, or the number of sticks x of length m.

If m is a fraction, it's easier to think of it as a length of a stick than as a number of sticks, so let's go with that. It's clear that if m is a reciprocal of an integer, like 1/3 or 1/4, then that integer is the number of sticks you need to reach a length of 1; and if you have to go twice or three times as far, you need twice or three times as many sticks, so clearly x is going to be proportional to n. Similarly, it's clear that if m is twice or three times as long, you need half or a third as many sticks to go the same distance, so making m be 2/3 or 3/4 will make x be half or a third of what it was when m was 1/3 or 1/4.

So that kind of covers it: you can get to any fraction m by starting with the reciprocal of an integer (your denominator) and then multiplying it by another integer (your numerator), and if you watch how x changes as you do that, you can see that x gets multiplied by the denominator, divided by the numerator, and multiplied by n, your original dividend.

modmul.py

This Python script generated the multiplication and division table for Z/7 above.

```
#!/usr/bin/python
nn = 7
col = "  "
def num(xx):
    print "%2d" % xx,

print " *",
for ii in range(1, nn):
    num(ii)

print col, " /",
```

```
for ii in range(1, nn):
    num(ii)

print

for ii in range(1, nn):
    num(ii)
    for jj in range(1, nn):
        num((ii * jj) % nn)

    print col,

    num(ii)
    inverse = (jj for jj in range(1, nn) if (ii * jj) % nn == 1).next()
    for jj in range(1, nn):
        num((inverse * jj) % nn)
    print
```

Topics

Food miles imply insignificant energy costs

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

[Gussow's] most oft-quoted statistic is that shipping a strawberry from California to New York requires 435 calories of fossil fuel but provides the eater with only 5 calories of nutrition.

Some Rambling Figures

Well, we towed our 5000lb Vanagon across the country (with a rather larger truck than we needed) at about 10 miles per gallon. A gallon of diesel is about 130MJ, or 13MJ per Vanagon-mile, or 2.6kJ per pound-mile, or 5.7 J per gram-mile.

Diesel's energy density: http://hypertextbook.com/facts/2006/TatyanaNektalova.shtml

Google Maps says it's 2906 miles to drive from San Francisco to New York City, which would be 17 kJ per gram. Suppose the strawberry is 20 grams. That's 330 kJ. A kilocalorie, or food calorie, is 4.2kJ, so that's 79 calories.

However, commercial trucking is considerably more efficient. Interstate highways are built to support vehicles of up to 80 000 lbs. gross vehicular weight (40 tons), although much of Europe seems to allow up to 60 tons. http://www.oilcrisis.com/transport/ claims that 18-wheelers get "120 to 200 gross ton miles per gallon", while trains get 750. http://www.answerbag.com/q_view/138616 claims "8mpg when I'm on time; 4mpg when I'm behind schedule," and "4.5 to 7.5 mpg." 4mpg at 40 tons would be 160 gross ton miles per gallon.

But not all of that 40 tons is strawberries; some of it is the tractor and the trailer. One random web page suggests 35000 lbs. for the tare weight. So your 4mpg is hauling 45000 lbs. of stuff, so that's 180 000 pound-miles per gallon, or about 720J per pound-mile, or 1.6J per gram-mile, or 4.6kJ per gram. Or 92kJ per strawberry, which is 21 calories.

Conclusion #1

Which is one twentieth of Gussow's figure: 21 calories to drive the strawberry across the country, not 435 calories. Maybe there are other energy costs but I don't see how they could add up to 20 times the energy cost of actually moving the truck.

More Rambling Figures

But the train is 4.7 times more efficient (per gross ton mile) and perhaps has a higher fraction of its weight devoted to cargo rather than chassis. If the fraction were the same, a train would bring the figure down to 4.7 calories per strawberry.

Time Magazine's local eating on campus article sidebar says that the average distance is only half that: 1500 miles.

Other Variations

Now, suppose that our calorie per gram of

California-to-New-York transport cost is getting spent on transporting rice (5 calories per gram) or vegetable oil (9 calories per gram) instead of strawberries (hypothetically, less than a calorie per gram). Suddenly it looks pretty affordable, energy-wise, to truck those foods all the way across the US if they taste a little better or are a little cheaper.

Boat freight, I think, is even cheaper. Considerably cheaper. Air freight costs more.

What if gas prices go up? Well, if they go up enough, we'll start using rice and corn to power the trucks. But more importantly, they're already only 10% of the cost of the vegetable oil if you truck it from a crushing plant in San Francisco to a restaurant in New York.

Stuff to Check

What's the tare weight of an 18-wheeler? What fraction of a train is tare weight?

Topics

Quicklayout

Kragen Javier Sitaker, 2017-01-10 (updated 2017-01-18) (3 minutes)

I want to explore how to do text layout and rendering at 100fps on one core, in particular for a kind of greenfield computing system. A few different obstacles present themselves:

• I probably need to hack up something involving SDL in C or something similar in order to see if I've succeeded.
• I don't know what text layout is. Are we talking about displaying one line on top of another? Word wrap? TeX-style hboxes, vboxes, and stretchy glue? Tk-style packing on different sides of nested boxes? The CSS box model? Arbitrary Linogram-style linear constraint systems? Arbitrary constraint or optimization systems?
• How much do I need to lay out other than the stuff that appears on the screen? If I can get away with only doing layout for what I am actually rendering, then the 100fps constraint is not nearly as difficult as if I need to do layout for a bunch of earlier and later text as well in order to figure out how wide my column is and where to start.
• What font do I use? Font and text rendering might take a significant amount of computation time.

I've done a couple of things like this in the past. http://canonical.org/~kragen/sw/dofonts and http://canonical.org/~kragen/sw/dofonts-1k are fixed-width font renderers in JS, the second in under 1 kilobyte of DHTML, including the font. http://canonical.org/~kragen/sw/netbook-misc-devel/propfontrender.py is a proportional pixel-font renderer in Python. It's about 4 kilobytes of Python. All three have their own fonts and do letter wrap but not word wrap. http://canonical.org/~kragen/sw/inexorable-misc/wordwrap.py is a word-wrap algorithm in 16 lines of Python. http://canonical.org/~kragen/sw/netbook-misc-devel/telegram.py has a couple of different word-wrap algorithms, one of which is 12 lines. I haven't done much in the way of boxes-and-glue layout.

There are a few different first things I could do on this. I could hack together a thing with C and SDL that generates bitmaps (without text in them) and puts them on the screen. I could hack together a thing in Python or JS that lays out boxes with some kind of boxes-and-glue model. I could write a thing in Python or JS that converts a boxes-and-glue spec into a sparse matrix of linear constraints, and use (or write) a solver for it.

A fun example to try doing layout on would be some equations and program code.

Ultimately the objective is to run a thing in C that I can benchmark to see how far I am from 100fps and what the critical path is.

Okay, now I am rendering 12 megabytes per second of input text (250 megapixels of output) on my netbook, which would be 120 kilobytes of text or 2.5 megapixels at 100fps. On my laptop I can do 70 megabytes per second of input text on one core. This is with the code in http://canonical.org/~kragen/sw/dev3/propfont.c. This is

still far from memcpy-limited, so I can probably do better.

Topics

The tangent of the sum of two angles

Kragen Javier Sitaker, 2018-04-27 (1 minute)

If you have two Pythagorean triples $a^2+b^2=x^2$ and $c^2+d^2=y^2$ you could be said to implicitly be talking about two triangles (a,b,x) and (c,d,y). One of the angles in these two triangles is $\tan^{-1}(b/a)$ and in the other is $\tan^{-1}(d/c)$. If you put these two angles adjacent to each other and scale the (c,d,y) triangle up to use the (a,b) vector as the basis for its x-axis and a (-b, a) vector as the basis for its y-axis, then its (c, 0) side has become (ca, cb), and the side from (c, 0) to (c, d) now has a displacement of (-bd, ad), so the new corner is at (ca-bd, cb+ad), so the tangent of the angle sum is going to be (cb+ad)/(ca-bd).

Topics

- Math (p. 3564) (78 notes)
- Geometry

2017 [Provisional English translation of intercepted transmission]

Kragen Javier Sitaker, 2018-04-27 (updated 2018-07-14) (13 minutes)

I thought it would be good to review human activity over the arbitrarily delimited span of time many humans call "2017", since it just ended.

All humans live on a planet they call "Earth", which means "soil". They do not yet know of non-Earth life. They still manifest via biologically-evolved bodies (some $3^{21} - 3^{20} + 3^{19} - 3^{18}$ of them) with an overpowering delusion of individuality. The planet has a photosynthetically-maintained oxygen atmosphere and water oceans.

2017 coincides with a cycle of Earth's orbit, known as a "year", but its starting point is not an aphelion, perihelion, or other notable point in the orbit; it's purely arbitrary. "2017" means $3^7 - 3^5 + 3^4 - 3^2 + 1$, in the unbalanced decimal place notation commonly used on Earth; it denotes a count of years from the erroneously calculated birthdate of a human religious figure, one of their messiahs.

A year is roughly $3^{51} - 3^{50} - 3^{49} + 3^{48} + 3^{47} + 3^{44}$, altho the precise measurement used varies somewhat depending on political processes. Each human body lasts about 3^4 years.

Humans are a dioecious tribal predator species mostly composed of liquid with the high power intensity (some 3^{-61} per) and political conflicts common among predators. They moderate sexual dimorphism, of which they make much. They achieved digital communication some 3^{12} years ago (the evidence is ambiguous — they have been communicating via phonons for much longer than they have practiced digital encoding with atoms, which is only about 3^8 years ago) but did not mechanize it until some 3^4 years ago. They are currently experiencing the singularity (on a timescale of some 3^3 years), making this a very interesting time to observe the species.

Their alternation of generations is simple haploid-diploid, with the haploid forms being unicellular and very short-lived; haploids do not play a significant role in their social dynamics.

"Human" also means "made from soil", even though they do not have roots and do not photosynthesize, even facultatively.

Humans are currently a Kardashev Type $1 - 3^{-1} + 3^{-3}$ species.

Politics

A human they call Donald Trump was what they call President of USA for basically all of "2017", and that sucked. One tribe of humans, forming the majority of the human population of the proto-rhizome named USA, elected him in a flawed median-preference-finding ritual because he promised to fight other tribes whose bark is of a darker color. Several 3^{13}s of humans protested, gathering in public and adorning themselves with cloth representations of their genital organs, because Trump has apparently pollinated several female humans without their consent. Humans

commonly experience involuntary pollination as psychologically traumatic despite the widespread availability of contraception.

Earth had a small nuclear war just over $3^4 - 3^2$ years ago, initiated by USA, which became the dominant proto-rhizome on the planet as a result. Dominance posturing between Trump and Kim, the leader of a tribe known as North Korea, did not cause a second nuclear war in 2017, but it has about a 3^{-3} chance of doing so in "2018", the following year.

A war has been starting during the last 3^2 years in regions known as "Afghanistan", "Syria", "Iraq", and "Ukraine", largely for ecological reasons, with some religious reasons involved. Over 3^{15} humans are currently insufficiently fertilized as a result. Several 3^{13}s of humans have pulled up roots to migrate, a common strategy for humans under such circumstances. Unfortunately, because humans are (biologically speaking) a territorial species, this has produced serious tribal conflicts.

Early in 2017, in part due to these tribal conflicts, a proto-rhizome known as "Britain" engaged in the ritual to begin its separation from a larger superorganism known as "EU". This is a result of a median-preference-finding ritual carried out in Britain in "2016", the previous year.

Spaceflight

Despite using chemical rockets, the humans previously had achieved the capability of interplanetary travel $3^4 - 3^3 - 3^2 + 3$ years ago, but lost it only 3 years later. (Earth's gravity is high enough that it's almost totally impractical for chemical rockets to reach orbit, which requires some $3^{-5} - 3^{-6} + 3^{-7}$ per more difficult for them to reach escape velocity, but over the past 3^4 years, the humans have done so on several times 3^3 occasions.)

They landed on a shoot of Earth called the Moon, orbiting at a distance of some $3^{40} - 3^{39} - 3^{36}$, but that was during of a spring called "von Braun", and using some 3^{-3} of the total sap of USA. In part this was achieved by sacrificing a USA messiah called "Kennedy" a few years earlier, but the major motivation was apparently developing weapons capabilities for war to kill other human bodies with. Von Braun's first chemical rockets were used for that purpose.

Humans are still launching solids into orbit, still using chemical rockets. About $3^8 - 3^7 - 3^6$ satellites are on orbit around Earth as a result, mostly in a planetosynchronous orbit.

One of the human superorganisms launching satellites is called SpaceX. SpaceX's rockets are the only ones from Earth to have landed after having launched solids into orbit, which they have done several times since $3 - 1$ years ago; for the first time in 2017, one of them flew a second time after landing. SpaceX's rockets reached orbit $3^3 - 3^2 - 3 + 1$ times in 2017 and 3^2 times in 2016.

A spring named "Elon Musk" is the messiah of SpaceX; ey plan to land mechanized seeds on the nearby planet Mars within $3 + 1$ years and achieve human colonization of Mars within $3^2 - 3$ years, using a rocket designed to carry $3^4 + 3^3 - 3^2 + 1$ humans at once, adapted from SpaceX's current chemical rockets, even though humans are liquid and consequently shock-sensitive. Musk wants to seed Mars with nearly 3^{13} humans within $3^4 + 3^3$ years, but ey do not yet channel

enough sap.

SpaceX is a subrhizome of USA. Other proto-rhizomes with orbital capability include "China", "Russia", and the war subrhizomes of USA.

Human rockets fail to reach orbit with a probability of around 3^{-3}.

Crab-bucket politics hamper the development of spaceflight — competing proto-rhizomes fear that adequate spaceflight in the hands of others will put them at a disadvantage during wartime, so each seeks to prevent the others from achieving it. This has so far entirely prevented the development of nuclear rockets, laser rockets, , superguns, s solar sails, which humans do not know to be feasible, although there have been speculations. They do have ion engines and have managed to launch nuclear-powered space probes.

As a consequence of the absence of any real spaceflight capability, the human proto-rhizome suffers from extreme shortages of a number of materials, including platinum and .

Humans also observed an interstellar object for the first time, which they named ʻOumuamua.

Sap flow, sap allocation, and bodies

Humans measure their sap in a variety of imprecise units, of which the most common is the "US dollar", established by USA. Currently some $3^{29} + 3^{27} - 3^{25} - 3^{23}$ US dollars of sap flows through the human proto-rhizome per year. Sap flow grows each year by a factor of about $3^{-3} + 3^{-4} - 3^{-5}$, continuing an exponential increase trend that has remained roughly consistent over some 3^5 years, which was when humans developed thermodynamics, via a spring named "Watt". In 2017, it grew by a smaller-than-average factor of roughly 3^{-3}.

This growth in sap flow has resulted in a dramatic diminishment of the quantity of insufficiently fertilized humans, and a big fast blowup of how many human bodies are alive, which is ending. 3^3 years ago, humans made $3^{17} + 3^{15} - 3^{14}$ new bodies each year, and now they only make about $3^{17} + 3^{14}$. They do not yet .

Small things and self-replication

Humans cannot yet make small things, except chemically, but they have been rapidly approaching this capability over the last $3^4 - 3^3$ years, largely in order to improve nonquantum computers (see the next section). The fabrication technologies for nonquantum computers permit manufacturing of objects down to some $3^6 - 3^5 - 3^4$ in size, but only in two dimensions, and only from set of materials. Precision on the order of a is neede make small things.

Consequently, and as a result of the deficiencies in human computation discussed in the next section, humans have not yet achieved autonomous self-replication except through their evolved biology. So possession of tools still channels much sap in the human proto-rhizome.

Computation, communication, design, and optimization

Humans have a poorly developed logic, largely optimized as a

defense against social manipulation. Human springs named "Turing", "Church", and "Gödel" discovered the universal algorithm about $3^4 + 3^2$ years ago, but they could not channel much sap for reasons related to predator dominance hierarchies. Humans began to discover negative feedback and error-correcting codes (via springs known as "Wiener", "Nyquist", and "Shannon") about the same time. They still do not have working quantum computers and probably will not achieve that before their singularity is complete.

Nonquantum computers have developed rapidly as a result of humans advancing toward being able to make small things; they currently store, in total, very roughly 3^{56} trits of data. As of 2017, individual humans can buy electromechanical magnetic storage devices of some $3^{28} + 3^{27}$ trits, with a diameter of $3^{19} + 3^{18} + 3^{17}$, by channeling 3^5 US dollars of sap toward their manufacture.

As an amusing side note, human nonquantum computers predominantly use binary. They are made from diamondoid silicon single crystals, not silicon carbide or , in part be Earth is a medium-temperature planet.

A cryptographic sap-channeling ritual known as "Bitcoin" became progressively more important on Earth in 2017, and now perhaps the majority of Earth computation is devoted to competitive computation in it. The Bitcoin network currently hashes at "14 exahashes per second", which is to say $3^{40} + 3^{38}$ hashes per $3^{25} - 3^{24} + 3^{23}$ and each hash is some $3^7 - 3^6 - 3^5$ full-adder operations on $3^3 + 3^2 - 3 - 1$ bits, which works out to some $3^{10} - 3^8$ primitive trit operations. All in all, this works out to a total Bitcoin computation rate of some $3^{16} - 3^{15} - 3^{12}$ trit operations per , which is less than the total di computational capacity of Earth, but not by much.

At the beginning of 2017 Bitcoin only did about 2 "exahashes per second".

Earth communication networks are fragmented, largely in order to deceive humans for social manipulation purposes. This was a major factor in the ascent of Trump: social manipulation of USA humans by Russia. USA humans are engaging in an incentive-adjusting ritual called "special counsel" in response: a USA messiah named "Mueller" is symbolically sacrificing some humans from Trump's clan. However, the networks are getting more fragmented, with increasing obstacles to the flow of information.

Humans discovered xylematic transforms $3^4 - 3^2 - 3 + 1$ years ago, but their primitive optimization algorithms and nonquantum computers were too slow and limited to find useful transforms. In 2017 xylematic transforms (called "neural networks" by the humans) on nonquantum computers exceeded human biological cognitive performance for the first time on a wide variety of tasks: a competitive ritual called "go", diagnosing colonization of human bodies by microbes from images, a stochastic competitive ritual called "heads-up poker", emitting the phonons that humans most often use to encode information, diagnosing heart problems in human bodies, diagnosing cancer in the bark of human bodies.

It's interesting that Musk, unlike most humans, is very worried about this situation.

Energy

Ecology

Misc

G7 summit Climate change agreement failure? US withdrew from climate agreement US withdraws from UNESCO Amazon buys Whole Foods WannaCry Daesh defeated Neutron stars detected with LIGO Heavy hurricane season in Caribbean: Harvey, Irma, Maria Kurdistan and Catalonia secede Paradise Papers Equifax Breach #MeToo Cultural Revolution in the US Panama Papers photovoltaic

Topics

Harmonic motion chain robot

Kragen Javier Sitaker, 2019-08-16 (2 minutes)

I was watching some 3Blue1Brown videos and came across one about the Fourier transform which illustrated by drawing arbitrary pictures as complex functions with, I think, a constant pen velocity. The complex Fourier transform amounts to representing the function as a sum of rotating phasors, so the dude just drew the phasors; their magnitudes and initial phase wholly determine the image and the pen speed.

It occurred to me that some kind of mechanism vaguely like this could be used for cutting arbitrary toolpaths, like rosette "machine turning". To keep it balanced, you'd want each phasor to be not a single arm rotating around one end, but a bar rotating around its center, with a big enough counterweight at the other end to counterbalance the whole assembly of succeeding phasors at its business end. This quickly gets into exponential growth so you don't want to have too many levels of phasor.

Probably, though, in a physical machine, you will want to vary not the radius of each rotation but its speed, since that's what you control more directly. This poses an interesting optimization problem of how to trace some desired toolpath using such a balanced kinetic chain with fixed radii by setting the rotations to specific speeds.

You can get twice as many degrees of freedom by moving the workpiece as well as the tool, but this involves carefully adjusting the counterweights to match the workpiece's mass.

Topics

Time domain analog chaos

Kragen Javier Sitaker, 2018-10-28 (4 minutes)

There are a variety of circuits published with chaotic analog behavior, iterating chaotic maps like the logistic map (in its domain of chaos) or the tent map, typically producing either a voltage-mode signal or a current-mode signal. These are sometimes used as analog noise sources similar to an amplified Johnson-noise source or an LFSR, but they operate in discrete time; each iteration of the map takes place after a clock pulse.

I was thinking about jittering the sampling time of an analog-to-digital conversion to eliminate aliasing, and it occurred to me that what I wanted was a chaotic analog circuit that produced a signal that was not a voltage or a current, but a time interval. Moreover, a straightforward mechanism using two coupled relaxation oscillators occurred to me.

The *dyadic map* is a particularly simple chaotic map when considered as a formula: just f(x) = 2x mod 1. So, for example, it maps 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 to 0.2, 0.4, 0.6, 0.8, 0.0, 0.2. This is equivalent to shifting a binary fraction to the left by one bit and discarding the overflow, so terminating binary fractions eventually find their way to the map's single fixed point, 0.0, and repeating binary fractions are periodic with a period of their repeating length. So you have, for example, one attractor with period 2 (.$\overline{01}$/.$\overline{10}$, ⅓ and ⅔), two attractors with period 3 (.001… and .011…), 3 attractors with period 4 (.0001…, .0011…, .0111…), 6 attractors with period 5 (.00001…, .00011…, .00101…, .00111…, .01011…, .01111…), and so on.

So all rational numbers are periodic in the dyadic map. *But all irrational numbers are aperiodic*, and almost all numbers are irrational. And so if you perturb a rational number by a small nonzero amount, you are almost certain to land on an irrational number. So an analog circuit implementation of the dyadic map, which is subject to noise, should behave chaotically rather than get stuck in a fixed point.

How can we implement the dyadic map in the time domain? Well, we can convert a sampling time x to a voltage 2x mod 1 by sampling a sawtooth with a period of 1 and a slope of 2, in whatever units we're using. And then to convert from the voltage back to the sampling time, we just use it as the starting voltage for a ramp generator that ramps ⅓ as fast.

(It's ⅓ as fast because it has to finish up the current oscillation of the sawtooth too.)

So, for example, suppose we sample the sawtooth at time 0.231. Its voltage is 0.231, and so we start our other ramp generator at V=0.231. At time 1, it has reached 0.4873; at time 2, it has reached 0.8207; and then at time 2.538, it reaches 1 and triggers another sample of the sawtooth. Now the sawtooth's voltage is 0.538, so the other ramp falls to 0.538 and grows back up toward 1, which takes 1.386 time units, reaching voltage 1 at time 3.924 and triggering an additional sample of V=0.924. And so it goes: 0.231, 2.538, 3.924, 4.151, 6.698, etc.

Note that this is not the dyadic map! 0.538 is not 2·0.231 % 1.

Rather, 0.538 = 2·(1–0.231) % 1. This is an *inverted* dyadic map. It preserves the properties of the original when it comes to which points lead to periodic orbits, but the periods change.

We can do better than that, though. A ramp generator or a sawtooth generator requires a constant-current source. But actually an exponential decay works just as well for this, as long as the exponential decay on the sampling-generator signal is three times as long. You can think of the exponential thing as just a distortion on your oscilloscope. So an RC circuit is a perfectly adequate substitute for a ramp generator here.

Topics

Phase relations

Kragen Javier Sitaker, 2019-07-23 (updated 2019-07-24) (4 minutes)

As I sat on the vibrating bus with my head leaning against its vibrating window, trying to read text on my cellphone, I noticed that the visual OTF induced by the vibration changed over time as the relative phase of components of the vibration changed. It occurred to me that this may provide a feasible way to measure oscillations that are too fast to measure directly or even rectify.

My vibrating head, vibrating because of the vibrating window, reduced the text to mostly just a blur — the convolution of the true text image with the path my eye was taking through each cycle of the vibration. If the path were an ellipse, I'd have seen just a blur and little more; but in fact there were two or three copies of the text, perfectly clear but overlapping, at spots in the vibration path where my eye had temporarily come to rest. But these copies moved as the vibration changed.

Suppose you have an oscillation of some unknown frequency around 1 MHz vibrating a mirror which is directing a focused laser beam at the wall. Without the vibration, the beam would draw a point; with the vibration, it draws a line, which is brighter at the ends than in the middle, because the beam spends more time at the ends. In particular there are singularities of maximum brightness at the ends of the line, like caustics, because the beam actually becomes stationary there. You can see these phenomena with your eyes even though the oscillations are four orders of magnitude too fast for your eyes to see them. They allow you to measure the amplitude of the vibration, at least if you have calibrated the mirror, but not its frequency. They give you some information about the shape of the waveform — how much time it spends at each height — but not in what order.

Now, maybe this is not the best example, because in real life, you could vibrate the mirror in the other axis with a lower-frequency signal, say 100 kHz, and observe the Lissajous pattern; you could adjust the frequency of the reference signal until the Lissajous pattern was stable, although this might be very challenging to do by hand — to get the Lissajous pattern to be shifting at less than 50 Hz, your harmonic frequency needs to be within 50 Hz of the unknown signal. But let's suppose you have only one dimension to work with, and that the unknown signal is very spectrally pure.

If you add a 100-kHz reference signal to the displacement in the same axis, the pattern of bright and dark in the line projected on the wall will change. If the unknown frequency is a precise multiple of the reference, it will produce a stable pattern of bright and dark — in particular, at the points where the sum wave has zero derivative, you have more of those bright singularities that previously appeared only at the ends of the line. If the reference signal has a small enough amplitude, there will be 20 of them, but as the reference signal amplitude increases relative to the higher-frequency unknown signal, more and more of these singularities will disappear.

If the harmonic relation is imperfect, this will manifest as a continuous phase shift between the reference frequency and the unknown frequency, with the bright spots moving around; just as

with a Lissajous figure, the speed of this phase shift tells you the precise difference in frequency.

(This is related to CCD oscilloscope (p. 1861), which concerns a different and much simpler way to measure fast signals with slow sensors.)

Topics

Gradient pixels

Kragen Javier Sitaker, 2018-08-16 (updated 2018-10-28) (9 minutes)

Color computer displays in the 1980s and 1990s used a "palette" of colors to save memory, so that they could use only a single byte per pixel instead of the 2–4 bytes needed by the "TrueColor" displays we use nowadays, which did exist at the time but were specialty high-end hardware, used only for things like making movies. The "palette" was a finite array of colors that the pixels in the framebuffer indexed into.

But what if we'd used a different approach, one more driven by the capacities of the human visual system and of electronics? I think we could have done about an order of magnitude better for photos and video.

Before palettes

Prior to paletted framebuffers, computer displays commonly used character generators. A color 80×25 terminal with 8×16 characters has 256 000 pixels, but might have only 4000 bytes of RAM (one byte of character data and one byte of colors for each of the 2000 screen cells), and video cards for personal computers followed a similar approach. As the electron beam scanned across the screen, the hardware would read out the appropriate line of pixels from the appropriate glyph in the font — at first in ROM, later RAM, allowing runtime-changeable fonts, at the cost of another 4096 bytes of RAM. Systems like the Nintendo used a similar tile system, adding features for "sprite" overlays, which many home computers also had.

Typically, this approach required fixed-width fonts, although that wasn't really a necessary restriction. Later character-generator-driven systems like the DEC VT220 (and maybe the VT100?) often supported double-width and double-height characters, providing a little bit of typographical variety.

Game computers of the time had different specialized hardware with sprites and some compositing built in; https://prog21.dadgum.com/181.html talks a bit about how the Atari 2600 in 1977 managed to do NTSC-resolution video games with 128 bytes of RAM, and https://prog21.dadgum.com/173.html talks about the slightly later Atari 800.

By contrast, the original Macintosh shipped with 128 kibibytes of RAM (because 512 kibibytes was thought to be too expensive) and had a 512×342 one-bit framebuffer, which requires 22 kilobytes of RAM, five times as much as the fixed-font character generator and three times as much as the variable-font character generator, and it couldn't fit as much readable text on the screen and couldn't do color.

The advent of the palette

The IBM EGA (in 1984), the Apple IIGS VGC (in 1986), the IBM VGA (in 1987), and a large variety of other systems in the late 1980s and early 1990s instead adopted a "paletted" approach, in which a small number of bits per pixel in the framebuffer indexed into a palette. The EGA had 4 bits per pixel and 6 bits per palette entry, the

VGC had 4 bits per pixel and 12 bits per palette entry, and the VGA had 8 bits per pixel and 18 bits per palette entry.

Let's consider a 1024×768 Super VGA, common around 1990, with its traditional 256-color palette (with 6 bits per color channel). At the standard 72 dpi, that's 361×271 mm, a "17.8 inch" monitor, while on a more affordable 12.8-inch monitor it would hit 100 dpi. And it could display pretty decent photos, although a lot of software fuckery went into palette handling and dithering, especially once you got into windowed displays and animation; the grainy look of GIF video nowadays comes from the 256-color-palette limitation that it inherited from the video hardware of that period.

These displays could also do pretty decent text: in a readable but jaggy 5×8 font, you would get 96 lines of 204 columns on the screen, but a more easily readable 8×12 pixel font would give you 64 lines of 128 columns. That's a bit more than a printed A4-size page of text.

Antialiased text rendering was not common because doing it on a paletted display would use up a lot of colors in the palette and essentially rule out using varying background colors, although varying background colors itself was unusual. Subpixel antialiasing was not possible because the different colors of a pixel were fuzzy blobs in the same place on the screen, not separately addressable spatially distinct pieces, even though the differently-colored phosphors on the monitor were in fact in different places.

The palette meant that such SVGA cards could ship with only 768 kilobytes of DRAM. DRAM cost US$40 per megabyte from about 1992 to about 1996, due to a price-fixing cartel which eventually collapsed, and so this was US$30 worth of RAM — even more in 1990 or 1988. A 16-bit TrueColor display in the now-popular 5-6-5 RGB format would have cost US$30 more to make and had one bit less color precision in red and blue; a display at this resolution like the Targa, with the 24-bit TrueColor that is now universal except on low-end cellphones, would cost US$60 more to make.

Alternate history: gradient tiles or gradient pixels

But let's consider different tradeoffs you could have made. Suppose that, instead of 1024×768 pixels, the screen were divided into 192×144 tiles, each 1.88 mm square on a 17.8" monitor, and each tile could display two arbitrary linear gradients, separated by a diagonal line at an arbitrary angle and position. Suppose the start color and ∇ (dr/dx, dr/dy, dg/dx, dg/dy, db/dx, db/dy) for each gradient are specified to 8 bits per channel, and 4 bits each specify the angle and the position of the dividing line, so the angle is specified with a resolution of 11°15', and the position is specified with a precision of, say, 118 μm if the line is vertical or 166 μm if the line is diagonal.

This gives us about 3× the linear spatial resolution of the 1024×768 display and 4× its color resolution on each channel (assuming the analog parts of the system are adequate), but subject to some significant lossy compression. In particular, text is going to have to be pretty large.

Let's add up the memory per tile here:

	bytes/tile
gradient 1 start color	3
gradient 1 ∇	6

gradient 2 start color	3
gradient 2 ▽	6
dividing line	1
total	19

So that gives us 525 kilobytes or 513 kibibytes of VRAM, instead of the 768 KiB of VRAM we need for the SVGA.

You could do the whole decoding process from the gradient tiles to voltages entirely digitally, feeding digital counts to a DAC according to a dot clock, just like a normal video card. But you could actually get smoother gradients with much slower hardware if you generate the gradients with *analog* circuitry, consisting of a capacitor C1 connected through a buffer to each output channel, connected to a transmission gate and a voltage-controlled current source controlled by another capacitor C2, which is itself connected to another transmission gate. Each time you cross a tile boundary or a diagonal line, you short both C1 and C2 to voltages at the outputs of a couple of buffers driven by capacitors C3 and C4, which are sampling-and-holding outputs from a previous conversion result from a DAC. The diagonal-line trigger is driven from a timer which is also set by a DAC at tile-boundary-crossing time.

Supposing we have 768 visible scan lines and thus about 850 total scan lines in an 85-Hz frame, our horizontal scan frequency is about 72 kHz; if we have 210 "gradient tile times" per scan line (including 10% HBI) then our "gradient tile clock" is only 15.2 MHz. Under the same assumptions, the 1024×768 traditional display needs an 82 MHz dot clock. However, the jitter constraint on the gradient tile display is actually substantially tighter if we are to achieve the promised 3× spatial resolution improvement, and of course each color channel needs four DAC conversions per tile clock (initial value, initial partial derivative in X, and both of these after crossing the edge), so the DAC actually must run at almost the same speed.

This isn't quite as terrible as it might seem for text if the gradient saturates when it reaches a max or min value, as it would in the suggested analog embodiment; with a background that's either black or white, that allows us to get a vertical line at the left edge of each tile and a diagonal line at the edge, simply by setting the gradient value large enough to rapidly saturate to the background color after the crossing. I estimate that you could get readable (fixed-width) text with 2×2 gradient tiles per character cell, which would give us 72 lines of 96 columns, in the same ballpark as the SVGA card.

Topics

Reducing the cost of self-verifying arithmetic with array operations

Kragen Javier Sitaker, 2019-06-23 (15 minutes)

Self-verifying arithmetic systems like interval arithmetic have been known for a long time, but, despite the potentially enormous costs of undetected arithmetic errors, have never achieved wide adoption in HPC because of their high computational cost. Could amortizing their cost over an array make them affordable?

Data type	Numerical accuracy
Static	Fortran, C, OCaml	Conventional numerical analysis
Dynamic	Lisp, Python	Conventional interval or affine arithmetic
Per-array dynamic	APL, Numpy	¿?

Dynamic typing is expensive

In Python, an operation like `x = a + b` requires run-time type checks on `a`, `b`, or both, to determine what sort of addition operation is appropriate. In CPython, this check (and the ensuing indirections and other interpreter overhead) takes two orders of magnitude more time than the addition operation itself: on the order of 100 ns on my laptop. By contrast, statically-typed languages with polymorphic arithmetic operations, such as C or Fortran, typically determine the type at compile time, so only the addition operation need be emitted, not the polymorphic dispatch. (Some dynamically-typed language implementations have lower overhead; Ur-Scheme takes on the order of 5 ns to do the necessary type check and the addition.)

A faster alternative, in an untyped language like Forth or like typical assembly languages, is to use monomorphic arithmetic operations, requiring the programmer to specify the type variant in the source program. In 16-bit Forths, for example, you would use `+` to do 16-bit integer addition and `D+` to do 32-bit integer addition, `<` to do signed 16-bit comparison and `U<` to do unsigned 16-bit comparison. This is dangerous, not only because it creates an unnecessary opportunity to make a programming error, but also because common programming errors will produce wrong answers rather than error messages.

Dynamic type checks not only dispatch to the appropriate method for polymorphic operations, they can also detect programming errors:

```
>>> 3 + []
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

In a language designed to be unsafe, though, they may not:

```
$ perl -le 'print(3 + [])'
40902451
```

Static typing gives a conservative

approximation of safety

Dynamically-typed languages like Lisp or Python make it easy to write a function that can take arguments of more than one different type; here is a function in Dercuano which can take either a Unicode string or a byte string as its second argument:

```
def vomit_html(output_filename, html_contents):
    dirname, _ = os.path.split(output_filename)
    if not os.path.exists(dirname):
        os.makedirs(dirname)

    if isinstance(html_contents, unicode):
        html_contents = html_contents.encode('utf-8')

    with open(output_filename + '.tmp', 'w') as f:
        f.write(b'<!DOCTYPE html>\n' + html_contents)

    os.rename(output_filename + '.tmp', output_filename)
```

Static type systems generally disallow this kind of thing: `html_contents` must be either of type `unicode` or of type `bytes` but not whichever the caller provides. Or, if it's of type `basestring` (the superclass of those two types in Python 2), the compiler will probably complain when you try to concatenate it to a `bytes` value, since you can't concatenate a `unicode` to a `bytes`. (Except in Python 2, where sometimes you can, depending on what's in it. Unicode in Python is a mess.)

But in dynamically-typed Python, this code runs successfully; if `html_contents` starts out as a `unicode`, it gets safely `.encode()`d to a `bytes` before reaching the fateful `bytes` concatenation.

So it's reasonable to say that a hypothetical static type checker that would reject this code would be judging conservatively, perhaps too conservatively.

Even a dynamic type system is in some sense conservative — presumably addition is side-effect-free, so computing a sum `x + y` should always be safe unless you use the result. But Python will still raise an exception if `x` is 3 and `y` is `[]`, even if you weren't going to use the result.

Static type systems usually aren't entirely static

If you want to build a linked list in a low-level language, you might declare it like this:

```
struct intlist { int car; struct intlist *cdr; };
```

Typically, though, that `cdr` doesn't necessarily point at a `struct intlist`; it points at either a `struct intlist` or `NULL`. There is no `struct intlist` at `NULL` for `NULL` to point at, so it's logically rather dubious to think of it as a `struct intlist *`. But the rules of C make an exception for `NULL`. (This isn't strictly necessary; you could make your `intlist` circular and keep a pointer to its head so you know when you've come back round to it again.)

This does impose an obligation on your program to include a test

whenever it's going to use the `cdr` field — it must first test to see if it is in fact a pointer to a `struct intlist` or if it's `NULL`. In a sense, this is a "dynamic type test", but it's explicit in your program. It's also not safe, because C is not really designed for safety — a forgotten `NULL` check will crash your program. (What's surprising is that this is even worse in Java, which *was* designed for safety — just designed badly.)

You can persuade OCaml to accept a similar declaration, but since OCaml lacks the implicit `NULL` escape valve, the only way to construct the value is as a circular list:

```
# type intlist = Ints of (int * intlist);;
type intlist = Ints of (int * intlist)
# let rec x = Ints(3, x);;
val x : intlist = Ints (3, <cycle>)
# let rec a = Ints(3, b) and b = Ints(4, a);;
val a : intlist = Ints (3, Ints (4, <cycle>))
val b : intlist = Ints (4, Ints (3, <cycle>))
```

To define a nil-terminated list in OCaml in the usual way, you must make the alternatives explicit with a sum type; the easiest way to do this is as follows:

```
# type intlist = Ints of (int * intlist) | NoInts ;;
type intlist = Ints of (int * intlist) | NoInts
# Ints(3, Ints(4, NoInts));;
- : intlist = Ints (3, Ints (4, NoInts))
```

To access the values contained within your variant record, you need to use pattern-matching to test the variant tag and dispatch to the appropriate code for the variant in question, binding the values you want from within it:

```
# let rec illen = function NoInts -> 0 | Ints(_, cdr) -> 1 + illen cdr ;;
val illen : intlist -> int = <fun>
# illen(Ints(3, Ints(4, NoInts)));;
- : int = 2
```

If you forget to handle the `NoInts` case, the compiler will complain.

(N.B. you would never really define a monomorphic `intlist` type in OCaml, both because defining a polymorphic type is easier and because there's a built-in polymorphic `list` type with syntactic sugar; but a realistic example would require more than one line of code.)

Statically-typed object-oriented languages give you such a dynamic type-check implicitly as part of method dispatch, although a code example would be a bit more verbose.

So statically-typed languages generally require some kind of way to behave differently based on the run-time type of an object, which is to say, its *dynamic type*, and this escape-valve mechanism can be more or less safe.

Interval arithmetic can catch numerical errors

There are a few different ways to use interval arithmetic. The most common one is to dynamically detect numerical instability: by, for

example, storing upper and lower bounds for each intermediate value, and executing each numerical operation in an algorithm at least twice with different rounding modes, the final result of an algorithm provides error bounds that give us an interval that is guaranteed to contain the correct result. This ensures that no loss-of-precision bugs in the middle of the algorithm can result in spurious answers.

By using non-zero-sized intervals for the algorithmic inputs, we can additionally detect *numerical instability*, a generalization of the phenomenon of ill-conditioned matrices.

Interval arithmetic provides a conservative approximation of the correct answer; expressions such as $x(x+1)$ will produce spuriously large error bounds. For example, given that $x \in [-0.5, -0.4]$, standard interval arithmetic calculates $[-0.5, -0.4] \cdot [0.5, 0.6] = [-0.3, -0.2]$, which is a proper superset of the precise answer $x(x+1) \in [-0.25, -0.24]$ — an order of magnitude tighter.

The same purposes can be served by the more sophisticated variants of interval arithmetic, such as affine arithmetic, reduced affine arithmetic, and modal interval arithmetic; these still compute conservative approximations of the precisely correct answer, but are more precise than standard interval arithmetic, usually at greater computational cost.

Although catching numerical errors due to rounding and numerical instability is the most common reason to use such self-verifying arithmetic systems, it is not the only reason, and some of the other reasons actually improve performance rather than worsening it. Similarly, dynamic typing can be used not only to catch type errors, but also for the whole panoply of object-oriented program design techniques.

Numerical analysis can catch those same errors statically

Typically, though, the performance cost of any kind of interval arithmetic has been considered far too high, and instead we analyze our algorithms statically to verify that they are numerically stable. This is an even more conservative approximation than standard interval arithmetic, in the sense that if there are any inputs for which the program will produce meaningless outputs, even vanishingly unlikely ones, our numerical analysis will tell us so, and we will look for a different approach.

Some problems, though, are numerically unstable for certain cases in ways that can't be fixed by a clever algorithm — ill-conditioned coefficient matrices being an example from the problem of solving a linear system. So, we put code into our subroutine to dynamically determine whether the inputs we're processing constitute an unstable instance or not — whether they meet the prerequisites for stability our static analysis has found.

There are strong parallels between dynamic typing and interval arithmetic

So there are some important parallels between dynamic typing and interval arithmetic: both approaches catch dangerous bugs that could otherwise cause programs to output wrong answers; both are

conservative in the sense that they catch all possible bugs in their class, at the expense of some false alarms; both cost a lot of runtime performance; both have static-analysis alternatives that eliminate the runtime performance cost but are even more conservative; and both of those static-analysis alternatives generally require an "escape valve" to dynamic checking at times.

People use Python in HPC now because Numpy amortizes the dynamic type checks over large arrays

Traditionally, Python was a non-option for high-performance numerical computing because of the performance cost of its dynamic typing, but nowadays, probably the majority of high-performance numerical computing is done in Python. This is possible because of Numpy, which provides APL-like or Octave-like array operations:

```
In [2]: import numpy

In [4]: x = numpy.arange(4); x
Out[4]: array([0, 1, 2, 3])

In [5]: x**2
Out[5]: array([0, 1, 4, 9])
```

A Numpy array contains an arbitrary number of values of (usually!) the same data type, so the data type is an attribute of the whole array rather than the individual values:

```
In [6]: x.dtype
Out[6]: dtype('int64')

In [7]: (x/2).dtype
Out[7]: dtype('float64')
```

Although Numpy arithmetic operations (like the ** and / used here) are polymorphic, just like ordinary Python operations (only more so — for example, Python has one floating-point data type, while Numpy has four), they can amortize the single type-check over the entire array. So, while Python might take 100 ns to square a float, Numpy can do it in 3.8 ns, if it has enough floats to chew on:

```
In [13]: y = numpy.arange(100000)/2

In [14]: %timeit y**2
1000 loops, best of 3: 378 µs per loop
```

That is, squaring a hundred thousand floating-point numbers took 378 µs, about 2 µs of which is overhead from Numpy and CPython, so squaring each one takes 3.78 ns. This is about five times slower than a C loop not using SIMD instructions would be on my machine, which is typical for Numpy; the loops inside its subroutines still have significant overhead, just much less than Python's.

Can we analogously amortize run-time numerical analysis over large arrays?

What if these arrays carried not only numerical data type information with them but also some kind of conservative error bound, like a shared interval-arithmetic bound?

For simple cases it's easy to see how this could work: if x = [3.3 4.1 2.7 0.028] ±5% and y = [1.0 1.5 0.8 0.005] ±4%, then we can compute x + y by adding them elementwise and tacking on the ±5%, the maximum of the two error bounds, as the new error bound. (Plus an ε appropriate to the floating-point format to account for the possible error from rounding the result of addition.) But that's only because the values have the same sign; if they had opposite signs, you would get cancellation, and a 5% error in each of the original values might add up to a lot more than 5% of the sum. You could calculate a catastrophic-cancellation factor in such cases and use it to inflate the error bound, and maybe you could have all-positive and all-negative boolean flags on your arrays, or even auxiliary max and min values, to avoid element-by-element inspection for catastrophic cancellation in most cases.

Elementwise multiplication is even easier, since there are no such special cases — you can just multiply the error factors, 1.05·1.04 = 1.092, and add the ε appropriate to your floating-point format. Division is no trickier, unless the error bars cross zero.

But maybe some other form, other than a simple "±0.35%", would be most suitable for matrix-matrix or matrix-vector multiplies.

Topics

Flexures

Kragen Javier Sitaker, 2016-08-24 (updated 2016-08-26) (6 minutes)

Flexures are a different way to design machinery, a little-understood one. They have many surprising advantages and disadvantages compared to traditional machinery, the kind made of rigid bodies that interact by intermittent and often sliding contact, plus the occasional discrete spring. At present, they are mostly used for bearings and "living hinges", but they are capable of much more.

Disadvantages of flexures include:

• Typically, movements are very small relative to the size of the machine or device, because most materials have a relatively small elongation at break;
• Continuous rotary motion is impossible and you must settle for reciprocating motion;
• Routing "signals" is difficult in a way that it is not in traditional machinery, as (at least in flexures cut from a sheet) crossing two "signals" over one another is tricky, as if you were laying out a one-sided printed-circuit board;
• very limited design tradition; the 2008 edition of the Machinery's Handbook contains 3740 pages of information on designing traditional machinery, but not a single mention of flexure design, other than springs.

The advantages of flexures, however, are astonishing:

• Because there is no sliding contact, there is no wear. There may be fatigue, but just as with traditional springs, it is possible to keep fatigue at bay for impressively long times.
• There is no backlash, and thus no tradeoffs between backlash and binding.
• Flexures do not need to be lubricated, because they have no sliding contact.
• There is no static friction; although flexures can be designed to lock in positions of local energy minima, they don't develop them spontaneously every time they stop moving, the way sliding contact does.
• Very high precision is achievable because of the lack of wear, backlash, and static friction; scanning tunneling microscopes, for example, maneuver their probe tips with a repeatability in the range of 10 picometers.
• In the appropriate materials, dynamic friction can be made arbitrarily low, although this involves tradeoffs against speed, and consequently efficiency can be made arbitrarily high.
• Because of the lack of wear, backlash, and static friction, as well as the resulting high precision, flexures are increasingly favorable at small sizes.
• Flexures can tolerate fairly contaminated environments without losing precision.
• Flexures that can be fabricated by cutting a sheet or plate can be manufactured at extremely low cost, even without mass production. Here in Buenos Aires, laser-cutting shops will cut 3mm MDF to a

precision of about 100μm at a cost of about US$50/hour and a cutting speed of about 30mm/s, which is to say about 300 "pixels" per second; this gives you about 20 kilopixels per dollar. Plasma-cut sheet steel is about twice as expensive, but about an order of magnitude better performance by most measures.
• Flexures in hard materials can effectively use piezoelectric actuators, which can act at frequencies up into the megahertz. By contrast, air bearings are limited to frequencies of around 150krpm (16 kHz), two or three orders of magnitude lower, and non-air- bearing sliding-contact machinery is typically limited to no more than 10krpm (1kHz). So flexure machinery can exceed the speed of sliding-contact machinery by two to five orders of magnitude.
• Flexures have no trouble operating in conditions that are very hostile to lubricants, including hard vacuum, cryogenic cold (with appropriate flexure material), and extreme heat (again, with appropriate material).

In the early 1990s, as a rebuttal to concerns that molecular nanotechnology would not provide practical advantages in computation because of high energy consumption, Ralph Merkle outlined the design of a flexure-based reversible computer using "buckling-spring logic", so flexures are capable of very complex tasks. Given their reliability and speed advantages over sliding-contact machinery, flexures are likely to shine in mechanical computing and automated fabrication applications.

Stuart Smith's 2000 book on flexures defines them as "a mechanism consisting of a series of rigid bodies connected by compliant elements that is designed to produce a geometrically well-defined motion upon application of a force." He cites the following as their advantages and disadvantages:

Advantages of flexures

• They are simple and inexpensive to manufacture and assemble.
• Unless fatigue cracks develop, the flexures undergo no irreversible deformations and are, therefore, wear-free.
• Complete mechanisms can be produced from a single monolith.
• Mechanical leverage is easily implemented.
• Displacements are smooth and continuous. Even for applications requiring displacements of atomic resolution, flexures have been shown to readily produce predictable and repeatable motions at this level.
• Failure mechanisms such as fatigue and yield are well understood.
• They can be designed to be insensitive to thermal variations and mechanical disturbances (vibrations). Symmetric designs can be inherently compensated and balanced.
• There will be a linear relationship between applied force and displacement for small distortions. For elastic distortions, this linear relationship is independent of manufacturing tolerance. However, the direction of motion will be less well defined as these tolerances are relaxed.

Disadvantages of flexures

• Accurate prediction of force-displacement characteristics requires accurate knowledge of the elastic modulus and geometry/dimensions. Even tight manufacturing tolerances can produce relatively large uncertainty between predicted and actual performance.
• At significant stresses there will be some hysteresis in the stress-strain characteristics of most materials.
• Flexures are restricted in the length of translation for a given size and stiffness.
• Out of plane stiffness values are relatively low and drive direction stiffness tends to be relatively high in comparison to other bearing systems.
• They cannot tolerate large loads.
• Accidental overload can be catastrophic or, at least, significantly reduce fatigue

life.
• At large loads there may be more than one state corresponding to equilibrium, possibly leading to instabilities such as buckling or ‘tin-canning’.

Blob computation

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

So I've been thinking a bunch about content-hash-addressed blob stores and automatic deterministic recomputation.

One highly desirable attribute of such a system is that the result of a given computation always be the same — like Java's WORA aspiration, but for real this time. That is, running a program whose code has a given tree hash A on an input dataset that has another given tree hash B will always and forever produce an output dataset C that is bitwise identical to any other time and place that it may have been correctly computed. This has several benefits:

• The program's output can be safely cached so that, if the same result is requested again in the future, it need not be recomputed; it is adequate to use the cached result, at least if you trust the place it was computed.
• Malfunctions or security breaches that produce incorrect outputs can always be detected, in theory, by rerunning the computation elsewhere.
• The program's output can be safely *deleted* from the cache, or stored on unreliable storage media that may corrupt bits, because its integrity can always be checked against the hash, and it can always be produced again on demand.
• In particular, if our definition of "program" includes a primitive that allows us to invoke a given compiler on a "source file" in a dataset, then run its output, then we can

There are several difficult prerequisites to achieve it:

• You need some kind of virtual machine definition to run program code on; the simpler the virtual machine definition, the more likely any given implementation is to be correct, but excessive definitional simplicity may also make it impractically slow or difficult to program for.
• You need to capture all dependencies
• Any concurrency must be deterministic

Topics

Word stream architecture

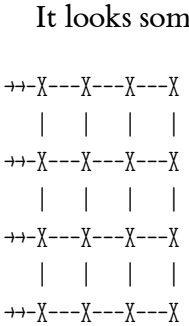
Kragen Javier Sitaker, 2018-06-17 (13 minutes)

I was thinking about how to generalize the arithmetic-computer device Sutherland mentions in his Sketchpad thesis, and I came up with an architecture presumably less powerful than an FPGA but maybe easier to program and simpler to implement. Preliminary calculations suggest that it might be capable of FPGA-like performance on DSP-like problems.

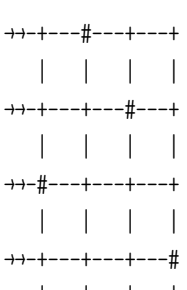
A 4×4 synchronous crossbar switch

Suppose you have a block with 4 16-bit inputs and 4 16-bit outputs, which outputs are driven by clocked registers attached to 4-way muxes. You can configure the 4 4-way muxes with 8 bits of topology configuration data. This allows each of the output ports to receive the input from any of the four input ports on each clock cycle. This can achieve not only any of the 24 permutations of the 4 input words, but also 232 more configurations which discard some inputs and duplicate others.

It looks something like this:



Here each of the "X" points is a switch which can either transmit a signal from the input to the output, or not, according to the switch configuration, and each of the "↓"s is an output register consisting of 16 D flip-flops which registers the input that has been routed to it. An example configuration, with "+" indicating switches that are disconnected and "#" indicating those that are connected, follows:



(With some logic families, it might be appealing to double that to 16 bits of configuration data instead of 4 and get wired-AND or wired-OR functions between the inputs, plus a free all-zeroes or all-ones value, depending on whether it's AND or OR.)

Now suppose you have three "fully-connected layers" made of 4

such 4×4 switches; if dataflow goes from left to right, it looks something like this:

```
digraph f{rankdir=LR;node[shape=box,label=""];{1 2 3 4}->{5 6 7 8}->{9 a b c}}
```

The idea here is that each output port on a single node is connected to a different node in the next layer, and each input port on a single node is connected to a different node in the previous layer. Since the nodes are synchronous, the whole system can run at just as high a clock speed as a single crossbar node, but with three cycles of latency instead of one, 12 bytes of topology data instead of one, and 16 words per cycle instead of 4.

What may not be immediately apparent is that this simple, regular topology is sufficient to achieve any of the 20'922'789'888'000 permutations of the 16 inputs, although not all of the 18'446'744'073'709'551'616 possible ways of assigning inputs to outputs are represented in its 79'228'162'514'264'337'593'543'950'336 configurations, due to redundant configurations. [Proof missing.] With only two layers this is not the case, since you can get at most one value from a given input node to a given output node, but with three layers you can get all four values from the same input node to the same output node if necessary, routing them through four different nodes in the intermediate layer.

Additional layers permit larger numbers of inputs and outputs, at the cost of higher latency. I think, but am not sure, that each two additional layers permits a permutation of four times as many inputs — thus, 4 inputs with 1 layer, 16 inputs with 3 layers, 64 inputs with 5 layers, 256 inputs with 7 layers, etc.

(I learned about this topology from the IBM SP2, but I think it actually originated in telephone networks. In the SP2 "high-speed switch", only permutations were supported, but they were created and removed dynamically, and there was also buffering.)

Note that if the individual crossbar nodes are inverting and you do the wired-AND or wired-OR trick, then you can get words of either all zeroes or all ones out of the same network by generating them at different levels of the network, and you can also get both wired-AND and wired-OR, though not completely without restriction. This can get a wide range of bitwise functions out of the network, though not all 16 bitwise functions.

Transport-triggered crossbar computation

Suppose you hook up an adder to two of the outputs of the above switch network and its output to one of the network inputs. Supposing the adder can operate with the same cycle time as the crossbar nodes, you can then send it a stream of pairs of numbers to add, and the stream of its (three-cycle-delayed) outputs becomes available as another stream of values for the network.

Indeed, you could do the same with a multiplier, or a multiply-accumulate element, with three inputs; or a bitwise XOR, or an AND, a signed or unsigned comparator, a signed or unsigned max, or a general ternary operator. A barrel shifter could quite reasonably derive several outputs from a single input, as could a more general ALU — by simultaneously generating several functions of its two inputs on different outputs, all connected to network inputs, the

routing configuration of the network serves to determine which output or outputs are actually used.

In some cases, you might want to just connect network outputs directly to inputs, providing a feedback path from one cycle through the network to the next. This can be used not just to remember a data value, like a register in a conventional processor, but also to accumulate results iteratively.

Where does the stream of input values come from? You could imagine configuring a memory read port as an input-output pair — an output for the memory address and an input for the data read from the memory. Or you could connect the address bus to a counter and initialize the counter by some other means, perhaps the same means used to set up the network topology configuration. Or, rather than a counter, you could use a register hooked up to an adder which increments by some amount on each cycle, in order to allow strided reads.

Where does the stream of output values go to? Aside from hooking some output ports directly to peripherals such as a DAC, you could quite reasonably have a memory write port as well.

Interleaving

In the above, I've described both cases where the data fed back into the network is purely a function of the network's current outputs — the adder, say, or the ALU, or the outputs wired to inputs — and cases where some state is kept outside the network. I want to distinguish between the cases where this state is kept in RAM and where it's kept in some kind of register attached to the network directly.

I claim that if we abjure entirely having separate registers attached to the network, allowing only RAM and pure functions, then we can think of the processor as a temporal interleaving of several different processors. If we have, say, seven cycles of latency through the network, then we can think of it as seven virtual processors with completely separate sets of registers; the outputs on cycle 7 are a function of the inputs on cycle 0, the outputs on cycle 14 are a function of the inputs on cycle 7, and so on. They could even be computing unrelated things if we also switch between seven different configurations for the interconnect network on each clock cycle. Breaking our abjuration with a one-cycle delay element — a register external to the network — suffices to provide communication from one virtual processor to the next.

This approach would suggest that we handle, for example, strided memory access with an adder connected to a fixed value and its own previous output, which is also routed to the memory read port, rather than an external register that increments by the stride each cycle.

Bit-serial variants

If you make the nodes bit-serial instead of parallel, you lose a factor of 16 in bandwidth. But you gain a factor of 256 (!!) in area, and you may also gain some bit-shifting power — suddenly a delay element is capable of shifting a word by a bit, as for a multiplication.

With this approach, it's still likely worthwhile to interleave virtual processors, but they're interleaved on a bit-by-bit basis rather than a word-by-word basis. The replacement for a 16-bit register in a

normal processor is, rather than wiring an output directly around to an input, a longish shift register from an output back to an input. (If the network latency is 7 bits, for example, you need a 112-bit shift register.) A delay element to shift a word by a bit could similarly be a shift register. (A 7-bit shift register if, again, your network latency is 7 bits.) But you don't need one — you can just use a path through the network! Doing the same thing for a 16-bit value would use up 16 inputs and outputs, though.

An interesting thing about bit-serial processors is that they more comfortably accommodate processing data of arbitrary-length words than data of fixed-length words. Dropping the carry from an addition after 16 bits or whatever is an extra exceptional case to add, rather than something you have to take special pains to avoid. Also, you get perhaps some extra use out of the elements used for addition — if you route a 0 carry to a bitwise adder, it gives you simultaneously XOR and (perhaps integrally delayed) AND, which you can use for other things.

Per-bit LUT

One possible interesting operation for the bit-serial case is demuxing: given four or eight bitstreams, demultiplex the values of two or three other bitstreams to select among them. This provides a programmable universal Boolean function which can, if desired, vary bit by bit.

Very crude performance analysis

The idea here is that you load the inner loop of your algorithm into the configuration of the network and just let it rip for a while. Under such circumstances, modern high-end CPUs can generally take advantage of SIMD operations, carrying out something like 128 bit operations per cycle. Small microcontrollers are stuck at something like one 32-bit payload operation every five cycles or so, so something like six bit operations per clock cycle.

If you have a 256-input 256-output bit-serial network that is 7 bits deep, you might allocate half of its lines to ALUish things, half a dozen or so to memory, and split the other half between long shift registers and simple loopbacks. This gives you a max of 128 bit operations per cycle, but maybe 64 is a more likely estimate.

Let's estimate that each D flip-flop uses 6 transistors and that we have 56 bits of FIFO on each of the 64 long-shift-register outputs, making them more akin to 8-bit registers. (It's easy enough to chain two of them together when you want a 16-bit register.) This gives us 21504 transistors in this register file, which is a lot but not the majority of the whole chip. The 1792 D flip-flops in the network itself are just 10752 transistors, and the 1792 corresponding MUXes might be another 10752 transistors. Then, on our 128 or so ALU lines, we have, say, 5 NAND gates each, each with 4 transistors, for a total of 2560 more transistors. The total, then, is 45568 known transistors, and if we add 50% or so for stuff I'm not thinking of, it's similar to the count on a 68000 or other mid-80s workstation CPU — but clockable at very high speeds because of its very short path length, and doing 128 bit operations per cycle instead of 6 or whatever.

However, that doesn't account for the control unit, the thing that

decides when and how to reconfigure the network and whatnot.

Topics

Keyboard-powered computers

Kragen Javier Sitaker, 2014-10-25 (updated 2018-10-28) (26 minutes)

(See also Low-power microcontrollers for a low-power computer (p. 2602).)

The energy available from keystrokes on a regular keyboard is almost 10 milliwatts: plenty to power a portable electronic word processor, so you could write a novel on one without an external power source. However, so far, energy harvesting keyboards are probably too inefficient to make this practical.

XKCD What if? 102 characterizes the energy produced by tapping keyboard keys; it cites Nagurka et al. 1999 as providing a smallish estimate:

> Using data from a study of rubber-dome keyboards—the most common type these days—… the energy required to press a key is around 1.5 millijoules for a letter key and 2.5 for a big key like the enter key or spacebar.

Summary of results

A keystroke adds, usually, a single letter or space to your novel. It provides 1300 microjoules of energy, of which 2.25 microjoules are needed by an LPC1100-family CPU to run word-processing software, 25 microjoules are needed by the E-Paper display to display the letter on the screen, and 0.5 microjoules are needed to eventually store the letter permanently in a Flash memory if you don't erase it, for a total of 27.75 microjoules out of the 1300 available. This means you need at least 2% efficient conversion of the keystroke energy, which commercially available pushbutton energy harvesting devices have barely achieved (they claim 2–5%.)

However, solar power or pullstring dynamos are probably better options. Typing at 90 wpm provides 10 mW of power, but only while you're typing, and that's 0.2–0.5 mW after conversion inefficiency; a 30mm-square solar panel provides 10 mW all the time.

Keystroke energy estimation

Nagurka et al.'s estimate seemed a little high to me: if for each keystroke your 50-gram fingers have to travel a 4mm key travel in, say, half a second, and then hit the bottom and stop, then the kinetic energy of your finger is closer to a microjoule per keystroke than a millijoule:

```
user@debian:~/devel/hojas$ units
2526 units, 72 prefixes, 56 nonlinear units

You have: 50 g * (4mm / 0.5 s)^2/2
You want: mJ
        * 0.0016
        / 625
```

A more recent open-access article by the same author explains in more detail. The conclusions of the paper are not very sensitive to how fast your fingers move, because the paper has graphs of the

measurements of the spring force of the rubber-dome keys, which is much larger than what is necessary to decelerate your finger and is not particularly sensitive to the speed at which the key is pressed: for 3mm of the 3.5mm travel of the "K" key in one keyboard he tests, for example, the force is between 0.4 newtons and 0.55 newtons regardless of whether the key is moving 0.5 mm/s or 80 mm/s, with the speed amounting to about 0.1 newtons of difference, i.e. about 25%.

0.45 newtons times 3mm is 1.35 millijoules.

0.45 newtons by itself would decelerate a 50-gram finger at 9 m/s/s, which is to say that it's just about the weight of the finger. If your finger were moving at 60mm/s without you applying continuous force to it, it would come to a stop in just under 7ms at 0.45 newtons. So the vast majority of the energy transmitted from your finger to the key is never part of the kinetic energy of your finger.

(Most of this energy is returned to your finger when the key comes back up, so it might be more difficult to type on a keyboard that harvests this energy. Experience with manual typewriters suggests that this problem will be minimal.)

Conservatively we might want to disregard the part of the energy estimate that's due to Nagurka and Marklin's model of how fast your fingers move, because he assumes that your keystrokes don't overlap, which they clearly do. But the difference, as you can see above, is small.

Even with this well-substantiated estimate, which is much higher than what I'd come up with, Randall goes on to calculate that this isn't a very useful amount of energy for things like running a modern laptop or microwaving a burrito.

At 90 words per minute, we have almost 10mW of available power:

```
You have: 90*5*1.3mJ/minute
You want: mW
        * 9.75
        / 0.1025641
```

Keyboard-powered computers

The original question, though, was this:

> As a writer, I'm wondering what would be the cumulative energy of the hundreds of thousands of keystrokes required to write a novel.
>
> —Nicolas Dickner

Perhaps Nicolas was actually wondering, in particular, whether you could run a computer to write the novel on with the energy from the keystrokes, so that he wouldn't have to worry about losing his battery charger or wearing out his battery and being unable to write.

In some sense the answer is clearly yes: people wrote novels on mechanical typewriters whose data storage mechanism was ink stamped onto paper with a mechanism powered by pressing the keys, although they were somewhat more effort to press than modern keyboard keys. Intuitively it seems like electronics ought to allow us to do *better*.

So how much power do you really need? Perhaps laptops are an

energy-inefficient way to write novels, compared to other kinds of portable computers. And in fact it turns out that running an electronic word processor on keyboard power is easily feasible, although I don't know of anyone who's done it.

CPU power consumption

CPU power consumption is not a problem, unless your CPU doesn't have a working sleep mode. Summary:

	nJ / instruction	instructions/keystroke
feasibility limit	170	7500
MSP430	0.9	1400000
PIC24	2?	600000
1990s StrongARM	1	1300000
LPC1110	0.3	4300000
Pentium	10	130000
STM32L0	0.23	5700000

Details

You can write a novel using a Commodore 64 or Apple][, and several people did. They can update text on the screen with relatively complex word-processing logic while keeping up with your typing speed, even at high typing speeds like 160 words per minute. These were 8-bit computers based on the MOSTek 6502 microprocessor and its variant the 6510, which typically run about 200 000 instructions per second. Because those instructions only manipulated 8 bits of data at a time, you usually need about twice as many of them as on a modern 32-bit microprocessor to do the same work. (In some cases you need more like 8 times as many, and in other cases you need the same number or even less, but those cases are rare.)

(Other 8-bit home computers of the time period, like the Nintendo NES, the Altair and all the CP/M computers that followed it, the Atari 2600, the various TRS-80s, and the Pac-Man arcade machine, were similar in speed. The IBM PC was maybe three to five times faster.)

So if we divide 100 000 32-bit instructions per second by 160 words per minute, we get a number of instructions per keystroke that is known to be sufficient: about 7500 instructions per keystroke.

So can one keystroke, in the microjoule to millijoule range, produce enough energy for 7500 32-bit CPU instructions? That would require power consumption of less than about 170 nanojoules per instruction.

The most common low-power microcontroller family today is the TI MSP430. It's a 16-bit microcontroller, and TI recently published a white paper on its power consumption which shows two members of this family running at a million instructions per second on 300 and 515 microamps on a 3-volt power supply; 300 microamps times 3 volts gives us 900 microwatts, and dividing 900 microwatts / 1MIPS gives us 0.9 nanojoules per instruction.

That is, for the MSP430F2001 described in the white paper, a keystroke provides almost 200 times as much power as would be needed to run a word processor to handle that keystroke.

Dividing it another way, a 1.3-millijoule keystroke provides enough

power to run the MSP430 at 1 MIPS (five times as fast as a Commodore 64) for 1.4 seconds.

The point of the whitepaper is to compare the MSP430 favorably to a PIC24 microcontroller from another chip vendor, which (according to TI) needs two or three times more power: they estimate that the PIC could run off a particular coin cell with a 1% duty cycle for less than two years without recharging, while the MSPs could last four to six years.

Academic papers report the fabrication of 10pJ/instruction and 2.6pJ/instruction microcontrollers, which is another two orders of magnitude better than the MSP430's 900 pJ/insn. Even as far back as the 20th century, the DEC StrongARM SA1110 used one nanojoule per instruction, and there were 22-picojoule-per-instruction microcontrollers reported and 1-picojoule-per-instruction ones being designed.

Zhai also published https://web.eecs.umich.edu/~taustin/papers/TVLSI09-subliminal.pdo of on the 2.6pJ processor.

The most interesting microcontroller to me right now is the 32-bit Philips/NXP LPC1110 family, which can apparently run at 48MHz (and 48MIPS) at 1.8 volts and just under 8 milliamps. Dividing, that's 0.3 nanojoules per instruction, three times better than the MSP430! However, the chip suffers about a 6 microamp leakage current in deep-sleep mode at room temperature, so each keystroke only provides enough energy to keep it in deep sleep mode for about 20 minutes. The LPC1110L variant cuts this sleeping energy consumption by about a factor of three.

So keystrokes provide about 600 times as much energy as you would need to power an LPC1110-family microcontroller running a word processor. You could write your word processor in BASIC or some other interpreted language and it would still work fine.

More recently (I think 2015), STMicroelectronics has released the STM32L011x3/4, based on the Cortex-M0+ core, which I think is the same one in the LPC1110. According to their measurements, it can run at 16MHz on 1.95 mA at anywhere from 1.65 V to 3.6 V, running code from RAM with the flash switched off; they claim 0.95 DMIPS/MHz. If we assume 1.8 V, this is 230 pJ/instruction, which is about 25% less than the LPC1110. This is according to the datasheet, "DocID027973 Rev 5".

An interesting feature is that the STM32L011x3/4 has a "low-power run" mode which clocks the CPU at 131 kHz at 31–120 µW (10–40 seconds per 1.3 mJ keystroke), as opposed to its normal 16MHz 3.5 mW run mode (370 ms per keystroke) and its 0.3–1 µW stop and standby modes (20–60 minutes per keystroke). This suggests that maybe you could continue doing less-CPU-demanding operations continuously for some time at a lower speed, without having to pay for sleep and wakeup. Unfortunately, the 7500 instructions guessed at above would amount to about 57 milliseconds at this speed, a noticeable lag.

Even non-low-power CPUs are efficient enough that your keystrokes are enough to power a word processor: a New Scientist article from 2006 explains that the Pentium from 1993, Pentium Ms from 2003, and Core Duos from 2006 all use 10 to 13 nanojoules per 32-bit instruction, which is still about 15 times as efficient as you'd

need to be able to power a word processor from them. The problem they have is that you can't put them to sleep and wake them up fast enough, so you couldn't actually use them for this.

Power consumption of other parts of the system

But a CPU isn't enough, by itself, to be a word processor. You also need some way to display the text while you're editing it and to store it when the machine runs out of stored energy, two functions fulfilled by the ink on the paper of a mechanical typewriter.

You may also want external RAM, since e.g. the LPC1110 family tops out at 8 kilobytes of RAM, which isn't much text, even compared to what you could keep in RAM on a Commodore 64. A small SRAM is probably a better bet than power-hungry DRAM, and you need to make sure to save it to stable storage before power runs out.

Display

The display is going to use more power than the rest of the system put together.

Probably the best current option for low-power text *display* is an E-Ink display, like the one used by the Amazon Swindle. E-Ink can maintain the same display for months or years without applied energy (as evidenced by broken E-Ink screens). A 6" E-Ink display needs about 750 mW during screen update, which takes 120ms, and that's for 122×91 mm at 167dpi. Breaking that down, that's about 190 nanojoules per pixel update, or 8100 nanojoules per updated square millimeter:

```
You have: 750 mW * 120ms / (122 mm * 91 mm * (167/inch)^2)
You want: nanojoules
        * 187.53217
        / 0.0053324185
You have: 750 mW * 120ms / (122 mm * 91 mm)
You want: nanojoules/mm^2
        * 8106.6475
        / 0.00012335556
```

Suppose you want to be able to display eight 20-em lines of 7-point text on such a display, similar to a TRS-80 Model 100, since that was a portable device that millions of reporters and other writers used on a daily basis in its day. (It sold six million units.) That's 7 points * 8 vertically and 7 points * 20 horizontally: 49mm×20mm, similar in size to a cellphone display. Updating the entire display then will require almost 8 millijoules:

```
You have: (7 points)^2 * 8 * 20 * 750 mW * 120ms / (122 mm * 91 mm)
You want: millijoules
        * 7.909696
        / 0.12642711
```

This means that, although you can do huge quantities of computation per keystroke, you can't afford to update the screen on every keystroke; you need about six keystrokes to provide enough

power for a full screen update. You'll have to make do with updating a small part of the screen for incremental updates, and save full-screen repaints for things like scrolling.

8 20-em lines of text is about 50 words (1 word ≈ 6 ens = 3 ems ≈ 3.2 ems = 160 ems/50) so 350 words per minute (a normal reading rate) works out to 7 screen updates per minute, or just under a milliwatt (1 mW).

(Calculating per character: that's 250 characters in 8 lines, or 31¼ characters per line, or 1.58 mm per character, or 3.9 mm² per character; this ends up being 31.6 μJ per character.)

Alternatively you could just make the keys six times as hard to press as on a regular keyboard, plus a tiny bit more to power the rest of the system. This doesn't sound like a great idea though.

(You can probably use a larger E-Ink display and keep your power consumption down by updating only small parts of it.)

You can't just cut a chunk out of a large E-Ink display; a small E-Ink display is Seeed Studio's 2.7-inch e-Paper panel, which is 70×46 millimeters and 264x176 pixels for US$26, and can be driven by a US$24 Arduino shield. That's 18 28-em lines of 7-point text, about 200 words, three times the Model-100-like display I suggested above.

Seeed's site suggests that this hardware needs 3.3 volts, 40mA, and 3 seconds to refresh the whole screen. This is unfortunately 50 times worse than the figures I got from the other site, totaling 400 millijoules for a full display redraw:

```
You have: 3.3 volts * 40 mA * 3 seconds
You want: millijoules
        * 396
        / 0.0025252525
```

You'd think another possible low-power display technology would be Pixel Qi "low-power" transflective/reflective LCD developed for the OLPC XO laptop. Their PQ 3Qi-01 is a 235 mm × 143 mm 185 g 1024×600 display, each pixel containing three transmissive subpixels and three grayscale reflective subpixels. However, its power supply current is specified as 135–228 mA at 3.3 V, depending on color (white uses 30% more power), even aside for its 1.5 amp initial startup current and the 83–525 mA LED backlight current. Even the lowest power mode, 30 fps reflective, is specified as 390–480 mW. So this device is way outside of our power budget. (All this is according to their datasheet, "Doc No.: PQ001-2", Sept 19, 2011.)

But those high-end LCDs aren't the only possible option. Old cellphones used supertwist nematic ("STN") LCD displays, which should be less power-hungry. In fact, I think that pixels that are turned off on these displays don't use any power. The Philips PCF8833 datasheet says it can refresh a 132×132×3 display at 35.8 Hz to 227 Hz; it seems to contemplate driving an LCD at 10 V and up to 10 mA but more normally 400 μA, but no more than 5 kΩ for each of its 132 output rows, only one of which is active at a time, which would seem to mean 2 mA. So it's designed to drive a load of somewhere between 4 mW and 100 mW but probably something like 20 mW. The chip itself consumes about 1 mA while accessing memory, which works out to be something like 3 mW.

So, at a rough guess, a 132×132×3 STN display without a backlight might use 7 mW continuously, which is equivalent to the power cost of about 37000 E-Ink pixel updates per second (at 190 nJ/pixel as above), which means that for motion above 2 fps the STN display is more efficient, but for slightly changing text, it's dramatically less efficient.

Flash or FRAM

(Summary: Flash is adequate.)

You also need to be able to store data permanently, so that your novel doesn't get lost when you stop pressing keys for a while. The most reasonable electronic means for that is probably Flash, although MRAM, PCRAM, and FeRAM ("FRAM") and I think ReRAM and CBRAM are currently available too and might be reasonable alternatives.

Some random PowerPoint presentation I found on the web says Flash uses 2µJ per 32-bit write and 150pJ per 32-bit read, while FeRAM uses 1nJ for either one. That means that a single keystroke provides enough energy to write 5 megabytes of FeRAM or 2600 bytes of Flash. Either memory technology is clearly sufficient for word processing from this point of view, but Flash is much cheaper.

(There are MSP430 microcontrollers that already use FeRAM ("FRAM") instead of Flash, but none of them have more than 128kiB, so none of them are big enough to avoid needing off-chip memory for a novel.)

Some other random PowerPoint presentation puts MRAM at 5nJ per "write energy", compared to 0.8nJ for SRAM, but I don't know if that's a bit, a byte, or a 32-bit word.

Abarrilado compared FRAM and MRAM chips and found that at 3V (?), the FRAM chip needed 5mA to write at 33MHz and 9 µA in standby, while the MRAM chip needed 23mA to write at 40MHz and 7 µA in standby. I think those bus cycles are mostly being used to write one bit each, which would give us 0.45 nJ/bit or 14nJ/32-bit to write the FRAM, or 1.7nJ/bit or 55 nJ/32-bit to write the MRAM. However, in both cases, he had to pay US$5 for about 32 kilobytes of storage, which means enough storage for a novel will be expensive.

The reason people are adopting FeRAM for embedded designs is that the write latency is lower, which lets the microcontroller go back to deep-sleep much sooner, which cuts power consumption. (Also, it's rad-hard and harder to reverse-engineer, which we don't care about here.) However, in this application, we're not under such severe constraints of available energy. We're worried about milliwatts, not microwatts and nanowatts.

Pullstrings

Another alternative to keyboard power and solar energy is a dynamo. 500 mm of pull at 50 N (about 5 kg of weight) should be within the capacity of most biological humans, and that's 25 J; a 50%-efficient dynamo would reduce it to 12½ J. At 10 mW (the power of typing at 90 wpm, disregarding potential conversion inefficiencies) those 12½ J are 21 minutes. If you need 30 µJ of energy to handle each keystroke (3 µJ to run 10000 instructions on a 300 pJ CPU, 25 µJ to update the E-Ink display, 0.5 µJ to store it in Flash, rounding up) then this is enough energy to handle 420 000 keystrokes.

Related work

enOcean has a commercial line of energy-harvesting pushbuttons, such as their ECO 200, and radio transmitters powered by them, such as the PTM 200. The ECO 200 consumes 2.7 to 3.9 newtons over 1.2 millimeters and produces 120 to 210 microjoules, which is about 2–5% efficiency.

Cherry, the well-known keyboard switch company, has a similar product.

Rashi Tiwari under Ephrahim Garcia at Cornell added an energy-harvesting device to a regular keyboard's Enter key and used it to flash some LEDs, I think in 2012, although they haven't published a paper on this, just a YouTube video.

Also, T Wacharasindhu and J W Kwon 2008 J. Micromech. Microeng. 18 104016 doi:10.1088/0960-1317/18/10/104016 is "A micromachined energy harvester from a keyboard using combined electromagnetic and piezoelectric conversion." They got 42 microwatts out of the setup, 95% of it from piezoelectric conversion. 42μW/10mW is 0.4% efficiency.

http://responsive.media.mit.edu/wp-content/uploads/sites/5/2013/10o2/A-Compact-Wireless-Self-Powered-Pushbutton-Controller.pdf got 500 μJ per Scripto "Aim 'n Click" lighter click, but improved circuitry could improve this substantially.

Combining sensing with generation

Variable-inductance position sensors are well-known in the form of e.g. the linear variable differential transformer, which senses the differential voltage induced in two counterbalanced secondary windings by a pulse or ac current through a primary between them to precisely measure the linear position of a high-permeability core. In this case, though, we don't care about precisely measuring its position; it's adequate to distinguish presence from absence.

If we additionally add a permanent magnet to the mechanism, every movement of the core will generate a voltage, even without a current through the primary. We can distinguish that from the higher-frequency sensing current because the core will never move significantly within 100 ns. So we can use a capacitive coupling to couple the fast pulses from the sense windings through to the sensing pins on the microcontroller, while rectifying the much slower ac signals with a Schottky diode or four.

(Alternatively, maybe the probe lines should be the ones the power is harvested from.)

No separate spring is needed, not even a rubber dome, if the magnet itself acts to return the key to the up position.

If we have, for example, a keyboard matrix of six probe lines by six sense lines (for 36 total keys), each wire runs through six coils. I think this means that the inductance of the other five coils will prevent the current from rising quickly, but not the voltage.

If the key makes a full magnetic circuit when in contact with the magnet, it should be easy to sense the initial voltage rise, since the circuit reluctance will rise and fall precipitously as the key moves through the 100 μm closest to the magnet, with a correspondingly

rapid change in flux density and thus a correspondingly large voltage.

The probe pulse needs to induce a strong enough current to charge the sense pin and other parasitic capacitances. If we assume that this is around 50 pF and we need to charge it to 1.2 volts to be sure of detection, this requires 36 pJ of energy; sending such a sense pulse 256 times per second on each of 6 lines requires 55 nW. So the power to probe the keyboard is minimal. However, this doesn't account for the fact that the microcontroller needs to wake up in this case! Somehow we need to wake up the microcontroller to generate the sense pulse.

Topics

Cached SOA desktop

Kragen Javier Sitaker, 2017-08-03 (updated 2018-10-26) (6 minutes)

(See also Window systems (p. 1335).)

I was thinking a lot about RESTish ways of doing stuff with caches and whatnot, and I was thinking that maybe you could do a desktop environment consisting of background images, alpha blending, text, text layout, windows, currying, and fonts, using redirects and caches.

The main idea is that, each frame (60 times a second), the window system virtually sends many requests (see What's wrong with CoAP (p. 560)) to the root window, one for each 16×16 pixel tile (see Caching screen contents (p. 2362)). This works out to 120×68 tiles on my 1920×1080 screen, or 8160 requests, for a total of 490,000 requests per second. The root window largely sends back temporary redirects: for subwindows they are redirects to requests for those subwindows, for background images they are redirects to those background images, and for text layouts they are redirects to those text layouts. In cases where more than one kind of thing participates in the tile, the window sends out the requests and takes responsibility for compositing them.

This is a rather large number of requests; 0MQ on my laptop with one single-threaded PUSH process and one single-threaded PULL process can only handle about 2.5 million (empty) messages per second. But the idea is that, almost all of the time, those messages aren't being sent at all; they're satisfied from a local cache.

There isn't a single unified cache, but rather a somewhat distributed caching system, using an asynchronous cache invalidation protocol. Each potentially cached request carries with it an identifier for the cache invalidation endpoint; a stateless responder can merely forward this identifier on to the resources it's fetching to compute the response, while a stateful one should generally store it for cache invalidation purposes.

A window with text in it can query the text layout to find out where to put the text (and which text is visible) and query the font to get the individual glyphs to display, then compositing them onto the background in the appropriate places.

The distributed nature of the caching system allows the display to apply an outdated-is-okay policy of using cached window contents if it hasn't received an up-to-date response by frame render time, or perhaps graying them out a bit or something, without globally applying such an outdated-is-okay policy or having a switch in the protocol for it or something, or having the cache first send an outdated response and then the real response, or whatever. Also, the display server can use a specialized cache structure — for example, instead of caching separate responses to the different tile requests, it can cache a single screen bitmap and maintain a tile validity bitmap (of only 8160 bits), and maybe even use a specialized invalidation handling policy of requesting tiles early. Downsides include cache duplication (many windows requesting the same glyph will cache it many times) and inability to globally optimize cache usage to minimize CPU time or improve responsiveness.

A cached redirected response can be invalidated because any of the

resources in the redirect chain was invalidated, including the final response. Using a redirect rather than just proxying the request allows the (potentially large) response to avoid an extra copy, or potentially many extra copies.

Currying a resource-with-arguments to be just like a normal resource allows requests and cache keys to stay small.

The requests and responses can be pipelined between a relatively small number of processes to amortize system call overhead over large numbers of small messages. Given potentially many outstanding requests, it may be reasonable to process them (on a given core) in an order determined by which piece of code is responsible for processing them.

I don't know, this seems like it might be a bit of mental masturbation in the sense that we already have working ways to build window systems. The main problem they have is that the window systems (and our operating systems) do a shitty job of guaranteeing responsivity to the user, and probably the way to solve that problem has more to do with avoiding swapping, prioritizing real-time interaction (like providing feedback to clicks and typing) over background tasks (like font rendering and relayouts), and having the whole user interface written in a way that guarantees finite and deterministic memory usage and CPU time — static allocation of memory and CPU time rather than dynamic, really. There are only so many pixels on the display, after all, and on a 60Hz video card you can't update them any sooner than 16⅔ ms in the worst case. That gives you 8.04 ns to decide what color to make each pixel. Not a lot of time to copy pixels around in messages!

I think there is something interesting in the idea of caching redirects to curried functions, though — maybe applicable in systems that don't involve IPC, too. If you can delegate a particular call to a chain of particular other functions (perhaps with added arguments) and cache that delegation in a way that doesn't involve running through the whole chain of functions again because the last function in the chain has an input change, then you can do efficient OO method dispatch in just that way, as well as things like delegating painting a particular part of a window to a particular widget. Maybe you could handle it like backtracking: save a series of backtrack points, and handle invalidation notifications by resetting the backtrack state for that cache entry to point to one of those earlier backtrack points, which could literally be an address to jump to in a piece of code. Maybe?

Topics

La vibración del hierro, ¿es de baja frecuencia o qué?

Kragen Javier Sitaker, 2016-10-07 (3 minutes)

https://www.facebook.com/jmpoio/posts/10154629260364445?comment_id=10154629631144445&comment_tracking=%7B%22tn%22%3A%22R%22%7D

Depende de comparado con qué.

Desde el siglo 17 hemos realizado varios avances en la alquimia y ahora podemos calcular con algo de precisión la frecuencia de vibración de los átomos de cada sustancia.

La frecuencia de vibración de los átomos de un sólido a la temperatura ambiente es alrededor de 10^{12} Hz, es decir, un terahertz. Esa es mucho más alta que la mayoría de las vibraciones que encontramos en la vida cotidiana, como los de sonido (20–10000 Hz), radio (20 Hz – 200 GHz), pero más baja que la frecuencia de luz (400–700 THz), rayos X, o rayos gama. En particular, los átomos de hierro, con su peso atómico alrededor de 56 amu, vibran más rápido que los átomos de cobre (63), oro (199), plata (108), o estaño (119), pero menos rápido que los átomos de aluminio (27), suponiendo que todos los metales en cuestión están a la misma temperatura.

Para el hierro en particular, su radio atómico es de unos 126 picómetros. Si supongamos que tiene tres grados de libertad translacionales que contienen la mayoría de su energía térmica, cada uno tendrá una energía de kT/6, que resulta ser aproximadamente $6{,}7 \times 10^{-22}$ J a 20°C; usando la relación oculta $E = ½mv^2$ descubierto por el alquimista Newton con $E \approx 6{,}7 \times 10^{-22}$ J y con una masa de 56 amu, entonces la velocidad de cada átomo está alrededor de 120 m/s en cada eje, o 210 m/s en total (ésto siendo el promedio de una distribución estadística Maxwell-Boltzmann, creo). A esa velocidad, el núcleo puede realizar un desplazamiento de, por ejemplo, 50 picómetros, en unos 0,24 picosegundos, implicando una frecuencia de vibración de alrededor de 4 terahertz.

Se puede lograr un cálculo más preciso con la ley universal de oscilación harmónica, basado en el trabajo por el mismo alquimista Newton en su investigación del movimiento de los planetas (como arriba, así abajo; como afuera, así adentro!) — en la simbología que usamos hoy en día, $md^2x/dt^2 = -kx$, cuyo sentido solo estará aparente a los que cultivan cuidadosamente su ojo interno con atención diligente al sendero filosófico. Pero confío que el número aproximado de 4 terahertz será suficientemente preciso para tus propósitos.

Lamentablemente, tanto hoy en día que hace veinte generaciones, hay muchos falsos filósofos que opinan sobre cualquier cosa sin saber nada al respecto, dejándose divagar por diez mil maestros falsos que no se dedican al conocimiento verdadero sino a estafar a la gente, porque la gente sigue buscando un sendero fácil al conocimiento, y así terminan repitiendo las palabras de las falsas maestras sin entender nada. El sendero fácil no existe, pero el sendero verdadero está disponible para todos y todas. Así seguimos ignorantes generación tras

generación, muriéndonos de sed ante un río de conocimiento hermético, y destruyendo la vida en la tierra por nuestra ignorancia.

Es una verdadera tragedia.

Topics

A design sketch of an air conditioner powered by solar thermal power

Kragen Javier Sitaker, 2016-12-22 (updated 2017-01-04) (29 minutes)

After reflecting on my solar dehumidifier idea (from August 2016 or maybe earlier) and the current air-conditioner-less state of my new bedroom, I did some calculations on making a desiccant-based air conditioner powered by the sun.

The basic idea is that you cool air by evaporating water, like a swamp cooler. But because swamp coolers work poorly or not at all when the air is already humid, you dehumidify the air before cooling it. And because the high-humidity output is undesirable in climates that aren't dry, you transfer the cool through a recuperative heat exchanger without transferring the humidity; this is known as "indirect evaporative cooling".

Standard swamp coolers use a recirculating pump to dribble water over wood wool ("excelsior") pads to evaporate it, but a mister might work just as well, and might reduce the available habitat for bacteria. WP says a high-pressure pump and 5-micron mist orifices are effective.

To dehumidify the air, you pass it over a desiccant, a hygroscopic substance which absorbs some of the moisture from the air — potentially nearly all of it, depending on the desiccant and how dry it is. The desiccant will eventually be saturated with moisture and stop absorbing it from the air; at this point it needs to be regenerated, normally by heating, normally by running hot air through it, then dumping the hot, moist air into the environment.

This design uses a large quantity of a cheap, inefficient desiccant, such as wood wool, rather than a more reasonable quantity of a more mainstream desiccant, such as calcium chloride. As a result, most of the heat from the regeneration air is deposited as sensible heat in the desiccant; only a fraction is used to drive out the water. In order to prevent this heat from leaking into the intake air to be desiccated, two more air circuits are needed: a closed circuit of dry air to remove the heat from the desiccant, and an open circuit of hot dry air to remove the heat from the closed circuit.

So there are five separate, isolated air circuits: a cool wet circuit running through the desiccant, the evaporator, and a heat exchanger; a cool dry circuit running through that heat exchanger; a hot wet circuit running through a solar heater and used to regenerate the desiccant; a closed dry circuit (half hot, half cold) to cool the desiccant; and a hot dry circuit to cool the air in the closed dry circuit down to ambient, transferring the heat to the outside air. Only the cool dry circuit connects to the indoor air.

Swamp coolers typically achieve 70% to 90% relative humidity on their output side; I think it should be possible for relatively simple desiccants (wood wool itself, for example) to get the relative humidity of the air going into the evaporator below 20% see calculations below), but I haven't performed experiments to show this

yet. As long as it's below 20%, we have a minimum swing of 50% relative humidity, which achieves half of the total theoretically achievable temperature swing. WP says, "Most efficient systems can lower the dry air temperature to 95% of the wet-bulb temperature, the least efficient systems only achieve 50%."

I am estimating that my new bedroom might receive 4 kilowatts of solar heat during the day (3400 frigories per hour, a bit over a "ton") that it might be necessary to extract. Converting this sensible heat adiabatically into latent heat by evaporating water consumes 1 g of water per 2260 J, which is to say (4000/2260) g/s of water: 1.8 g of water per second, or 6.4 kg (= 6.4 ℓ) per hour. And let's suppose that the desiccant can be recycled every 30 minutes.

Output airflow rates

Let's suppose that it is adequate to reduce the temperature of the humid air in the evaporation stage to 15° — indeed, the cool air output will probably need to be mixed with enough warm air to avoid being uncomfortably cold. At 15°, the vapor pressure of water is about 1.7 kPa or about .017 atm; at about a 18:28 molar mass ratio of air to water vapor, that works out to .0109 g of water per g of air; air's density of 1.2 g/ℓ means that our 1.8 g/s of water will need 165 g/s of air occupying 138 ℓ/s to carry it along.

But wait, how much cooling can we get out of .0109 g of water per g of air? Will it cool it enough? Air holds 1.005 kJ/kg/K of sensible heat. 2260 J/g * .0109 g/g / (1.005 kJ/kg/K) is about 24.5 kelvins of ΔT, so we should be fine as long as the incoming air is below 15° + 24.5K ≈ 40°. Checking the psychrometric chart, that seems about right; the 15° diagonal wet-bulb line hits the horizontal 0% RH line at a dry-bulb temperature of 41°. 40° or 41° is fine. It never gets that hot here.

Taking into account the finite efficiencies of evaporative coolers mentioned earlier, a 60%-efficient evaporative cooler (justified below) would only lower the air temperature by 14.7°, which gets us to 15° for temperatures of 29.7° and below, and even at 38° still gets us to a quite tolerable 23.3°.

Wood wool as a desiccant

Common high-capacity desiccants include Glauber's salt (sodium sulfate, mirabilite), plaster of Paris, silica gel, calcium chloride, and sodium bentonite. But many other hygroscopic substances could potentially be used, including Play-Doh, pot shards, coffee beans, cocoa husks, shredded PET bottles, wood wool, straw, or palo borracho fibers. But let's consider wood wool, since it's probably pretty typical of many of these substances, and relatively cheap.

Let's suppose that by heating wood wool or a similar cheap hygroscopic substance after saturation with outdoor air we can easily drive out 5% of its mass in water.

5% is a conservative estimate for wood, because dry wood normally has about 12% moisture content, says WP, compared to its mass after being oven-dried at 103° for 24 hours. The fiber saturation point, which wood in a 99% relative-humidity atmosphere, is typically 25% or 30% moisture content; equilibrium moisture content ranges from 0% at 0% relative humidity through that 25% or 30%, hitting about 7% moisture content at 40% relative humidity and 12% moisture

content at 60% relative humidity. (Buenos Aires summer usually ranges from 30% to 60% relative humidity.) Wood outdoors in Miami ends up with 13% to 15% moisture; in Phoenix, it's more like 5% to 10%.

Wood drying is normally a very slow process, taking days for hardwoods at tens of millimeters of thickness, roughly proportional to the 1.52th power of the wood thickness, according to the Simpson–Tschernitz wood drying model. At 500 microns (a typical thickness for wood wool), this should be several hundred times faster, needing only 15 to 30 minutes to come near equilibrium, or less for softwoods.

To me, the above implies that exposing air to a sufficiently large quantity of wood with, say, 7% moisture content, will result in reducing its relative humidity to 40%, like Phoenix, and if the incoming air has 60% relative humidity, the wood will stop removing water from it once the wood reaches 12% moisture content. So I think 5% moisture content swing is a reasonable estimate for wood. But can we do better? Like 10%?

Wood starts to char above about 200°, and sugar starts to caramelize at 170°. Let's say we heat our desiccant up to just 150° to give a good margin of safety. What will the air's relative humidity be? And what will it leave in the wood?

If we take some incoming air at 35° (a worst case scenario in Buenos Aires!) and 65% relative humidity, the psychrometric chart shows that it contains .022 g of water per g of air, I guess because water's vapor pressure at 35° is 5.63 kPa = .0556 atmospheres, 65% of that is .0361 atmospheres, and water vapor is only about 18/28 of nitrogen's density, which gives a fraction of .0232 grams per gram — close enough. If we then use a solar oven to heat it up to 150°, water's vapor pressure goes from 5.63 kPa to 476 kPa, so the relative humidity falls by the same factor of 84.5, to 0.77%. This should leave the wood's equilibrium moisture content at about 0.1%; the EMC chart in the WP page I linked above has wood at 120° reaching 5% EMC at 60% RH, a ratio of about .08 between the two percentages.

That is to say, heating wood up to 150° should be sufficient to remove over 99% of its moisture content from ordinary levels of air humidity once it reaches equilibrium, a process whose speed depends on the thickness of the wood but should only take a few minutes for the thicknesses I'm talking about here.

This means that we probably *can* get 10% of the wood's weight in water, and perhaps more importantly, we can use it to reduce the relative humidity of the air being desiccated down below 20% or so, at the expense of a lower moisture load in the wood. Indeed, using countercurrent regenerator principles — where the air being desiccated runs through progressively dryer and dryer wood — perhaps we can reduce its relative humidity down below 5% or even 1% in this way.

Wood has a specific heat typically between 2 and 3 kJ/kg/K, with white pine being in the middle at about 2.5. The energy used to heat up the wood to the drying temperature is essentially wasted, so it would be nice to minimize it; going from 20° to 150°, it's about 390 kJ/kg of wood, or about 3900 kJ per kg of moisture absorbed in the wood. Additionally, we need energy to evaporate the water, which is

the same 2260 J/g or 2260 kJ/kg of moisture, for a total of about 6200 kJ/kg of moisture removed by the dehumidifier.

You could consider this a fairly appallingly terrible level of efficiency, but solar thermal energy has been very cheap for the last several billion years, and will probably remain so until at least 2030. So we can afford to "waste" an enormous amount of such low-grade thermal energy.

The air output from the desiccant, at 20% humidity or less, then gets humidified by the evaporator to 80% or more, so we can get at least 60% of the theoretical 24.5K temperature reduction, as long as the output is at or above 15°.

Sizing the desiccant and the hot wet circuit

Above I've talked about a 30-minute cycle time, 65% input humidity (at up to 35°, thus involving up to .022 g/g of moisture), 20% output humidity from the desiccant, and 138 ℓ/s of airflow for the cool, wet air circuit. The amount of water absorbed in the desiccant should be somewhat less than the 1.8 g/s evaporated in the evaporator, because the input humidity is lower than the evaporator's output humidity. 0.022 g/g of water vapor in air of 1.2 g/ℓ is .0183 g/ℓ of water vapor, which, at 138 ℓ/s, works out to 2.5 g/s. XXX this is clearly wrong. Over 30 minutes, 2.5 g/s works out to 4.5 kg of moisture; our equilibrium moisture content is only 10%, since we're using an inefficient cheap desiccant, so that works out to 45 kg of desiccant.

This is clearly a manageable quantity of stuff to put on the roof of your bedroom — it's not 4500 kg or something clearly impractical like that — but at Lowe's they sell "excelsior blankets" for landscaping at [US$183 for 65 pounds, US$6.20/kg], so we're talking about almost US$300 of material.

2.5 g/s by 6200 kJ/kg gives us an estimate of how much thermal power we need: about 15.5 kW, which is about, say, 20 m² of sunlight, and about 4× the amount of thermal power we're trying to reject.

15.5 kW of thermal power at 150° with a ΔT of 130 K in air carrying 1.005 kJ/kg/K is 119 g/s of hot wet air, or 99 ℓ/s, a bit lower than the other circuits.

Output airflow

The 138 ℓ/s of cool wet air can be used to cool a somewhat larger quantity of cool dry air to, perhaps, a somewhat higher temperature. If the cool dry air temperature is to be 20°, you could perhaps have 200 ℓ/s or more of cool dry air.

If we compromise at, say, 160 ℓ/s, we still need to figure out how we move this air around, ideally somewhat quietly. If the ducting is a generous 400mm in diameter, this works out to about 1.3 m/s of air velocity, which is going to be somewhat noisy, but tolerable and easily achievable.

Desiccant cooling

The 3900 kJ/kg deposited as sensible heat to raise the desiccant from 20° to 150° must be removed somehow; otherwise it will heat the intake air rather than desiccating it. My friend Mina suggests

perhaps separating the desiccant from the outside air with only a very thin moisture barrier, such as aluminum flashing, so that it can absorb the ambient temperature in a rest period after being regenerated; if this is workable, it is clearly simpler than what I propose here, which I am certain is workable.

The desiccant passes through three stages: desiccation, regeneration, and cooling. The cooling stage passes cooling air in a closed circuit through it, heating the air; that air then passes through a recuperative heat exchanger to cool it so that it can return to cool the desiccant further.

This air is at, I think, an average temperature of 85°, thus transferring 65 K of ΔT, and holding 65 K · 1.005 kJ/kg/K = 65 kJ/kg of air. The sensible heat involved is 2.5 g/s · 3900 kJ/kg = 9.7 kW. Consequently this circuit must circulate 149 g/s of air, 124 ℓ/s; I think a similar quantity is needed on the hot dry circuit, although maybe it's twice as much air due to a lower ΔT.

Fans

160 ℓ/s is 340 cubic feet per minute. For comparison, a US$24 6" (152 mm) duct fan pumps 240 cfm on 37 watts and supposedly makes 68 dB of noise. This works out to US$0.21 and 330 mW per ℓ/s; the five air circuits of 138, 160, 99, 124, and 248 ℓ/s add up to 770 ℓ/s, US$160, and 250 W. This is less than 2% of the power needed to regenerate the desiccant. It could be reduced further by using wider ducts like the 400mm ducts I suggested above; perhaps it could be provided by solar heat as well, rather than by electricity, particularly since Buenos Aires is prone to power outages at the hottest times. At 1 m/s, 250 W is 250 N, the weight of about 25 kg, which gives some idea of how much work this device requires.

Some kind of wacked-out dude has made a solar Stirling engine of "probably about 100 to 150 W" or "an estimated 70–100 watts" at about 400 mm diameter, "from the 'Andy Ross Stirling' design," with the two cylinders at right angles ("V-position"). At a typical 25% Carnot efficiency and 800 W/m² insolation, 250 W would require a solar concentrator of some 1.25 m². It looks like the nutbag is using a Fresnel lens of about 1 m² for his somewhat smaller engine.

Freezing stuff by multistage water evaporation

At a minimum, even with a perfect system, to get to 0° with evaporative cooling, you need to start with dry air at 9°; to get to 9° with evaporative cooling, you need to start with dry air at 27°; but to get to 27° with evaporative cooling, you can start with dry air at any livable temperature whatsoever. So a cascade of two or three such desiccant-based coolers could cool food or whatever down below freezing.

Water consumption

6.4 ℓ/h is probably 30 ℓ/day, which is probably an acceptable cost. I can't think of a way to condense either the water from the desiccant or from the evaporator.

Feedback

There are a few process parameters that need to be controlled to keep the system working properly.

If the cool wet air circuit runs too fast, the desiccant could be overloaded, leading to air still moist as it enters the evaporator, and thus output air not as cool as it should be.

If the cool wet air circuit runs too slow, it may not be able to evaporate enough water — it will be the right temperature, but there won't be as much cool air as there should be.

If the evaporator water runs too slowly, output air will not be as cool as it should be.

If the evaporator water runs too quickly, it could accumulate. Swamp coolers normally use a float valve to prevent this.

If the hot wet air circuit runs too fast, it might not be as hot as it should be, and so the desiccant might be less regenerated than intended, leading to diminished cooling capacity; however, unless this is super extreme, the diminution should be minimal, because the drying process is an exponential decay already close to its asymptote at this point.

If the hot wet air circuit runs too slowly, it might be too hot, which could set fire to the desiccant or melt other parts of the machinery, maybe including the solar heater itself. This would be disastrous. Some kind of thermostatic control on the solar heater seems essential.

If the cool dry air circuit runs too fast, it won't be as cool as it could be, while if it runs too slow, some of the cool will escape up the chimney with cool wet air.

If the desiccant rotates too slowly, it will become saturated with moisture in the cool wet air circuit, diminishing cooling performance; if the regeneration air is properly temperature-controlled, there should be no further risk, but if the regeneration air is too hot, then the desiccant rotating quickly enough be the only thing preventing it from overheating.

If the desiccant rotates too quickly, assuming a countercurrent configuration, I think the only bad thing that will happen is that it will leak more heat from the hot wet circuit into the cool dry circuit, mildly degrading performance.

XXX what about the desiccant cooling circuits?

With electronics, of course, it's very simple to monitor and compensate for all of these problems. But one of the appealing benefits of this design is that, at least in theory, you should be able to construct and repair it with Stone Age materials and tools.

So it should be possible to make it work reliably under varying conditions with the following feedback and safety mechanisms:

- A float-valve mechanism to control the moisture in the evaporator (and maybe a recirculating pump);
- making the desiccant rotate considerably more quickly than the minimum necessary;
- ???

Perhaps of use is the fact that most species of flat grain wood will change size 1% for every 4% change in moisture content, but only crosswise, so you can make humidistats out of wood.

Desiccant rotation systems

The desiccant could be either in the form of a single, slowly rotating wheel, physically rotating through the three phases it needs, or it could be three or more separate pebble beds, controlled using valves.

A 1% prototype

Suppose we just want to see if the idea can work. It should be possible to scale it down:

- 40 W thermal cooling output;
- 1.4 ℓ/s of input and 15° cool wet air output;
- 1.6 ℓ/s of 20° cool dry air output;
- 990 mℓ/s of hot wet air output;
- 18 mg/s or 64 mℓ/h of water consumed and evaporated;
- 25 mg/s of water removed from input air and transferred to hot wet air output;
- 450 g of desiccant, costing US$3.
- 30-minute cycle time on the desiccant;
- 150 mm diameter air pipes.

wait, how much thermal input?

Alternative fuels

Any kind of fire, including a propane or wood fire, should serve to heat the air to regenerate the desiccant. With a coke or charcoal fire, you could pass the combustion exhaust directly through the desiccant, but other common kinds of fires probably produce too much water and other contaminants which could damage the desiccant in several ways: directly hydrating it, clogging pores in it, coating its surface with waterproof coatings, catalyzing pyrolysis at lower temperatures, and supporting fire. So this involves using another heat exchanger in place of the solar heater.

Alternative coolants

Air has some significant advantages as a coolant: it can withstand temperatures from cryogenic to red-hot, it isn't combustible, it's readily available anywhere in the world, it has low viscosity, and it's compatible with a wide variety of materials (at least at ordinary temperatures). But it also has very low density, which means that transferring significant heat flows with it requires very large pipes and large heat exchangers.

In this case, I don't think there is a reasonable alternative to air for any of the five circuits. The cold wet circuit and the hot wet circuit need water to evaporate into them, so they need to be air;

Alternative desiccants

Wood scraps, paper scraps, and such things may be a cheaper alternative to such freshly manufactured materials. In some places, soil containing a substantial amount of clay, or maybe organic matter, might be cheaper and work adequately. Watts, Bilanski, and Menzies show adsorption "isotherms" for various bentonite clays showing about 100–200 mg adsorbed water per gram of dried clay at 0.6 "relative vapour pressure", which I think means 60% relative humidity. (They never explain the terms in the paper.) US patent

4,254,565 from 1981 says that bentonite left out to dry in the sun for a few weeks to months typically ends up with 10% to 18% moisture content.

So a porous soil mixture that's half sodium bentonite and half, say, sand, might perform similarly to wood as a desiccant. It would also have has the safety advantage that it's not combustible.

Perhaps the second cheapest alternative, following dirt itself, is agricultural crop residues like straw or bagasse. It turns out there are a number of studies of equilibrium moisture content of straw, because moisture absorption is crucial for, among other things, straw-bale construction.

Kymäläinen and Pasila published an experiment in 2000 on flax and hemp fiber "moisture regain", typically 12% in flax or hemp at 21° and 65% RH, but only 7% in cotton, because most of the sorption is not from cellulose. They found curves quite similar to wood — 5% EMC at 15% RH, 10–15% EMC at 76% RH, and 25 or 30% EMC at 97% RH, depending on harvest time, all dry basis. They also say, "Pectins and hemicellulose absorb more moisture from the air than does cellulose," which sounds intriguing.

Farmers discussing online say you normally bale straw at 15% moisture content, though sometimes you can get away with 20% (at which point hay will rot, posing a fire risk, but straw may not), and if you leave it out long enough it gets down to 8–10%. Farmers won't buy hay with over 16% moisture. For construction, anything below 20% is OK.

Some anonymous Canadians at CMHC in maybe 1996 measured straw bale equilibrium moisture content and found a similar curve to that of wood: 5% EMC at 20% RH, 10% EMC around 50% RH, and 15% EMC around 80% RH, all dry basis.

Duggal and Muir measured EMC of wheat straw in 1981 and found 7–9% EMC at 35% RH, 7–12% EMC at 55% RH, and about 10%–14% EMC at 70% RH, all wet basis, all depending on temperature.

Taha Ashour et al. in 2010 measured EMC of some plasters used for straw-bale buildings, finding EMCs mostly in the 2%–4% range, even when the plasters were 75% reinforcing fibers made of wood or straw and 25% made of soil (which was 31% clay). The article doesn't mention the fact that these EMCs are like five times lower than the EMCs of the incorporated fibers, so I don't know if the author has any idea why this large discrepancy exists, or even if they know there is a discrepancy.

Heath and Walker in 2009 measured EMC of straw bale walls and found a fairly straight line from 2% EMC at 4% RH up to 13% EMC at 70% RH, followed by a sharp upward turn to 50% EMC (!!) at 93% RH, all dry basis.

Alfalfa hay bales currently cost AR$35 per 25-kg bale, which would reduce the desiccant cost from US$300 to AR$70, which is about US$5. (Actually there are some shipping costs which almost double the cost, but whatever.) Wheat straw bales cost AR$70.

Hay can spontaneously combust due to thermophilic bacteria if wet. At 200°F (93°) farmers are advised, "Most likely, a fire will occur.", but even as low as 160°F (71°) it can happen. So it's possible that straw, being in many ways similar to hay, might not withstand temperatures as high as 150° without catching fire.

Advantages over standard indirect evaporative cooling

You might reasonably ask why not simply use a normal cooling tower to cool some water (or oil or propylene glycol or whatever), then run that water through a heat exchanger in your house. That is a much simpler approach, and it also avoids raising the humidity in the house. However, more or less by definition, it can only cool the air down to the wet-bulb temperature of the outside air, and usually it can't quite make it that far. On an unpleasant summer day in Buenos Aires, when the temperature reaches 35° and the relative humidity 60%, the wet-bulb temperature is 27° — still unbearable. Worse, on simply cooling that air to 27° without dehumidifying it, you would raise its relative humidity to 90%, which is very unpleasant indeed. (Air conditioners generally reduce this problem by dehumidifying air by cooling it even further, then allowing it to mix. XXX maybe I should tackle that by a different approach)

Topics

Hipster stack 2017

Kragen Javier Sitaker, 2017-04-28 (updated 2017-05-04) (26 minutes)

What's the hipster stack in 2017? Or, maybe, what's the most powerful, highest-leverage software tool stack available? Here are my notes on trying to learn it.

Overview of the current hipster trends

I'm not totally sure, but I'm thinking that it's probably something like React (with JSX) for the frontend (with maybe Less or something for the CSS), Node for the server (with nginx in front of it), and SQLite for the database, though there seems to be a lot of fragmentation here, with lots of people using MariaDB, Redis, MongoDB, or whatever the hell. For administering the server, we use Docker (probably on Ubuntu) and Puppet and Kubernetes, and for an editor, we use Atom. When we're not using Docker, we virtualize our testing environment with Vagrant in VirtualBox. And we chat about it all on Slack.

StackOverflow trends

http://stackoverflow.com/tags?tab=popular has JS as the top language tag (followed by Java, C#, and PHP); Android as the top platform tag (followed by iOS); MySQL as the top database tag (followed by SQL and sql-server); etc. I did a SQL query to see what the current ones were at https://data.stackexchange.com/stackoverflow/query/664354/most-popular-stackoverflow-tags-in-2017 and got JS (followed by Java, Python, and PHP); Android (followed by iOS); jQuery as the top JS framework (followed by Angular, twice, and then React); MySQL as the top database (followed by SQL and sql-server); Excel as the most popular IDE (followed by XCode, Visual Studio, Android Studio, Matlab, and Eclipse); the top UI toolkit is HTML (followed by jQuery, CSS, Angular, and finally WPF at #60 and Qt at #96); etc. Firebase, Typescript, Pandas, Swift, AWS, Azure, Android Studio, and Spark also seem to be hot compared to previously. https://data.stackexchange.com/serverfault/query/664369/most-popular-serverfault-tags-in-2017 is the same query on ServerFault, from which we learn that nginx and AWS are hot.

The relatively most fashionable things this year are centos7, docker, apache-2.4, azure, windows-server-2012-r2, linux-networking, amazon-web-services, php-fpm, and reverse-proxy, each of which had more than twice the proportion of all stackoverflow tags this year as over the life of stackoverflow. However, nginx is far more popular than Apache now! And AWS is still more popular than Azure, and Ubuntu still more popular than CentOS.

Hacker News trends

The fashionable things on Hacker News seem to be, at the moment:

• Machine learning: 2 items on front page, 1 item in 30 newest, 2 comments in 30 newest

• Linux: 4 comments in 30 newest
• Computer vision: 1 item on front page, 2 comments in 30 newest
• Rust: 1 item on front page, 2 comments in 30 newest
• Python 3: 1 item on front page, 1 comment in 30 newest
• C++: 1 item on front page, 1 comment in 30 newest
• React: 2 items in 30 newest
• C: 2 comments in 30 newest
• Docker and similar containers: 1 item on front page
• Kubernetes: 1 item on front page
• Redis: 1 item on front page
• TypeScript: 1 item on front page
• Dataflow programming (Fathom, in this case): 1 item on front page
• Rails: 1 item in 30 newest
• Golang: 1 comment in 30 newest

I mean that's super crude. Note how different it is from the StackOverflow list!

Github trends

GitHub has a list of language implementations developed there. Swift has 38k stars and 5k forks; Golang has 27k stars and 3k forks; Rust has 21k stars and 4k forks; TypeScript has 21k stars and 3k forks; CoffeeScript still has 13k stars and 1k forks; Ruby has 12k stars and 3k forks; and PHP has 11k stars and 3k forks. Of course, this excludes languages whose implementation isn't GitHub-hosted and those with many implementations, like C++.

Similarly, they have a list of NoSQL databases hosted there, leading with Redis, RethinkDB, and MongoDB. Redis is actually more than twice as popular as Mongo. RethinkDB is probably dead. Their list of trending repositories this year includes Mastodon, which is built on Rails, Docker, Vagrant, Nginx, Redis, Postgres, and Node; a guide to learning bash; an IDE for React, React itself, react-bits (tips), and react-sketchapp; a repo of minimal examples of algorithms in Python; TensorFlow, a TensorFlow-based NN library, and Caffe; a medium.com clone built on React, Angular, Node, and Django (?); a Sendgrid clone built on Rails, MySQL, RabbitMQ (!), and Node called Postman; Vue.js; Atlassian's LocalStack, an AWS clone; a list of hot Mac apps (headed by Atom), and at #1, a "roadmap".

The Roadmap

The roadmap recommends learning, for a frontend developer, HTML, CSS, JS, ES6, npm scripts, Gulp for running tasks, Yarn, npm, TypeScript, some JS test framework (Jest or Mocha), webpack, Bootstrap, Sass, and some JS framework (Angular, React (in which case either Flux or Redux), or Vue.js); or, for a backend developer, one of Python, PHP, Node, Ruby, C#, Java, or Golang. For Python, they recommend pip, unittest, Django, and aiohttp (which is apparently an async/await thing built on Python 3's new (2013) asyncio library); for Node, they recommend npm, Yarn, Express, and Mocha. For going deeper on the backend, they recommend nginx, RESTful APIs, reading about MVC, authentication with OAuth 2.0 and JWT (JSON Web Token), "SOLID/YAGNI/KISS etc." (i.e. learning how to program), regexps, security, and Docker; deeper still, they recommend memcached and Redis for caching, Postgres, MariaDB, and MySQL as relational databases, and Redis and

MongoDB as nonrelational databases. To "up your game further", they suggest search engines (ideally ElasticSearch), GoF design patterns, architectural patterns, "give DDD a shot", and "learn different testing techniques".

There's an entirely separate roadmap for devops, recommending either Linux or Unix, AWS, automation (either CloudFormation, Puppet, Ansible, or Terraform), CI/CD (Jenkins, Travis, or TeamCity), monitoring and alerting (Nagios, PagerDuty, or AppDynamics), Docker (or maybe rkt), Apache and nginx, cluster managers (Kubernetes, Mesosphere, Mesos, Docker Swarm, or Nomad), loving the terminal, vim/nano, bash scripts, compiling apps from source, and a tree of commands:

text: awk sed grep sort uniq cat cut echo fmt tr nl egrep fgrep wc
ps: ps top htop atop
perf: nmon iostat sar vmstat
net: nmap tcpdump ping traceroute airmon airodump

Then there's a set of other things you should know: the OSI model (oddly subtitled TCP/IP/UDP), knowledge about different filesystems, setting up a reverse proxy, setting up a caching server, setting up a load balancer (HAProxy or nginx), setting up a firewall, TLS, STARTTLS, SSL, HTTPS, SSH, SCP, SFTP, and "postmortem analysis when something bad happens".

It seems like most people are using Macs on the desktop and Linux on the server.

Proggit

On Proggit, the top few items are mostly the same as on HN (last night), but not quite the same. Trendy topics include computer vision, machine learning, D, security (which is everywhere even though I haven't mentioned it previously), Docker and containerization, Clojure, Postman (the Sendgrid clone I mentioned above), a .NET thing called "Entity Framework", NixOS, Git, XAML, Heroku, AWS, Vue.js, vim, TLS, Python, Racket, AI, interviewing, Unicode, Spotify, Postgres, Rails, and OpenGL GLSL.

Leverage

The things that are fashionable are not necessarily the things that provide the most leverage. SQLite is a *lot* faster than MySQL, not to mention a lot easier to install and administer, for what it does. Java and Rust are a lot lower leverage for most things than Python or, often, PHP.

Slightly more in-depth overview of the top hipster technologies

Here's my subjective perception of the top 32 hipster technologies for 2017, some of which are actually old. Here's a paragraph about where each one seems to be in 2017:

- JS
- CSS
- nginx
- Firebase
- Angular

• Slack
• Reactive and dataflow programming
• Node.js
• Mobile-friendly web design
• Python
• Less
• MySQL and MariaDB
• Neural networks and other machine learning
• Android
• TensorFlow
• Golang
• Docker
• Linux (Ubuntu and systemd)
• Mocha (the testing framework)
• Bootstrap
• TLS/SSL
• AWS
• RabbitMQ and ZeroMQ
• TypeScript
• Redis
• Atom
• Swift
• JSON
• SQLite
• Chef
• Rust
• React

As a bonus, I'm adding:

• Jenkins

JavaScript is by far the dominant programming language, with performance close to that of C, but garbage collection and a flexible object model like Python or Smalltalk. A lot of server-side development is still being done in C++ (which still runs faster) or Golang, or still being done in Python (which is still more flexible and easier to read), but the gap seems to be closing rapidly, as more and more is done in Node.js. In web browsers, JS is still the only real runtime option, although WebAssembly is working to open the browser to a wider range of languages, and TypeScript is gaining popularity as a statically typed and therefore more maintainable version of JS. ECMAScript 6 is now broadly implemented, and its features seem like they close most of the usability gap that used to exist between JS and more ergonomic languages like Python.

CSS has accidentally become Turing-complete recently and may become sentient soon. Most people use preprocessors like Sass (or Less). There are conferences devoted entirely to CSS. But there still isn't a browser that supports both hyphenation and widow-and-orphan control for printing. Not satisfied with the original CSS box model and its variants, they have now stuffed three entirely separate layout models into CSS: standard, flexbox, and grid. CSS transforms allow pretty trippy (GPU-accelerated!) effects now. Also apparently CSS animation is a thing now.

Nginx is event-based, shuffling any kind of multiprocess management off to some backend process, communicating either via

FastCGI (or SCGI, or uWSGI for Python) or via HTTP (acting as a reverse proxy). This supports WebSockets a lot better than Apache's preforking MPM can, and it's the sensible way to structure things if you're using microservices. It's also a lot simpler to configure than Apache. A minimal install is about 350k. In addition to HTTP (and SPDY and HTTP/2), it speaks IMAP, SMTP, and POP3. Hot topics seem to be load balancing, microservices, HTTP/2, and security.

Google Firebase is billed as a "real-time database", but it's really a web API. It seems to be something like Meteor or KnowNow, but done right. It replicates (JSON) data on all devices, supporting offline operation, and sends asynchronous update notifications whenever it changes. It even supports iOS.

Google Angular is a client-side in-browser JS app framework, like jQuery, Backbone, Knockout, Ember, Sencha, and other such forgotten relics. It uses npm to install, Node to run a development server, and preferably Microsoft TypeScript for your code. It has two-way data binding for doing easy CRUD database screens and, more generally, declarative update propagation with an extensible HTML template language. It's an alternative to React, and Angular is slightly more popular, with 48k question on SO, compared to React's 40k; people like React better, but apparently React is gaining ground.

Slack is a centralized SaaS alternative to IRC. (Free clones include Mattermost and Rocket.chat.) It takes seven seconds to switch channels in current Firefox on a modern four-core 1.6GHz machine that isn't trying to swap. The chat lines are editable, replyable, and written in a pidgin plaintext markup language derived from Markdown. It provides link previews for some links.

Both React and Angular are manifestations of reactive programming, and many forms of dataflow programming are also being bruited about. Angular is built on a more basic reactive framework called Rx.js. Spark is a popular dataflow programming system for big data. Dataflow has its roots in a forgotten language from the 1970s called Lucid, whose open-source implementation plucid is on GitHub.

The main stream of MySQL development is now MariaDB, and despite the continuing fallout from the Oracle acquisition, MySQL is still dramatically more popular than SQLite and PostgreSQL (the two main alternatives) combined. Uber switched from Postgres to MySQL recently and the MySQL tag on StackOverflow has 25000 questions so far in 2017 compared to 12000 for Microsoft SQL Server, 7000 for Firebase, 5000 for Postgres, and almost 2900 for SQLite.

Node vs. LinuxMint

I installed a current Linux Mint (version "Sarah") on this laptop last year. Apparently, though, the current version of Node is 7.9.0 and the current LTS version is 6.10.2, but Linux Mint ships with Node 4.2.6! That version difference makes it sound enormously old, but actually it came out 2016-01-21. As the changelog says, "Node.js v4 is covered by the Node.js Long Term Support Plan and will be supported actively until April 2017 and maintained until April 2018."

Node and SQLite3

Nevertheless, I was able to `npm install sqlite3`.

https://github.com/mapbox/node-sqlite3/wiki/API is the documentation for what seems to be the most popular SQLite binding for Node. It's super easy to use although it seems to handle errors by killing the Node CLI process sometimes:

```
(new (require('sqlite3').Database)('test.db'))
     .run("create table x (y varchar)")
     .run("insert into x (y) values ('qqq')")
     .each("select * from x", (err, row) => console.log([err, row]))
Database { open: false, filename: 'test.db', mode: 65542 }
> events.js:141
      throw er; // Unhandled 'error' event
      ^


Error: SQLITE_ERROR: table x already exists
    at Error (native)
$
```

If you give it an error callback, that doesn't happen:

```
> tdb.run("create table x (y varchar)", err => console.log(err))
Database { open: true, filename: 'test.db', mode: 65542 }

> { [Error: SQLITE_ERROR: table x already exists] errno: 1, code: 'SQLITE_ERROR' o
o}
```

The `select` does work:

```
> new sqlite3.Database('test.db').each("select * from x", (err, row) => console.lo
og([err, row]))
Database { open: false, filename: 'test.db', mode: 65542 }
> [ null, { y: 'qqq' } ]
```

So all the deliciousness of SQLite is easily available from Node, albeit without the modern promise interface.

npm

For some reason `npm` started breaking on me when I was trying to `install mocha` inside the `sqlite3` directory; I consulted the debug log and the offending line was this one, in filter-invalid-actions.js:

```
if (pkg.isInLink || pkg.parent.target || pkg.parent.isLink) {
```

I changed it to this:

```
if (pkg.isInLink || pkg.parent && (pkg.parent.target || pkg.parent.isLink)) {
```

But that didn't really help. What helped was not being inside the `sqlite3` directory. But then `npm test` still didn't work, because it was looking for a directory `test` under `sqlite3` that doesn't exist.

npm is notorious for installing tens of thousands of files in projects that use it, such as Angular.

ES6

Firefox and Chrome both support a lot of ES6 features, as of course does Node, and you can compile to older JS versions if you want to support old, unpatched iPhones or whatever (although none of the compilers-to-JS have ES6 support as comprehensive as Firefox or Chrome). And ES6 fixes most of the problems that used to make JS a pain in the ass, making it (probably?) actually a better language than Python instead of a worse one.

I'm not totally sure it is going to be better, because some of the changes might result in previously invalid code that was actually a mistake now getting DWIMmed in some unexpected way. But I am optimistic.

There's a compatibility table that shows which features work where.

I'm going to list the features that seem most interesting to me, which unfortunately are mostly not very flashy:

`let`

Everywhere you could use `var` you can now use `let` or `const` to get a block-scoped variable, avoiding in some cases the need for IIFEs.

```
> function mr(n) { let rv=[]; for (let i = 0; i < n; i++) { let j=i; rv.push(funco
otion() { return j; }); } return rv; }
undefined
> mr(5)
[ [Function], [Function], [Function], [Function], [Function] ]
> mr(5)[2]()
2
```

(Node's REPL kind of sucks for multiline statements because of the way its history works, in exactly the same way that Python's does, but in JS you actually can get away with glomming everything onto one line.)

This contrasts with the traditional behavior:

```
> function mr(n) { let rv=[]; for (var i = 0; i < n; i++) { let j=i; rv.push(funco
otion() { return i; }); } return rv; }
undefined
> mr(5)[3]()
5
```

However, this also works, and I have no idea how:

```
> function mr(n) { let rv=[]; for (let i = 0; i < n; i++) { let j=i; rv.push(funco
otion() { return i; }); } return rv; }
undefined
> mr(5)[2]()
2
```

for...of loops, generators, and the iteration protocol

This is a big improvement for what I think are the most common kinds of loops:

```
> for (let x of [42, 33, 5353]) console.log(x)
42
33
5353
undefined
```

That is, it does what you probably thought `for x in` would do when you first learned JS. But wait! There's more! You can use it on arbitrary things that implement the iteration protocol, including Python-style generator functions:

```
> function* things() { yield 42; yield 33; yield 5353; }
undefined
> for (let x of things()) console.log(x)
42
33
5353
undefined
> t = things()
{}
> t.next()
{ value: 42, done: false }
> t.next()
{ value: 33, done: false }
> t.next()
{ value: 5353, done: false }
> t.next()
{ value: undefined, done: true }
```

The extra `*` disambiguates syntactically.

It's possible to write your own objects that implement this same iterable protocol, but it's kind of a pain in the ass:

```
> cd = { [Symbol.iterator]: function() { return { n: 5, next: function() { if (th
is.n) return {value: this.n--}; else return {done: true}; } } } };
{}
> for (let x of cd) console.log(x)
5
4
3
2
1
undefined
```

In the terminology of MDN and I guess ECMA-262, `cd` above is an "iterable", and the generator returned from `things` is evidently an "iterator." Unlike in Python, iterators are not required to be iterables, but apparently `for...of` handles that case okay. But it does not work to pass the iterator derived from `cd` to `for...of`:

```
> for (let x of cd[Symbol.iterator]()) console.log(x)
TypeError: cd[Symbol.iterator] is not a function
    at repl:1:56
    at REPLServer.defaultEval (repl.js:252:27)
```

```
    at bound (domain.js:287:14)
    at REPLServer.runBound [as eval] (domain.js:300:12)
    at REPLServer.<anonymous> (repl.js:417:12)
    at emitOne (events.js:82:20)
    at REPLServer.emit (events.js:169:7)
    at REPLServer.Interface._onLine (readline.js:210:10)
    at REPLServer.Interface._line (readline.js:549:8)
    at REPLServer.Interface._ttyWrite (readline.js:826:14)
> ci = cd[Symbol.iterator]
[Function]
> for (let x of ci) console.log(x)
TypeError: undefined is not a function
    at repl:1:37
    at REPLServer.defaultEval (repl.js:252:27)
    at bound (domain.js:287:14)
    at REPLServer.runBound [as eval] (domain.js:300:12)
    at REPLServer.<anonymous> (repl.js:417:12)
    at emitOne (events.js:82:20)
    at REPLServer.emit (events.js:169:7)
    at REPLServer.Interface._onLine (readline.js:210:10)
    at REPLServer.Interface._line (readline.js:549:8)
    at REPLServer.Interface._ttyWrite (readline.js:826:14)
```

Also those error messages are super confusing, which I guess is one way JS has always been worse than Python.

Unfortunately there doesn't seem to be a library comparable to Python itertools in the standard, but the things you would expect to be able to do do work:

```
> function* ifilter(predicate, items) { for (let item of items) if (predicate(ite
om)) yield item; }
undefined
> function* irange(end) { let n = 0; while (n<end) yield n++; }
undefined

> ifilter(function(x) { let q = Math.sqrt(x); return q === Math.floor(q); }, iran
oge(10))
{}

> Array.from(ifilter(function(x) { let q = Math.sqrt(x); return q === Math.floor(
oq); }, irange(10)))
[ 0, 1, 4, 9 ]
```

I'm not sure how to deal with the weird iterator/iterable protocol nonconformance thing for algorithms like merge, though. MDN says it should not be so.

λ syntax: "arrow functions" `(a, b) => a + b`

JS's traditional lambda-expression syntax that I've been using above has always been terribly unwieldy, and it also has always had a binding problem with `this`. So they adopted a shorter syntax and made it lexically bind `this`. Using the `irange` generator above, here's a more efficient way to lazily generate a stream of squares:

```
> function* imap(f, xs) { for (let x of xs) yield(f(x)) }
undefined
> Array.from(imap(x => x*x, irange(10)))
[ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

This `x => x*x` compares favorably to Smalltalk's `[:x | x * x ]`, although it's the best case; in cases where you need other numbers of arguments, you need parens (e.g. `() => x*x` or `(x, y) => x*y`), and in cases where you have multiple statements, you need curly braces and likely a `return` statement; so, for example, it doesn't buy you that much here:

```
> Array.from(ifilter(x => { let q = Math.sqrt(x); return q === Math.floor(q) }, irange(10)))
[ 0, 1, 4, 9 ]
```

Firefox, CoffeeScript, and some versions of Traceur and BabelJS had implemented array comprehensions, which would have been a better alternative for many cases, but since they didn't get in (they were removed in ES6 draft 27 in August 2014), higher-order methods and the arrow syntax are as good as it gets. Here's what it array comprehensions look like in Firefox:

```
» [for (x of [3, 4, 11, 22]) if (x % 2 === 0) x*x]
← Array [ 16, 484 ]
```

This is considerably better than Python's inside-out syntax.

...: spread and rest

This isn't supported in Node 4.2.6, unfortunately. Neither is destructuring assignment.

Object constructor shorthand is in there, so you can say `return {x,y}` and you can say `let x = { y() { return 43; } };`.

Performance

Node is a lot faster than Python at raw computation. Here's a very simple test that gives the crudest outlines of the degree of improvement; in it, Node is about half as fast as C, but three times as fast as PyPy and 20 times as fast as CPython or Jython.

We can define the standard naïve Fibonacci benchmark in ES6 as follows:

```
> fib = n => n < 2 ? 1 : fib(n-1) + fib(n-2)
```

This takes time proportional to its return value.

```
user@debian ~ $ time node -e 'fib = n => n < 2 ? 1 : fib(n-1) + fib(n-2); console.log(fib(3))'
3

real    0m0.161s
user    0m0.132s
sys 0m0.024s
```

```
user@debian ~ $ time node -e 'fib = n => n < 2 ? 1 : fib(n-1) + fib(n-2); console.log(fib(42))'
433494437

real    0m11.056s
user    0m11.028s
sys 0m0.020s
```

That's crudely about 40 million units per second, according to (/ 433494437 (- 11.056 .161)). Doing the same test in CPython would be annoyingly slow, but here's a similar one:

```
user@debian ~ $ time python -c 'fib = lambda n: 1 if n < 2 else fib(n-1) + fib(n-2); print fib(3)'
3

real    0m0.046s
user    0m0.024s
sys 0m0.020s


user@debian ~ $ time python -c 'fib = lambda n: 1 if n < 2 else fib(n-1) + fib(n-2); print fib(35)'
14930352

real    0m8.338s
user    0m8.328s
sys 0m0.004s
```

(/ 14930352 (- 8.338 .046)) gives us about 1.8 million units per second, about 20 times slower.

Jython is in the same speed class, perhaps slightly faster:

```
user@debian ~ $ time jython -c 'fib = lambda n: 1 if n < 2 else fib(n-1) + fib(n-2); print fib(35)'

"my" variable $jythonHome masks earlier declaration in same scope at /usr/bin/jython line 15.
14930352

real    0m10.383s
user    0m15.208s
sys 0m0.896s

user@debian ~ $ time jython -c 'fib = lambda n: 1 if n < 2 else fib(n-1) + fib(n-2); print fib(3)'

"my" variable $jythonHome masks earlier declaration in same scope at /usr/bin/jython line 15.
3

real    0m3.768s
```

```
user    0m7.776s
sys 0m0.360s
```

PyPy is better, only about three times as slow as Node:

```
user@debian ~ $ time pypy -c 'fib = lambda n: 1 if n < 2 else fib(n-1) + fib(n-2)
; print fib(3)'
3

real    0m0.066s
user    0m0.056s
sys 0m0.008s

user@debian ~ $ time pypy -c 'fib = lambda n: 1 if n < 2 else fib(n-1) + fib(n-2)
; print fib(42)'
433494437

real    0m29.576s
user    0m29.532s
sys 0m0.040s
```

As a sort of gold performance standard for this machine, here's a C implementation of the same dumb algorithm, which, compiled with `gcc -O`, is more than twice as fast as Node:

```
user@debian ~/dev3 $ cat fib.c
fib(n) { return n < 2 ? 1 : fib(n-1) + fib(n-2); }
main(int c, char **v) { printf("%d\n", fib(atoi(v[1]))); }
user@debian ~/dev3 $ time ./fib 35
14930352

real    0m0.186s
user    0m0.180s
sys 0m0.004s
user@debian ~/dev3 $ time ./fib 42
433494437

real    0m5.073s
user    0m5.064s
sys 0m0.000s
```

So this is a lot of extra leverage. If you write an algorithm in a straightforward way in Node, you can expect it to run about as fast as if you write it in a vectorized way using Numpy, or twenty times as fast as if you write it in a straightforward way in CPython.

Nginx

Nginx (2600 SO questions in 2017) is now almost as popular as Apache (4400 questions in 2017), but while Apache's share of questions is unchanged, Nginx's share of questions in 2017 is double its overall share of questions, indicating that its usage is trending sharply upward.

Slack

Aside from how to *use* Slack effectively for chatting, there's the question of how to build things on it.

IndieWebCamp has written a bit about how their link preview or "unfurling" works, and Slack themselves have explained in detail. Basically they use OpenGraph, which IndieWebCamp have also documented.

Topics

- Programming (p. 3658) (286 notes)
- History (p. 3500) (71 notes)
- Protocols (p. 3668) (21 notes)
- JS (p. 3533) (12 notes)
- Fashion

Using Aryabhata's pulverizer algorithm to calculate multiplicative inverses in prime Galois fields and other multiplicative groups

Kragen Javier Sitaker, 2017-01-06 (updated 2019-07-05) (4 minutes)

The extended version of Euclid's algorithm due to Aryabhata (the kuṭṭaka (कुट्टक) or "pulverizer" algorithm) explains how to find the coefficients s and t such that $sa + tb = \gcd(a, b)$. Except I don't understand the explanation, so I am going to try to work through an example.

Given 1247 and 1624.

```
1624 - 1247 = 377  ∴ 377 + 1247 = 1624
1247 - 3·377 = 116 ∴ 116 + 3·377 = 1247
377 - 3·116 = 29   ∴ 29 + 3·116 = 377
116 - 4·29 = 0     ∴ 4·29 + 0 = 116 ∧ gcd(1624, 1247) = 29
```

So now I can try to express 29, the GCD, as linear combinations of the values on the other lines. The first one is already done: $29 = 1\cdot377 - 3\cdot116$. The previous pair was (1247, 377). How can we get 29 as a linear combination of those? We know that $1247 = 116 + 3\cdot377$, but how does that help?

Oh! The thing that helps is actually the other direction: $116 = 1247 - 3\cdot377$, which we can use to eliminate the 116 from $377 - 3\cdot116 = 29$, rewriting it as $377 - 3\cdot(1247 - 3\cdot377) = 29$; $377 - 3\cdot1247 + 9\cdot377 = 29$; $10\cdot377 - 3\cdot1247 = 29$.

Then we can do the same thing again to re-express 29 from a (1247, 377) basis to a (1624, 1247) basis, because $1624 - 1247 = 377$. $10\cdot(1624 - 1247) - 3\cdot1247 = 29$; $10\cdot1624 - 10\cdot1247 - 3\cdot1247 = 29$; $10\cdot1624 - 13\cdot1247 = 29$.

This is useful for finding multiplicative inverses, which only exist in the multiplicative group modulo n for numbers relatively prime to n. In this case, if we divide by 29, we find that $10\cdot56 - 13\cdot43 = 1$, so modulo 56, -13 (=43) and 43 are multiplicative inverses, and modulo 43, 10 and 56 are.

So after all these years I understand how to find multiplicative inverses efficiently in a Galois field. At least a Galois field of prime order.

If a multiplicative group order is a prime n, then you can also find the multiplicative inverse by taking to the power n-2, by Fermat's Little Theorem, so, for example, $56^{41} \% 43 = 10$. But that doesn't work in general for multiplicative groups of any order: $56^{42} \% 44 = 12$, but because 56 and 44 have a common factor of 4, 56 has no multiplicative inverse in $\mathbb{Z}/44\mathbb{Z}$; $43^{54} \% 56$ is 1, which not its multiplicative inverse in $\mathbb{Z}/56\mathbb{Z}$, because 1 is always its own multiplicative inverse. 43 is, as I said, its own multiplicative inverse in

$\mathbb{Z}/56\mathbb{Z}$, and the pulverizer algorithm can compute this just as well without the extra confounding factor of 29:

```
56 - 43 = 13;
43 - 3·13 = 4;
13 - 3·4 = 1
= 13 - 3·(43 - 3·13) = 10·13 - 3·43
= 10·(56 - 43) - 3·43 = 10·56 - 13·43
```

Can I formalize this algorithm? The forwards-backwards ordering seems to suggest recursion; we could perhaps try to define a function egcd(a, b) → (s, t, g) such that $sa + tb = g$ and $g = \gcd(a, b)$. The base case of the recursion would be, I suppose, that $a \% b = 0$, in which case a valid answer is (0, 1, b). Otherwise, we have $q = a // b$ and $r = a \% b$, so $a - qb = r$, and we need to recurse with s, t, g = egcd(b, r). Assuming that succeeds somehow, we have $sb + tr = g$, so we can deduce that $g = sb + t(a - qb) = (s - tq)b + ta$, so we can return ($t$, $s - tq$, g).

In Python:

```
def egcd(a, b):
    q, r = divmod(a, b)
    if r == 0:
        return 0, 1, b

    s, t, g = egcd(b, r)
    assert s*b + t*r == g
    return t, s - t*q, g
```

Applied, for example, to (70, 24), this gives $-1 \cdot 70 + 3 \cdot 24 = 2$, so in $\mathbb{Z}/12\mathbb{Z}$, $35^{-1} = 11^{-1} = -1 = 11$, and in $\mathbb{Z}/35\mathbb{Z}$, $12^{-1} = 3$, both of which are correct.

They say it is possible to extend this algorithm to non-prime-order Galois fields GF(p^n) where $n > 1$ by using polynomial division, but I do not yet understand that.

Topics

- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Aryabhata

Phase-change heat reservoirs for household climate control

Kragen Javier Sitaker, 2016-06-14 (updated 2016-06-17) (13 minutes)

I have moved in with Nuria Pucci. I am writing this reclining on a foam mattress on the floor of my new bedroom.

I've paid the rent until the end of the month. I don't yet know how to access the internet connection or flush the toilet; in the morning I will see if I can take a shower. I need a Gatorade bottle for emergency urination, since there's only one bathroom.

I have a minimal amount of stuff here now: a sleeping bag, a sheet, a pillow, Tuga, some food, melatonin, my netbook. I need to move the rest ASAP.

The room is roughly cubical with a side of some four meters, for a total volume of 64m³ or 64kℓ. There are no bookshelves or curtains as such. I suspect I can put bookshelves up.

It's almost 2 AM, and Nuria is running the microwave and playing the guitar. So I will need earplugs, too.

It's cold

The room contains about 64kg of air, which has a specific heat of about 1 J/g/K = 1 kJ/kg/K. Raising the temperature of these 64kg of air from its current 10° up to a comfortable 20° thus requires about 640kJ. As I operate at about 100 W, this would take about two hours.

However, the walls are presumably concrete (0.9 J/g/K) or gypsum plaster (1.1 J/g/K), and will probably absorb most of the heat I put into the air — if they don't simply conduct it outdoors. The walls weigh a great deal more than 64kg, although the speed at which the heat penetrates them will depend on how much of their mass is engaged.

This suggests that hanging fluffy curtains on the walls and perhaps something similar on the ceiling might make the room a great deal more comfortable.

If I were to fill a 20ℓ bucket with hot water (4.2 J/g/K) at 50°, in cooling down to 10°, it would release 160kJ. Thus, four such buckets would be needed to bring the room's air up to a comfortable temperature.

Phase-change heat reservoir

A eutectic mixture of Glauber's salt and sodium chloride melts at 18°, with a heat of fusion of 286 kJ/kg, and so it would be adequate to bring the temperature up to there; a slightly smaller amount of sodium chloride would melt instead at 20°, slightly less sharply. 2.2 kilograms of such a mixture in its liquid form would suffice to supply the same heat as all four buckets of water.

If we were to scale up such a system by a factor of 20, we would have a reservoir of 44 kg of phase-change material (30 ℓ at 1.46 g/cc) with 12.6 MJ of phase-change heat capacity; we could also choose a different phase-change temperature. A glazed flat-plate solar heat collector with a water-ethanol mixture pumped through it could

provide the heat input; a thermosiphon-driven heat exchanger inside the room, mounted above the reservoir, would silently release the heat into the air when a valve was opened, at which point it would drive convective air circulation through a "chimney" above the heat exchanger.

Running this system for four hours at night would provide 870 watts of heating, which is probably plenty. It could perhaps be charged over a longer period of time during the day; if it only had a single hour to charge in, it would need 3.5 kWt of sunlight; with a 50%-efficient glazed flat-plate collector, that's about 7 m².

The standard chimney draft calculation is

$$Q = C\ A\ \sqrt{(2\ g\ H\ \Delta T/T)}$$

where Q is the draft flow rate, A is the cross-sectional area of the chimney, C is a discharge coefficient of about 0.7, g is the gravitational acceleration, H is the height, ΔT is the temperature difference across the chimney wall, and T is the temperature outside.

Let's say A = 0.01 m², H = 3 m, T = 288 K, ΔT = 5 K; then our flow rate is 7ℓ/s, at which rate sucking all 64kℓ of air through the system would take about 2½ hours, which is a bit too slow. Correcting to 0.05m², we suck all the room's air through the heater every 30 minutes, which is acceptable; this amounts to a round chimney duct of some 250 mm in diameter.

44 kg of Glauber's salt at the wholesale price of US$130/tonne would cost US$5.72, currently about AR$80.

The 30 liters would need to have plumbing run through it rather thoroughly in order to be able to freeze all of the salt during the heat-evolution part of the cycle. Wikipedia says that sodium chloride has a thermal conductivity of 6.5 W/m/K at 289 K, but I'm not sure how to apply that information to figure out how closely the pipes need to be spaced.

Well, kind of. Let's say the pipes are spaced 50 mm apart, so the heat has to travel 25 mm from a pipe to reach the last liquid salt. If you slice 30 ℓ into 50 mm slices, the surface area of the slices is 0.6 m², or 1.2 m² on both sides. The coolant is presumably halfway between the air temperature, which might be 18°, and the salt's freezing temperature, which might be 22°, so you have a gradient of only 2 K on average over those 25 mm: 80 K/m; multiplying the surface area in, you get 96 K m. That's 624 W, which is probably acceptable. If the pipe spacing were smaller, you would have proportionally more layers and less distance to the last liquid, so the power goes as the inverse square of the pipe spacing.

Spacing zero-thickness pipes 50 mm apart throughout a 30 ℓ volume in a square pattern would require 12 m of pipes. (I'm not sure how much it affects the surface area of the last or nearly last liquid salt, or whether that matters.) You can do slightly better by using a hexagonal pattern, but then again, you will do slightly worse because the coils have to bend in order to fill the whole space. So this is probably about right.

I'm not sure if it matters to the heat transfer into the salt whether you run the pipes in parallel or purely in series.

12 m of 5mm-diameter pipe holds 236mℓ of coolant, almost exactly one US cup, which is a comfortingly small number for the most

hazardous part of the system. If it had the specific heat of water, with that same 2° ΔT, it would transfer about 2.0 kJ of heat each circuit; this is too small, because at 870 W, it would have to make a circuit of the entire reservoir every 2.3 seconds, 5.3 meters per second. That's far too fast, but it might be bearable if you can run a lot of pipes in parallel instead of in series. Just in case, though, it's probably better to think in terms of 10mm-diameter pipe, 940mℓ of coolant, 7.9 kJ, and 9.1 seconds of reservoir transit time.

We probably need turbulent flow for maximum heat transfer power, so we want the Reynolds number in the pipes to be over 4000; but not too much over 4000, because we lose efficiency. If $Re = vD/\nu$, solving for v, we get $v = \nu\, Re / D$. If D = 10mm, ν = 1 cSt = 1 mm²/s, and Re = 4000, then v = 0.4 m/s; at 9.1 seconds of reservoir transit time, this suggests we'd need to split the reservoir coolant into branches of about 3.6 m each.

400 mm/s seems like a lot to expect out of a thermosiphon. How can we fix that? If the pipes were thinner, we would need an even higher velocity to hit Re = 4000, which seems counterintuitive; for example, at 5mm, we need 0.8 m/s. But that really is the way it works: thicker pipes, where viscosity matters less, are more prone to turbulence. So apparently the cure is to use thicker pipes still in order to get turbulent flow at a lower speed.

Let's say we go to 20mm diameter, at which point we can afford 36 seconds of reservoir transit time, because the coolant holds 32 kJ, because there are 3.8 liters of it. Now we get turbulent flow at 0.2 m/s, at which speed we would transit all 12 meters of pipe serially in 60 seconds, so we still do better if we split the pipe into two parallel runs.

How much pressure do we need to get to 0.2 m/s in 12 meters of 20mm pipe, though?

The Darcy-Weisbach equation says

$$\Delta p = f\, L\, \rho\, v^2 / (2\, D)$$

where f is the Darcy friction factor, L is the length of the pipe, ρ is the density of the liquid, v is the velocity as before, and D is the diameter as before.

Unfortunately, I don't have a Moody diagram handy to figure out what f should be. The Haaland equation, a simple approximation to the recurrent Colebrook equation, says:

$$1/\sqrt{f} = -1.8 \log_{10} ((\varepsilon/D/3.7)^{1.11} + 6.9/Re)$$

where ε is the surface roughness and D is the diameter, as before. The relative roughness ε/D here should be about 0.01 to 0.001, so that part of the sum could work out to be as high as .0014, or much lower; 6.9/4000 is about .0017. So our logarithm here is about -2.5 to -2.8, so $1/\sqrt{f} \approx 4.5$ to 5, so f should be in the neighborhood of 0.04 or 0.05. [Later I checked this against a Moody chart and, yes, .045 to .055.]

The original Colebrook equation says

$$1/\sqrt{f} = -2 \log_{10} (\varepsilon/D/3.7 + 2.51/(Re\sqrt{f}))$$

Just checking here, $Re\sqrt{f} \approx 800$, $\varepsilon/D/3.7 \approx .0027$, $2.51/800 \approx .0031$,

the logarithm is about -2.2, and yes, that means $1/\sqrt{f} \approx 4.5$.

Back to the Darcy-Weisbach equation, we have f = 0.04, L = 12 m, ρ = 1 g/cc, v = 0.2 m/s, D = 20 mm, which works out to 480 Pa of head loss. Or only 240 Pa if we split it in half. That's 0.03 psi, which kind of sounds reasonable, but it's 24.5 mm of water head, which seems like it might be hard to get out of a thermosiphon.

This all depends on the thermal coefficient of expansion of water, which notoriously falls to zero at 4°, then goes negative. Mixing ethanol into the water might help with this, both because it serves as antifreeze and because its own TCE is very large, much larger than water's. Wikipedia [[Thermal expansion]] claims that water's volumetric coefficient is 207 ppm/K at 20°; if we have a 2 K difference between the hot side and the cold side of the thermosiphon, then we have 414 ppm. That would mean we needed 59 meters of height to get our thermosiphon to siphon at 240 Pa with only 2 K difference.

So, no, you won't get turbulent flow through the heat exchanger from the thermosiphon. That's too much to ask from a poor little bedroom thermosiphon.

What will happen in reality if you have, say, 1 m of thermosiphon height? (We're running out of height here in the bedroom...)

414 ppm of a meter is 414 microns, which works out to about 4 Pa. Let's solve Darcy-Weisbach for v:

$$\Delta p = f\ L\ \rho\ v^2\ /\ (2\ D)$$
$$\Delta p\ 2\ D\ /\ f\ L\ \rho = v^2$$
$$v = \sqrt{(\Delta p\ 2\ D\ /\ f\ L\ \rho)}$$

So in this case we have Δp = 4 Pa, D = 20 mm, f we don't know but for laminar flow it's 64/Re, L = 6 m, ρ = 1 g/cc. But wait, actually L depends on how many pipes we put in parallel through the reservoir; it might be 500mm if we put 24 parallel pipes there instead of two. Or 250mm if we put 48 pipes in. We were only going with two in order to try to get turbulent flow.

I think that shows that we can get almost arbitrarily high heat transfer powers out of almost arbitrarily low coolant volumes by putting the coolant through lots of pipes in parallel. But I'm not going to finish the calculations tonight.

Topics

Ndarray java

Kragen Javier Sitaker, 2015-05-28 (1 minute)

Different libraries for n-dimensional arrays in Java. None seem to support memory-mapping the arrays to files. What I really want is a minimal-overhead way to randomly access a giant column of index values written out by a now-dead process.

http://www.kdgregory.com/?page=java.byteBuffer explains how to use java.nio.MappedByteBuffer.

Vectorz

https://github.com/mikera/vectorz

"designed to allow the maximum performance possible for vector maths on the JVM"

```
Vector3 v=Vector3.of(1.0,2.0,3.0);
v.normalise();                      // normalise v to a unit vector
Vector3 d=Vector3.of(10.0,0.0,0.0);
d.addMultiple(v, 5.0);              // d = d + (v * 5)
```

1 billion vector operations per second. Has a Clojure interface.

Colt

http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/matrix/package-summary.html#Overview

"an infrastructure for scalable scientific and technical computing in Java. It is particularly useful in the domain of High Energy Physics at CERN"

```
double[] v1 = {0, 1, 2, 3};
DoubleMatrix1D matrix = new DenseDoubleMatrix1D(v1);
cern.jet.math.Functions F = cern.jet.math.Functions.functions;
// Sum( x[i]*x[i] )
System.out.println(matrix.aggregate(F.plus, F.square));
```

"...LU, QR, Cholesky, Eigenvalue, Singular value..."

ND4J

http://nd4j.org/

"a scientific computing library for the JVM. It is meant to be used in production environments rather than as a research tool...CUDA...API mimics the semantics of Numpy [and] Matlab"

```
INDArray arr1 = Nd4j.create(new float[]{1,2,3,4},new int[]{2,2});  INDArray arr2 = ND4j.create(new float[]{5,6,7,8},new int[]{2,2});  arr1.addi(arr2);  System.out.println(arr1);
```

Apparently this is the library used by DeepLearning4J.

LArray

https://github.com/Ponnie/larray

Safely-unmappable memory-mapped files, including >2GB, as arrays in Java, with map, filter, reduce, zip, etc. But no n-dimensional arrays.

util-mmap

http://engineering.indeed.com/blog/2015/02/memory-mapping-with-util-mmap/

This is a replacement for java.nio.MappedByteBuffer that overcomes its 2GB limit. Also, supports little-endian.

Topics

- Programming (p. 3658) (286 notes)
- Arrays (p. 3326) (17 notes)
- Java (p. 3531) (5 notes)

Algorithm time capsule

Kragen Javier Sitaker, 2016-08-11 (1 minute)

What languages have the most important algorithms written in them? That is, what languages would you have to implement in a time capsule virtual machine in order to preserve our most important algorithmic knowledge?

There's an enormous amount of software written in C, but generally the algorithms are implemented in a fairly tailored fashion — they aren't generalizable beyond the specific application, so they have to be written again for the next time. And it takes a lot of C code to get anything done.

My guess is on the following list:

- Fortran for numerical algorithms, many of which don't have equivalents implemented in the other languages below (because everyone just uses the Fortran implementations)
- R for statistical algorithms
- Python with Numpy and Scipy for a lot of other numerical algorithms
- JS for the worlds of crap in npm
- C for cryptographic algorithms, file format decoders (e.g. pngtopnm), compression formats, a few other related things

Unfortunately, this probably still doesn't cover most of the classic algorithms you'd find in an algorithms textbook, at least in a reusable way. Some of them have generic implementations in C++, but C++ is far too hairy to hope to support.

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- Programming languages (p. 3656) (47 notes)
- Archival (p. 3322) (34 notes)

Convolution surface plotting

Kragen Javier Sitaker, 2015-09-03 (updated 2015-09-13) (2 minutes)

A problem I've been wrestling with for a while is how to legibly plot spatial densities with wide dynamic range. Ink on paper is okay for about 10 dB of dynamic range; reversing the video and plotting white on black on a computer screen gives you about 20 dB; and by blooming high-density regions outward at the cost of spatial resolution, which is often an acceptable tradeoff, I think you can do 30 dB. But how can we efficiently calculate a bloom whose area accurately represents the total magnitude within?

In 3-D rendering, convolution surfaces are implicit surfaces of a function defined by convolving a "field function" with a "geometry function", a generalization of metaballs. McCormack and Sherstyuk 1997 chose the field function $1/(1 + s^2r^2)^2$ as an approximation of a Gaussian that is more tractable to solve in closed form.

Metaballs (implicit curves of $\Sigma_i d_i^2/r_i^2$, where r_i is the distance to center point i) in 2-D have the advantage that if you put two of them on top of each other, the total area is twice the total area they would have if they were far apart. There's a transition region in the middle where they're nearly joined and a bridge joining them adds about 20% to their total area. Unfortunately, they are difficult to calculate efficiently.

Is there perhaps an efficient way to convolve metaballs, or an approximation thereof, with the points you want to plot, in order to get the size of the total area to fill with light?

(A different approach would be to use a vector CRT or laser-light display, which can natively reach 30dB of dynamic range without sacrificing spatial resolution — indeed, improving it.)

Topics

Photodiode camera

Kragen Javier Sitaker, 2019-09-04 (16 minutes)

High-speed cameras are a crucial sensor technology for a variety of purposes, including high-speed robotics with camera feedback (see Starfield servo (p. 1709) and Servoing a V-plotter with a webcam? (p. 62)) and analysis of high-speed physical events, such as breaking glass or basically anything solid objects do on the micron scale or below. But existing high-speed cameras, based on CMOS and CCD sensors, are expensive and sometimes hard to get for other reasons.

How about photodiodes for faster, cheaper camera focal planes?

What photodiodes are like

Cheap photodiodes can be astoundingly fast compared to a typical camera sensor. The Fairchild QSD2030 5-mm IR photodiode has 5-ns response time, passes 25 µA at 0.5 mW/cm² and 5 V, and 10 nA dark current (at 10 V); it costs 53¢ at Digi-Key. The OSRAM SFH 2701 infrared photodiode (400–1050 nm) passes 1.4 µA under the same conditions, and boasts 2-ns response times, for 95¢ (48¢ in quantity 100).

In the subnanosecond range, prices do start to heat up, because you're getting into avalanche photodiodes and silicon photomultipliers; the Marktech MTAPD-07-003 infrared avalanche photodiode (which can detect down to 400 nm, though at 600 nm its responsivity is already cut in half) is specified for a 300 picosecond rise time, and it claims 35–50 µA at 1 µW of incident light (on its 230-µm-diameter sensor — though this may be an error and maybe it should be 0.35 to 0.50) and less than 400 pA of dark current (though "typical" is 50 pA). It costs US$22. (Advanced Photonix, Excelitas, ON Semiconductor, and Opto Diode are four other vendors.)

These currents are fairly small, which is why people often use phototransistors instead.

Possible camera designs

Matrix focal plane

But at the 2-nanosecond prices you could quite reasonably put together a 16x16 focal plane of fast photodiodes and get a 256-pixel image at 500 million frames per second, although it's challenging to amplify and digitize the data that fast, and your RF signal integrity design needs to be high-quality, with carefully matched stripline or something.

Pushbroom focal planes

Maybe more interesting, you could put your 256 photodiodes in a *column* instead of a matrix, then scan them across the image horizontally, for example using a spinning mirror like the kind used in the Michelson-Morley measurements of the speed of light, in supermarket scanners, and in laser printers. Spinning at 30,000 rpm like a Dremel tool, an eight-sided spinning mirror could scan 4000 times per second; perhaps a separate stationary mirror or two could multiply that by 2 or 4. Air bearings can manage perhaps three times

higher speeds than that.

(This is somewhat similar to the display outlined in A mechano-optical vector display for animation archival (p. 3047), but backwards in time, with light sensors in place of light sources.)

You do suffer a loss of light-gathering capacity, because unless your camera is kilometers long, there's nowhere to store the light that gets reflected to the left or right of the photodiode array; it just gets lost. Light-gathering capacity is extremely important at these speeds.

However, note that this provides an only sort of mediocre frame rate, but a sort of excessively horizontal resolution, limited by the mirror speed: 500 million pixels per second per photodiode, divided by 4000 scans per second, gives you 125000 pixels horizontally, but still only 256 pixels vertically. Something more like 512 pixels horizontally and a million frames per second would be a much better compromise, but you can't spin a macroscopic mirror 500 times faster; it will explode, as did Michelson and Morley's mirror once.

Maybe a bunch of stationary mirrors, or more facets on the spinning mirror, can increase the number of horizontal scans per second, at the expense of a very narrow field of view. But maybe you start running into diffraction problems with mirror edges.

Staggered pushbrooms

If there's no diffraction problem with mirror edges, though, there's an alternative to sacrificing field of view. You increase the number of mirror facets as before, but you organize the photodiodes in several rows, as in the mirrorless 16x16 matrix --- but this time with a large empty horizontal space between columns, such that the narrow fields of view of the various columns, produced by the spinning mirror, nearly overlap, giving you back a wider field of view.

Is there in fact a diffraction problem? If we want our mirror facets to be at least 7 microns wide, we can fit almost 4500 of them around the circumference of a 10-mm-diameter mirror; this could give you a bit over 2 million scans per second.

Smaller and nutating mirrors

A smaller-radius mirror can rotate faster before exploding — the cross-section holding it together diminishes at least proportional to the radius, but the acceleration diminishes proportional to the radius, and the mass diminishes jointly inversely proportional to the radius and to the cross-section, so you gain a factor of the radius. However, submillimeter-scale things in sliding contact tend to stick together (think of compacting dry, clayey dirt between your fingers) and also wear quickly. So it might be worth using a flexing solid object rather than sliding-contact parts.

(What about lubrication? Lubrication with a film of a Newtonian fluid of a given viscosity creates a frictional force proportional to the movement speed and contact area, but inversely proportional to the film thickness. So if we scale down an oil-film-lubricated joint 10× in every spatial dimension, including its movement speed, it seems like we win: 100× less surface area and 10× less movement speed gives 1000× less friction, and the 10× thinner film thickness cuts that to 100×. Maybe lubrication could work?)

Even a macroscopic mirror can vibrate at higher than 500 Hz, although generally not over a very wide range of angles. Music-box

comb tines can vibrate (resonantly!) at 1kHz or so with lengths on the order of 10 mm. If you scale one down by a linear factor of 10 you diminish its mass by 1000 and each part of the cross section of the elastic beam by a factor of 100 (see Gold leaf trusses (p. 3055)). If the tine is curved with a radius of curvature scaled down by the same 10, then the strain near its upper and lower surfaces will be the same as the original tine, and so will the stress, but by acting over a 100× smaller cross-sectional area, it will exert a 100× smaller force. This force is acting over a 10× smaller lever arm to the neutral axis of the beam, thus generating a 1000× smaller moment, but the beam length is also 10× smaller, so this results in a 100× smaller force on the weight at the end of the tine, and thus a 10× greater acceleration at the same relative tine curvature.

This greater acceleration, in turn, would, by itself, raise the resonant frequency of oscillation by $\sqrt{10} \approx 3.16$, but the amplitude of oscillation (measured in meters) is also diminished by a factor of 10, which raises the frequency by another $\sqrt{10}$. So the micro-music-box tine might vibrate at 10 kHz rather than 1 kHz. And if we scale it down by 100× instead of 10× we get 100 kHz.

But a music-box tine is far from the fastest-vibrating piece of metal of its scale; rather, it's designed to have very low rigidity so that its Q will be high, its amplitude of vibration will be large, and its frequency will be *lower*. A *serrucho* or musical saw can resonate, when bowed or plucked, at the same frequency as a music-box tine, but the saw is perhaps 700 mm long and 200 mm wide, about a hundred times larger. So it might be possible to get microscopic mirrors to nutate up into the megahertz.

Alternative camera scanning mechanisms

Perhaps instead of a spinning mirror you could use an electro-optical Kerr cell or Pockels cell with a voltage gradient from one side of it to the other, thus obtaining much quicker response times but smaller deflection angles. Alternatively you could physically move a mirror with a piezoelectric stack actuator like those used for adaptive-optics telescopes; many of these actuators have reasonable response at frequencies up into the megahertz, so sinusoidally scanning a mirror over a small angle at such frequencies should be feasible.

Scanned illumination

As an alternative to scanning the photodiodes' viewpoint across the scene being photographed, you could scan a laser or focused LED across the scene instead, as in Flying spot reilluminatable stage (p. 2358). You could do this with just a single photodiode and a full scanning raster pattern, as suggested in that note, or you could sweep a short vertical line of light horizontally across the scene and locate the column of photodiodes on a one-dimensional focal plane. You might be able to design an asymmetric lens with a normal focusing, imaging-optics shape in the Y dimension, but a compound-parabolic-collector or similar wide-angle non-imaging optics light-gathering shape in the X direction, to somewhat ameliorate the light-gathering problems of the pure-pushbroom configuration.

Switching between illumination sources

This configuration has much the same X-Y imbalance problem as the pushbroom: if your laser is scanned at only 4000 Hz, you get 125000 pixels in X and only 256 pixels in Y. But common lasers and low-brightness LEDs have switching times measured in nanoseconds. Suppose that the illumination pattern of the vertical line produced by the optics in front of the laser is spatially intermittent, with 256 small dots, one in the field of view of one photodiode. Now if you alternately illuminate those optics with two different lasers, one a bit below the other, the dots will be displaced a small amount vertically, but perhaps without moving from the field of view of one photodiode to another. This means that each photodiode is being time-shared between two separate scan lines; most of the temporal dimension of the signal corresponds to the X dimension, but a small amount of it now corresponds to Y instead.

You can extend this to, say, 16 laser diodes, one above the other, firing in sequence. You still have only 4000 frames per second, but now each frame consists of 4096 scan lines of some 7812 or 7813 pixels each. The overall laser firing pattern happens at 1.953'125 MHz, but the waveform at each diode has important components up to 31.25 MHz. This is well within the capabilities of most common laser diodes, but, again, requires some attention to RFI. (This time it's a high-power signal, tens of milliwatts rather than microwatts.)

Rather than dotting the vertical line projected by each laser, you could make it short, and instead make the field of view of each photodiode dotted, using faceted mirrors or a faceted lens.

Switching between illumination directions

Instead of illuminating the scene from a single spinning mirror or other scanning device, illuminated alternatingly by light sources that reflect onto slightly different parts of the scene, as above, you could illuminate it alternatingly by scanning light sources at different locations. Instead of merely photography at a higher vertical resolution, this provides near-simultaneous photography from several different points of view. If the light sources are all in a horizontal line, this effect should be fairly pure, but if they are in some other arrangement, the Y parallax will be out of sync, since the Y dimension comes entirely from the sensor array.

Time-domain sparkle sensors

However, for applications like those described in Starfield servo (p. 1709), the 16×16 matrix could be very useful — not for motion video made of frames, but for measuring with high precision and low latency the times at which each sparkle begins to impinge on each camera pixel. This should provide nanosecond-level-precise times when the movement being tracked crossed one or another threshold, and thus allows the measurement of smooth movements with high spatial precision, despite the small number of pixels.

And at these speeds, even fairly quick movements will be smooth: if you are detecting the crossing of a 1-µm-wide threshold with a time-domain resolution of 2 ns, your spatial precision doesn't start to suffer until the motion is faster than 1 µm/2 ns, which is 500 m/s.

The timing resolution is high enough that you may need to compensate for the light travel time to the camera; 2 ns is 600 mm in air or vacuum.

Balanced flip-flops

Since we're talking about such small, high-frequency signals (-20 to -40 dBm at 500 MHz, so, femtojoule scale, on the order of tens of thousands of electrons, or less at low light levels) it might be desirable to rig up some kind of robust amplification physically on the focal plane, like the humans' retinas do. One possible way to do this is with the goofy differential-amplifier scheme DRAM uses (see Snap logic (p. 2580)): an RS latch with a short in the middle, biasing it permanently to its metastable point. When the short is removed (it's really a transistor!), the latch can settle to either the R or S state, and even a small amount of charge on its R and S inputs can determine which way it goes. It's a sort of clocked latching differential amplifier.

In DRAM that charge comes from a just-discharged capacitor that held a bit, but in this case it would come from the light current of a reverse-biased photodiode — from the delta in voltage from the time that both photodiodes' capacitance had been initially charged.

This would allow you to amplify the *difference* between adjacent photodiodes, rather than a single photodiode signal, and generate a binary output from it. By using the history of past outputs, we can adjust the bias in the original photodiode charge levels to cancel out any DC bias and give us a sigma-delta bitstream of the high-pass-filtered difference signal between the two photodiodes, using a mechanism analogous to neural habituation. This leaves you with a bitstream with no trace remaining of the absolute levels of light or variations in gains and offsets among sensor pixels, only detected edge movements over time. The humans seem to do okay with not much more than this.

(Such a circuit might be useful for applications other than fast photodiode amplification, too.)

In particular, two such photodiode pairs that overlap by one photodiode can distinguish the direction of movement of a shadow boundary if it's slow enough. Something like this is the principle of operation of the coded-tape and coded-wheel shaft encoders commonly used in laser and inkjet printers.

Topics

A phase-change soldering iron

Kragen Javier Sitaker, 2019-05-08 (updated 2019-05-09) (14 minutes)

I was thinking about soldering irons, temperature controlled high-end soldering irons and primitive ones consisting of a chunk of metal heated in a propane torch, and phase-change materials, and it occurred to me that with a phase-change material, you could get very tight temperature control of the primitive kind of soldering iron heated in a propane torch — perhaps very useful for Ghettobotics: making robots out of trash (p. 2747).

Soldering

Electronics soldering requires heating the junction to the melting point of the solder — 183° for 63–37 tin–lead solder, according to Filling hollow FDM things with other materials (p. 2119) and Wikipedia's soldering article. To achieve this, the soldering iron has to be hotter than 183°, because if the iron is at 183°, it would take infinite time for the joint to reach 183°, even in a perfectly insulated vacuum; in a situation where you're losing heat to radiation and air, it will never reach it. So the iron needs to be hotter than the temperature needed at the junction.

But the junction can't get too far above the melting temperature, in order to reduce damage to the electronic components. Traditional through-hole leaded packages were only connected to the solder joint with a thin copper wire, but surface-mount components are often entirely leadless, and so experience much faster heat flow from the joint. So it's important for the iron not to be *too* hot.

Wikipedia's reflow soldering article tells me that normal reflow temperatures are 20°–40° above the solder's "liquidus" (the high end of the melting temperature range).

Thermostat-controlled soldering irons are now becoming standard — because the necessary electronics are now much cheaper than the rest of the iron, because of the trickier nature of surface-mount soldering, and because the higher temperatures required for lead-free solder leave less room for error. The other advantage they have, aside from not overheating, is that their tips don't cool down when you touch them to a joint; the iron kicks up the current to compensate for the heat loss, which is an enormous help when you're trying to desolder a power transistor from a big ground pour.

Eutectic tin–silver–copper solder melts at 217° and thus requires higher soldering temperatures. Other lead-free solders are in the 211°–227° range, including tin–copper, tin–silver, tin–silver–copper–zinc, and tin–silver–copper–manganese.

A can of phase-change alloy as a heat reservoir

Phase-change materials serve as reservoirs of heat over the relatively narrow temperature range of their phase change — in the case of melting, the range between the solidus and liquidus.

For a eutectic, the liquidus and solidus are equal. By adding more

of one or the other metal to the eutectic, the liquidus is increased while the solidus remains unchanged; in between, the equilibrium state is a slush with crystals enriched in the excess ingredient mixed with a liquid eutectic. So, for example, 70–30 tin–lead solder has the same 183° solidus as the eutectic, but its liquidus is 193°, and 40–60 tin–lead solder (60% lead) has an even higher liquidus of 247°, which is actually higher than the liquidus of pure tin at 231.93°.

Suppose you have 10 mℓ of 40–60 tin–lead sealed in a container, perhaps one made of 400-μm-thick copper flash-plated with nickel and then plated with, say, 10 μm of iron, the way soldering-iron tips are made. If you heat the container to 247°, the contents will be entirely liquid; if you allow it to cool uniformly to 183°, the contents will have entirely solidified. For its density, let's say 9 g/cc, since molten lead is 10.7 g/cc, room-temperature lead is 11.3 g/cc, and molten tin is 7.0 g/cc, so this is about 90 g of molten metal, which will yield up its heat of fusion as it solidifies; I don't know what the eutectic's is, but lead's is 4.77 kJ/mol (= 23.0 J/g) and tin's is 7.03 kJ/mol (= 59.2 J/g), so let's suppose the mix is somewhere around 20 J/g; that gives us 1800 J of heat released through this transition. (Plus the small amount of sensible heat from the 60° temperature change, about 0.2 J/g/K (lead being 0.129 and tin being 0.227), adding another 18 J.)

A typical non-temperature-controlled hand soldering iron is 15–30 W, although soldering guns are commonly 150 W. 1800 J at 150 W is 12 seconds of soldering; at 30 W it's 60 seconds of soldering.

So you could heat up such a reservoir (with fire or whatever) and then solder with it for a few seconds to minutes while it stays hot enough to melt tin–lead solder — up to a total mass of solder similar to the 45 g it contains — or a shorter time while it remains hot enough to melt lead-free solder.

Since solidified solder is a much poorer conductor of heat than copper (401 W/m/K for copper, vs. 35.3 W/m/K for lead, 66.8 W/m/K for tin, so solder is probably somewhere in that neighborhood), some copper bars inside the can might be useful in conducting heat from its interior to its surface. In the other direction, some fiberglass or porous ceramic around the can might be useful in preventing heat loss through other than the tip.

How big a reservoir can you reasonably hold?

I have one of those goofy multicolored ballpoint pens, with eight sliders near the back end to select the color. It's comfortable to use, but it's toward the chunky end of what's comfortable; it's about 17 mm in diameter and 130 mm long, and the tip curves down to a cone of about 60° included angle; so its total volume is about 30 milliliters.

Electric soldering irons are normally held much further from the tip, which I assume is because the old thermostat-less type only had its temperature limited by losing heat to the air; if they had only a little area exposed, then they could only use a very small amount of power, and consequently would heat up slowly and require very low duty cycles. But this would seem to be unnecessary with any kind of

thermal regulation. Certainly it is possible to insulate the iron sufficiently that you can grasp it quite near the tip, and this would be advantageous. So this pen is probably a reasonable way to estimate the comfortable volume that can be used for a phase-change soldering iron.

However, if my estimate of solder's density above is in the ballpark, a reservoir of 30 mℓ would weigh some 270 g — in folk units, nearly a pound. Finely manipulating such a heavy reservoir would probably tire the hand before long (I did a brief test with a half-full 600mℓ plastic coke bottle, and found fatigue but no diminution of dexterity), so most likely the reservoir-based iron should be smaller than this.

Phase-change alloy choice

If you only wanted to solder traditional tin–lead solder, maybe instead of a 40–60 tin–lead phase-change reservoir which needs to be heated up to 247° to be fully melted, you should use 70–30 tin–lead, which is fully melted at 193°. This gives you a much more precise temperature, dramatically reducing the maximum temperature the delicate electronic components can reach — but the temperature is a bit low, meaning that heat flow will be slow, and this can actually increase the risk to the components by giving them too much time to heat up.

Better choices for tin–lead soldering might include Sn 89%, Zn 8%, Bi 3% (191°–198°), which has the advantage of being entirely non-precious-metal; Pb 60%, In 40% (195°–225°); the tin–zinc eutectic Sn 91%, Zn 9% (199°); and the near-eutectic ternary alloy Sn 86.9%, In 10%, Ag 3.1% (204°–205°), which is 87% non-precious-metal. (Indium and silver are about the same price.) For lead-free soldering, pure tin (232°, as I said above) or the tin–silver eutectic (3.5% silver, 221°) might be the best phase-change material.

Hot-oil soldering for larger reservoirs and higher powers

As described in Hot oil cutter (p. 3287), you can heat a thin pipe to a consistent high temperature by pumping hot oil through it; as described in Coolants (p. 3235), sunflower oil, glycerin, and mineral oil are all liquid and fairly stable at the temperatures under discussion as well as at room temperature. So you could perhaps maintain a large "solder pot" partly molten, with a thick coolant pipe serpentining through it to carry off the heat of fusion, connected to a thinner coolant pipe to heat the iron tip. However, this is probably more elaborate than a thermostatically-controlled electric tip would be, so probably isn't justified.

Thermal conductivity; a steel can is fine

I suggested above using a thin can of copper to hold the phase-change alloy, but that's probably unnecessary; a thick can of steel would surely work just as well, and would be cheaper, stronger, stiffer, and not at risk of dissolving in the phase-change medium. As mentioned above, copper's thermal conductivity is 401 W/m/K, so sending 150 W through a 400-µm wall with a 20° temperature difference would require a 3.8-millimeter-diameter circle of wall

with an area of 7.5 mm²; this is about the diameter of a soldering tip, so you could reasonably just expose a conical tip of the can as the soldering tip. That's the only part of the can that would need to be copper, though.

A more sensible approach, though, is to use a separate conical solid-copper tip to conduct the heat, which communicates with the can through a much larger contact area, at which point the can itself can be steel — despite its lower thermal conductivity (80.4 W/m/K for iron), the can is no longer contributing the majority of the thermal resistance. This also makes it practical to use the plated copper-bar or plated copper-sheet conductors mentioned earlier to conduct heat out of the center of the can.

Copper bars? Nah, maybe heat pipes?

Earlier I talked about using copper bars to conduct the heat from the midst of the solidifying reservoir to the tip where it is used, because of copper's five-times-higher thermal conductivity. Thinking about conducting heat lengthwise through a long cylinder, if the copper is to conduct more heat than the rest of the reservoir contents — much less *far* more heat — it needs to have at least a fifth of the cross-sectional area. That suggests that, for example, if a cylindrical reservoir is to have a diameter of 15 mm, the copper bar must have a diameter of 6.7 mm. That's not enough to make the idea impractical, but it's a fucking hell of a lot of copper.

Consider, though, the total power we can get. Suppose my reservoir is 100 mm long and 17 mm in diameter, a bit smaller than the multicolored pen mentioned earlier. If nearly all of it is filled with a massive copper bar, the most heat we can get from the middle of it to the end at a ΔT of 20 K is 401 (W/m/K) · 20 K · 2π(8½ mm)² / (50 mm) = 72 W. This is barely adequate, and becomes less so once more realistic amounts of copper are considered.

Materials are known with higher heat conductivity than copper. They are silver, diamond, carbon nanotubes, and graphene. At present none of these is a practicable alternative.

A possible alternative is the "heat pipe", called by Wollaston, its inventor, the "cryophorus". This is an elongated, evacuated chamber, with a little liquid coolant in equilibrium with its vapor, in modern realizations with a wick to carry the liquid throughout the system. When one end of the chamber is cooler than the other, the vapor condenses at that end, lowering the pressure until the liquid vaporizes equally fast at the other; with the wick, this is an endless cycle. Because the vapor carrying the enthalpy of vaporization from one end to the other moves bodily through space, it can travel considerably faster than heat diffusing through a body — the heat transfer power is almost independent of the distance the heat must be transferred.

Most often, heat pipes are used near room temperature and employ water as the coolant; in theory, this works from water's triple point at 0.01° up to its critical point at about 374°, but as I understand it, water is normally only used up to about 200°, perhaps to avoid dangerously high pressures.

Heat pipes don't suffer from the drawbacks of the hot-oil approach mentioned earlier, in that they don't need any pumping and don't contain any moving parts.

Alternatively, perhaps a much shorter, more bulbous heat reservoir

could enable mere copper to deliver the heat adequately.

Topics

An affine-arithmetic database index for rapid historical securities formula queries

Kragen Javier Sitaker, 2019-09-15 (15 minutes)

I started writing a README.md for a project called "affinebase" in 2017, but then never wrote any code for it; this note outlines what I had in mind to implement.

Suppose you want a database of financial tick data that can efficiently evaluate queries for things like "when the price of SPY was at least 1% higher than its 15-minute moving average but lower than its 4-hour moving average", or "when the price of GOOG.A was more than 26 times the price of GOOG.B".

Conventional database indices provide very little help with such queries, but an organization based on affine arithmetic occurs to me as an efficient structure for such things.

Query evaluation with interval-annotated trees over sequences (p. 1423) considers associating an interval-arithmetic interval with substrings of a sequence of records as a form of index; this note considers moving from interval arithmetic to affine arithmetic.

Affine arithmetic

Affine arithmetic is a system of "self-validating arithmetic" similar to interval arithmetic, but supporting linear cancellation of approximation errors.

In affine arithmetic, instead of evaluating expressions to numbers under some assignment of numbers to their free variables, we evaluate them to *affine forms* under some assignment of affine forms to their free variables; these affine forms take the form $k + \Sigma i a_i \varepsilon_i$ and are stored as a vector $[k, a_0, a_1, \ldots a_n]$. The ε_i are gremlin variables that are free to introduce error into your results by taking any value within some fixed range, usually $[-1, 1]$. (The standard term for "gremlin variable" is "error symbol".) Whenever you execute an arithmetic operation that may introduce rounding error, you introduce a new ε_i to account for that rounding error, with an a_i sized appropriately for the computed size of the rounding error.

In its most basic form, this is a somewhat less conservative form of interval arithmetic; in ordinary interval arithmetic, the expression $x - x$ evaluates to some interval around zero whose error is twice the size of the error of x, but in affine arithmetic the errors cancel exactly and you are left with exactly 0, as the SF intended. In general, affine arithmetic can precisely cancel the linearly-varying parts of numerical errors, but nonlinearly-varying parts will be incompletely canceled, so it still provides error bounds that are wider than the real error can be.

The part where this gets interesting is when you assign non-negligible a_i to the free variables. Suppose you want to plot the surface $x^2 + xy + 2$, for example. With affine arithmetic, you can directly evaluate it over a region such as $x \in [-1, 3]$, $y \in [2, 4]$, by assigning $x = 1 + 2\varepsilon_0$, $y = 3 + 1\varepsilon_1$. Depending on the particular evaluation approach, the result will depend not only on ε_0 and ε_1, but

also two to four more ε variables representing the rounding error from the additions and the nonlinearity of the multiplications. If we ignore the rounding errors, this works out to $8 + 10\varepsilon_0 + 1\varepsilon_1 + 2\varepsilon_2 + 2\varepsilon_3$.

For use as self-validating arithmetic, which is to say saving you the trouble of calculating static bounds on your algorithm's approximation errors, you probably want to run each calculation more than once with different floating-point rounding modes by calling the C99 function `fesetround` or something similar: 0 is round-toward-0 and 1 is the default round-to-nearest, but the relevant ones are 2 for toward-positive-infinity and 3 for toward-negative-infinity; with GCC I think you also need to compile with `-frounding-math`. However, in this note, I'm focusing on the non-self-validating-arithmetic uses of affine arithmetic.

Now, the simplest reading of this result $8 + 10\varepsilon_0 + 1\varepsilon_1 + 2\varepsilon_2 + 2\varepsilon_3$ is that if x and y are inside the specified ranges, then the surface will be between the heights of $8 - 10 - 1 - 2 - 2 = -7$ and $8 + 10 + 1 + 2 + 2 = +23$, and this is true. In fact the surface actually does reach +23 at (3, 4), but its lowest point in this range is at (-1, 4), where it reaches -1, so the -7 is a bit conservative. This is in fact precisely the same result that ordinary interval arithmetic would have given us on the factored form $(x + y)x + 2$, but affine arithmetic gave it to us without doing the factoring step.

However, a *much more interesting* reading of this result is to re-express it in terms of x and y. $5x = 5 + 10\varepsilon_0$, so it's $3 + 5x + 1\varepsilon_1 + 2\varepsilon_2 + 2\varepsilon_3$, which is $0 + 5x + y + 2\varepsilon_2 + 2\varepsilon_3$, which is to say, $0 + 5x + y \pm 4$. This gives us *much tighter* bounds on the result: instead of ±15 we have ±4. (They are still conservative bounds, because the function never actually gets below $5x + y - 2¼$, which it reaches at (½, 4).)

So, a very interesting thing we can do here is to start with a large interval of our independent variables and recursively subdivide it to get a piecewise-linear approximation of our function of choice. We can choose whether to subdivide the interval based on criteria such as: the remaining error; for plotting, the geometric angle, in radians, between adjoining line segments; or simply whether it's possible for any point within the interval to satisfy an equation or an inequality — like the inequalities in the example queries at the beginning of this note.

As noted in Affine arithmetic has quadratic convergence when interval arithmetic has linear convergence (p. 1029) and Affine arithmetic optimization (p. 2801), the affine-arithmetic approximation has a higher order of convergence than the interval-arithmetic approximation — as the size of the interval decreases, the error of any regular function diminishes quadratically in the size of the interval with affine arithmetic, but only linearly with interval arithmetic.

In this connection it's worth mentioning "reduced interval arithmetic" in which we restrain the proliferation of the ε_is by introducing an ε_ω not subject to linear cancellation; its coefficient represents the errors proceeding from things like small rounding errors that we don't bother to track separately. This way we can still get the tasty quadratic convergence and even the self-validating property without paying the high cost of an ever-growing flock of ε_i variables on every operation.

Existing work

The above is not original to me; there are at least three papers

describing it, none of which I have managed to finish reading.

These papers are especially relevant to Reduced affine arithmetic raytracer (p. 2007), which is why I was reading them.

Jorge Eliécer FLÓREZ DÍAZ wrote his 2008 dissertation, "Improvements in the Ray Tracing of Implicit Surfaces based on Interval Arithmetic", on using this approach to accelerate the ray-tracing of animated scenes, but using a modal "completion" of ordinary interval arithmetic ("modal interval arithmetic") rather than affine arithmetic. I read someone else's dissertation in French on doing something similar with affine arithmetic, but I can't remember who it was or what it was called.

Knoll, Hijazi, Kensler, Schott, Hansen, and Hagen wrote a paper "Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic" in 2008 describing how to use this approach to accelerate raytracing, including in GPU shaders; they trace the approach back to Toth in 1985 and also cite, among many others, a 2006 paper by Flórez Díaz.

Gamito and Maddock

There's also a paper on the subject by Gamito and Maddock from 2004 or 2005, "Ray Casting Implicit Fractal Surfaces with Reduced Affine Arithmetic"; I think this may be the paper that introduced reduced affine arithmetic.

I think it has a couple of errors in it. On p. 6 equation (13), computing the reduced affine arithmetic product of two variables $\hat{w} = \hat{u}\hat{v}$ represented as three-tuples (center, parametric coefficient, error bound) says $w_2 = |u_0v_2 + v_0u_2| + (|u_1| + |u_2|)\cdot(|v_1| + |v_2|)$, but this can incorrectly cancel the error bound v_2 of $\hat{v}$ against the error bound u_2 of $\hat{u}$ if their signs happen to be opposite, which would be pure happenstance; that first term should be $|u_0v_2| + |v_0u_2|$, not $|u_0v_2 + v_0u_2|$. (This is assuming that it's possible for the error-bound's sign to be negative, which would arise from a direct application of the affine operations in equation (7) on p. 5.)

On p. 5 I think equation (8) gives an avoidably pessimistic bound for the extra error coefficient of a product w_k. It says $w_k = \Sigma_i|u_i| \cdot \Sigma_i|v_i|$, which is safe but unduly pessimistic in the case where the two i variables coincide. $\hat{u} = u_0 + \Sigma_i u_i e_i$ for i > 0 (Gamito and Maddock use e_i rather than the ε_i used above), and similarly for v, and if we take $e_0 = 1$ this simplifies to $\hat{u} = \Sigma_i u_i e_i$. Then $\hat{w} = \Sigma_i\Sigma_j u_i e_i v_j e_j$. So the case $i = j > 0$ corresponds to a term in the fully expanded sum $u_iv_ie_i^2$, and the implicit presumption of that sum is that $e_i^2 \in [-1, 1]$.

This is not *incorrect*, but a tighter and also correct bound is that $e_i^2 \in [0, 1]$; if you take this into account, you need to add $\frac{1}{2}u_iv_i$ to the constant term w_0, so rather than being $w_0 = u_0v_0$, it's $w_0 = u_0v_0 + \frac{1}{2}\Sigma_i u_iv_i$ for $i > 0$; this halved sum is also what you would subtract from w_k. I think. I haven't really tried it, so I might be overlooking an absolute value or something.

Similarly, the "interval optimisation" algorithm they give in §4.3 and figure 2 on pp. 6–7 is not *wrong* but it is *suboptimal* — they had the brilliant idea of using the affine form to tighten the interval where they're trying to find the root, which is the whole thing that Fast mathematical optimization with affine arithmetic (p. 3163) is about, but then they wastefully divide the interval in half even if the affine-arithmetic-based tightening was very successful, guaranteeing

an additional time through the loop and perhaps even an additional branch to recurse down.

Also they misspelled "Lipschitz" as "Lipchitz".

A univariate affine-arithmetic database

So, if we have an affine form that summarizes a time-varying quantity, such as a stock price, in the form $k + a_0t + a_1\varepsilon_1$, for some interval, where ε_1 is a bound on the error of the linear approximation over that interval, then we can efficiently compute some bounds on the kinds of expressions in the introduction, and efficiently reject huge swaths of history at once as not meeting our query condition, and efficiently accept other huge swaths of history as always meeting it. For intervals where the condition may possibly be met, we can recursively subdivide them into smaller intervals with tighter error bounds.

But where do we get these affine forms? They already exist in the technical analysis of securities prices, where they go by the name of "channels" — a "channel" is a linear approximation of a securities price over some period of time within some error bounds. Although there is no guarantee of this, it is typically very efficient to compute the best channel for a given time period by computing the convex hull of the prices over that time period, which takes linear time, and then considering the slopes of the line segments of that convex hull; the best channel slope will be one of these, and it suffices to consider the points on the convex hull. In theory there could be a linearly large number of line segments on both the upper and lower convex hull, but in practice the number is much smaller.

You can subdivide history at arbitrary points and compute the best channel for those arbitrary intervals, but you can get much tighter channel bounds if you instead look for "natural" points to make the divisions. The points on the convex hull are promising candidates for "natural" division points, since the largest local extrema of oscillation from the trend line will necessarily be part of the convex hull, but I think there's a linear-time algorithm to find the "most natural" division point.

If you use FP-persistent stack structures to build the convex hull using the convex-hull algorithm mentioned in Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216), you compute the convex hull not only of the whole interval but also every prefix of the interval in a single linear-time, linear-space pass. Doing this once in each direction allows you to evaluate every possible division point within the interval without redundantly recomputing those convex hulls.

In this way you can build a tree over time that permits rapid branch-and-bound evaluation of ad-hoc historical queries on arbitrary computable inequalities.

A multivariate affine-arithmetic database using PCA

Simply approximating security prices or other time series as linear functions of time plus guaranteed error bounds does allow you to compute things like ratios and differences between them efficiently with guaranteed error bounds. However, it's very common — not to

say nearly universal — for securities prices to have correlations, negative or positive, that go beyond a simple linear trend over some time period, and if you can take these correlations into account, you may be able to get much tighter error bounds.

One possible way to do this is to run a principal components analysis over historical prices, and then store the time series of several principal components in your database alongside the actual securities prices. Then you can summarize each interval in the tree of a security as a linear function not merely of time but also of these principal components. This should permit much tighter bounds on the results of the arbitrary expressions over long time intervals, thus permitting much faster branch-and-bound evaluations.

Non-market applications

This technique, of course, is applicable to any time-series quantitative data, not just securities prices — market prices of commodities, temperatures and other climate data, system administration metrics such as network traffic and error rates, telemetry data from satellites and space probes, audio signals, image data (with two "time" dimensions), and so on.

Topics

When should you give up waiting for the bus and just walk?

Kragen Javier Sitaker, 2019-04-24 (5 minutes)

Suppose you're waiting for the bus on a Sunday. You know that the bus goes where you want to go, and it will get you there in five minutes. But you may not be sure when or how often the bus runs on Sundays. You can walk where you want to go, but it will take you 50 minutes. At some point, if the bus hasn't come yet, you'll give up and start walking. You would like to arrive as soon as possible. How long is it optimal to wait before you start walking?

(To simplify the problem I'm ignoring many complexities of the real-world situation; for example, you might get lost walking, or fall off a bridge and die; or, if the bus comes while you're walking, you might be able to run back to the bus stop, or to the next bus stop; or the bus might catch on fire while you're on it, or take a detour that takes you further from your desired destination; you might be hallucinating the destination, or waiting at the wrong bus stop, or indeed be an Alzheimer's patient in a nursing home whose belief in both the bus and the destination is a demented delusion; you may be dreaming, and thus able to cause the bus to arrive whenever you want; and so on.)

This problem depends on two parameters: one is your prior probability distribution of bus arrival times, and the other is your utility function over destination arrival time distributions. There are some simple cases that are easy to solve. For example, if your probability distribution is just a Dirac delta at some time t_0, which is to say that you know for certain when the bus will arrive, then you should wait if $t_0 < 45$ minutes, walk immediately if $t_0 > 45$ minutes, and do as you like if $t_0 = 45$ minutes. If you absolutely need to arrive within 51 minutes, perhaps for a court date to appeal your upcoming execution, you should walk if there is any possibility of the bus taking longer than 45 minutes to arrive. (Of course, in reality, there is always some possibility of that.)

One solution which is robust (that is to say, it's not optimal, but the optimal solution can only beat it by some reasonably small margin) over wide ranges of these parameters is to wait 45 minutes. This bounds your destination arrival time to 95 minutes, which is less than a factor of 2 worse than the best result you can guarantee (if you just start walking immediately), and it has a "regret property" that is also bounded to a factor of less than 2: if you arrive at 95 minutes, the soonest you could have arrived if you'd waited longer would have been 50 minutes, which is better by less than a factor of 2. I want to make some kind of argument based on Lipschitz constants of utility functions here, but I'm not certain how to formulate it yet.

This problem is closely analogous to a wide range of planning problems in domains with large variance.

For example, suppose you want to write a video-codec subroutine for 60-frame-per-second video, and you are trying to choose whether to write it in C or Python. You know that you can write it in C, and if you write it in C, it will be plenty fast, but writing things in C takes

longer — say, 500 minutes. You know you can write it in Python in about 50 minutes, but you don't know if it will be too slow to use. (If it ever takes more than 16.7 milliseconds to run, it will make the video playback skip frames.) How long should you fiddle around with trying to optimize the Python version to be fast enough before you throw it away and write the C version?

In an analogous way, if you only have 500 minutes to solve the problem, you should just "walk", writing the C version to begin with. But if you can afford to take a bit longer — say, 600 minutes — you should try writing the Python version and see if you can get it to run fast enough. Maybe you'll finish the task in 50 or 100 or 150 minutes instead of 500 minutes. But by trying that option, you're taking the risk that the overall task might extend to 550 or 600 or 650 minutes, if the Python approach doesn't work out.

This simplified model highlights one reason that it is so costly to maximize the predictability of processes for, among other things, writing software. The fastest and highest-quality process is only very rarely the most predictable one.

Schedule estimation errors are widely observed to be lognormally distributed rather than normally distributed. This is not the only reason, but it is a significant one.

Topics

Backwards cockcroft walton

Kragen Javier Sitaker, 2019-12-01 (2 minutes)

I was thinking about capacitive-dropper power supplies, which limit the current from the 120V ac or 240V ac powerline with capacitive reactance (see Capacitive droppers and transformerless power supplies (p. 887) for more.). These can only supply a very small current without using very beefy capacitors and being quite hazardous. And it has to drop a lot of voltage through those capacitors before it reaches the load, if the load is something like a 5V or 3.3V electronic device.

In Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) I concluded that a Cockcroft–Walton generator could be used as a sort of variable-ratio rectifying autotransformer, stepping up the output dc voltage from a fixed ac input level to a variable level determined by the load voltage. But if you try to use it to get a ratio of less than 1, it won't work; in the limit of low output impedance, you just get the input ac voltage at the output plus a dc offset that just keeps it from going negative.

Is there a Cockcroft–Walton-like circuit that works for stepping voltages *down*? Because then you could use a capacitive dropper to reduce the input voltage to, say, 48 V, limited to 30 mA, "galvanically isolated" from the input powerline by the capacitors, and then use the Notlaw–Tforckcoc circuit to step that down to something like 5 V at 280 mA, enough for a standard USB1 charger.

The way the Cockcroft–Walton circuit works, from a certain point of view, is that the diodes in its string are in parallel across the ac input (since the capacitors act somewhat like wires at ac, albeit wires with some reactance), but in series across the dc output. In this case what we want is the opposite: for the diodes to be effectively in series across the ac input while being in parallel across the dc output.

It isn't yet obvious to me how to do this (without active control circuitry, which can of course do this by switching capacitors around using MOSFETs), but I suspect there might be a way.

Topics

• Electronics (p. 3430) (138 notes)

Profile-guided parser optimization should enable parsing of gigabytes per second

Kragen Javier Sitaker, 2019-05-23 (8 minutes)

It should be possible to make parsing nearly always as fast as KMP string search, or even faster, without even resorting to SIMD.

parselov

As mentioned in Parallel DFA execution (p. 2337):

> Bjoern Hoehrmann has been working on an algorithm called "parselov" which compiles a context-free grammar to a finite-state automaton that approximates the CFG using a limited-depth parse stack (one stack item, I think).

The idea, as I understand it, is that you can parse a deterministic CFG with a deterministic pushdown automaton ("DPDA"), which can be analyzed as a finite state machine augmented with an unlimited-depth stack. It follows that you can parse a limited-stack-depth approximation of a context-free grammar with a finite state machine, since there are a finite number of stack states with N items or less. (Evidence from human errors in speaking natural languages, and from the verisimilitude of natural language generated by LSTMs and similar neural networks, suggests that in fact this is what we do most of the time.)

The potentially exponentially large number of stack states suggests that enumerating them for any reasonable depth might be infeasible, but typically the stack states are strongly dependent on one another; in most languages, for example, you can't be parsing a formal parameter list unless you're just inside a function declaration, and you can't be parsing a string unless you're just inside an expression. So the base of the exponential might be surprisingly low.

But what happens if your pushdown automaton wants to push an (N+1)th item on the stack? Of course, you could decide to abort, and if your stack depth was deep enough, that might be fine. But, as an alternative, you could shift the N-1 items on the stack over by one and enter an "overdepth" state. Then, when popping, you can deterministically reduce the number of stack items in your finite state while keeping the overdepth flag, until you reach some low water mark of such stack items; then you add nondeterministic transitions to unshift each of the possible next stack items, and nondeterministically clear the overdepth flag or not.

If the distance between the low water mark and N, the high water mark, is large enough, then only a tiny fraction of the tokens or characters input will result in a nondeterministic transition.

(The low-water mark could be as low as 0, but in that case, the state you could have to shift back in could be any arbitrary state. If it's at least 1, you have some context that reduces the number of possible transitions.)

By itself, this gives you a nondeterministic finite state automaton which functions as a sort of amnesiac pushdown automaton, accepting a regular language that is a strict superset of your original context-free

language. But, when implementing this, you can add an extra "spill stack" in memory, which precisely tracks the items shifted out of the finite stack window. Then, when doing an unshifting transition, you can pop the spill stack to see exactly which state to transition to.

Thus you have transformed a DPDA into another equivalent DPDA with many more states and much less frequent pushes and pops — most of the time it is just a DFA which occasionally pushes things, but occasionally it pops and consults the popped value, as well as the input, to determine its next transition. The transformation is parameterized by two numbers, the low-water mark and N, the high-water mark. If they are both 0, it's the identity transformation on DPDAs; if they are both 1, as I understand it, it's parselov.

(A precisely analogous transformation can transform a nondeterministic PDA into another, similarly more efficient, nondeterministic PDA.)

But we have more degrees of freedom available for optimization than that; we could vary those numbers depending on the stack situation. In particular, we would like to avoid stack activity in situations that occur frequently, but we don't care about avoiding stack activity in situations that occur rarely; instead we would like to avoid state proliferation.

But grammars don't carry any information about the relative frequency of different situations.

Profile-guided optimization

PGO is a technique used to enable modern AOT compilers to compete with JIT compilers. You compile your program with profiling turned on, run it for a while, and then compile it a second time using the profiling information to guide the optimization, so that it optimizes things that are important instead of things that aren't.

In this particular case, the only thing you would need to produce from "profiling" is the sequence of inputs and state and stack changes in your DPDA as it parses some sentences. Then, given a candidate set of sets-of-top-few-stack-items to assign to finite states to avoid pushes and pops, and some kind of description of the low-water-mark policy, it is straightforward to compute the number of pushes and pops that remain, and which states they are from; a greedy optimization process should do a reasonable and perhaps optimal job of choosing where to merge states and where to split them.

However, this PGO should enable us to not just equal lex's performance, but exceed it substantially, because, on modern processors, the cost of indirecting through a jump table is greater than the cost of a comparison tree in nearly all cases — but the comparison tree's performance can vary substantially depending on how it's laid out. So, by taking advantage of statistics about which characters are encountered in which states in practice, we should be able to parse context-free grammars at gigabytes per second on modern CPUs.

Eliminating caching of parse results

If that works out, it should have substantial implications for the architecture of software, because a substantial fraction of modern software is dedicated to caching the results of parsing. Python's .pyc bytecode files, the Emacs-Lisp .elc they aped, Java .class files, minified JS, the browser DOM, and all kinds of CSV storage things exist

mostly to avoid slowly reparsing things that have already been parsed. I'm typing this in Emacs, which has all kinds of stupid hacks to avoid reparsing the buffer for syntax highlighting until there's some idle time.

If you can parse at gigabytes per second, then in many cases it isn't useful to store those parse results. If Emacs reparses the buffer from the beginning after every keystroke for syntax-highlighting, that will only begin to impede its responsiveness once the buffer is several megabytes in size; if it checkpoints the parse state every megabyte, there is no danger. Similarly, you could run your database queries directly on the CSV rather than importing it into the database. (None of these examples are pure, though: indexing a column of ten million numbers so you can join it with another table would still take an appreciable fraction of a second, even if your CSV parsing were instant, and both Java and Elisp do substantial optimization before writing their bytecode. But maybe they shouldn't.)

And storing the parse results *is* potentially harmful, since in the absence of a general computation-result-caching system like A minimal dependency processing system (p. 911), Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870), or Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257), or at least the kind of reliable filesystem change detection discussed in Immutability-based filesystems: interfaces, problems, and benefits (p. 1672), there's a constant risk of that preparsed data being outdated with respect to its original source.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Caching (p. 3361) (25 notes)
- Parsing (p. 3618) (15 notes)
- Automata theory (p. 3335) (11 notes)
- Parselov (p. 3617) (3 notes)

Foil origami robots

Kragen Javier Sitaker, 2019-06-13 (updated 2019-06-14) (10 minutes)

Doing the crude experiments documented in Single-point incremental forming of aluminum foil (p. 769), I was surprised to learn how thin aluminum foil is — in the neighborhood of 10 μm, though heavy-duty foil is 25 μm. I had thought it was considerably thicker because of its strength and tendency to retain creases.

So it occurred to me that maybe cut, laminated, and creased aluminum foil, or for that matter steel foil, was a potentially useful material for self-replicating machinery, along the lines of cardboard furniture. The fact that aluminum foil is so thin means that you can increase its rigidity immensely by forming it into relatively small parts, which would then have the stiffness to easily bend the still-flat foil.

Stiffening by forming

Consider, for example, forming a corrugated sheet from aluminum foil similar to the corrugated iron sheets commonly used for the roofing of military bases and other slums. If you make it 10 mm thick, then you have about ¼ of the foil resisting tension on the outside of the bend with a lever arm of 5 mm, ¼ of it resisting compression on the inside in the same way, and the other ½ resisting tension and compression with an average lever arm of 2½ mm; the average lever arm is then 3¾ mm, while in the untouched sheet, the average lever arm might be 2½ μm. So the corrugated sheet is on the order of 1500 times stiffer. It's fucking magic.

(There's an additional work-hardening factor if you're working with ordinary aluminum foil from the grocery stre, which is annealed, rather than whatever comes out of your aluminum-foil-making machine on the moon or whatever.)

That also works for increasing flexural strength (in the sense of the bending force on a member needed to provoke plastic deformation, not the stricter sense of the tensile stress in the fibers of the material needed to provoke plastic deformation) and resistance to buckling, but there's no such magic for increasing tensile strength — although perhaps you could use the folded aluminum as a mold to be filled with some other material.

Origami

There's a bunch of work in computational origami for things like unfolding satellite space shades, which unfortunately I don't know anything about. Robert Lang is the big name in computational origami, and satellite space shades are folded with the Miura fold. My level of origami is basically that I can fold an origami crane, so I did that using this aluminum foil. It wrinkled considerably more than paper does, but it also creases better. To get the square to fold it from, I cut along the edge using my zirconia knife, a process which required only the tip of the blade, the last 30 μm or so — so presumably you could do the same thing with a 30-μm-long blade, which wouldn't require very much sharpened zirconia or alumina or similar material.

(At some point I had dropped the knife on its tip and chipped it, so

last week I bought a 750-grit diamond hone and sharpened the tip on it with some dish detergent.)

There's a "Handbook of Compliant Mechanisms" from 2013 out of the BYU flexures research group, published by Wiley; nearly half of it is a rather poor-quality "library of compliant mechanisms", much of which consists of things that can be cut out of a sheet but then flex into a three-dimensional shape (to which they have given the name "lamina emergent mechanisms"). The book doesn't mention origami at all, and much of the book focuses on techniques that are difficult or impossible to apply to origami. To me it seems clear that origami is an important technique for flexure fabrication, and it turns out that Magleby and Howell, two of the editors of the "Handbook", published a couple of papers on this, even before the Handbook came out. So I'm not sure why they didn't include anything about this in the book.

Some traditional origami forms, like the flapping-wing bird and the jumping frog, are designed as flexures; these are called "action origami" or sometimes "kinetic origami" (a term due to Magleby and/or his coauthors in a 2011 paper). Papers typically analyze these as rigid flat panels connected by flexible hinges at the folds, a model which seems unmotivated by physical considerations — in paper typically the folds are slightly less rigid than the panels in the "hinge" direction, but in aluminum foil, they are typically slightly more rigid.

Many of the annoying features of aluminum, such as its high cost, large springback, abrasive nature, and tendency to accumulate internal stresses during heating that produce delayed distortions, are less important in this situation.

Because the aluminum is about ten times thinner than paper, but only 2.7 g/cc (according to Compressed sensing microscope (p. 306)), this aluminum is about a third as heavy as paper, square millimeter for square millimeter. Consequently the crane is much lighter than a paper crane of the same size would be, rather astoundingly light. Unfortunately, it isn't very sturdy; I can plastically deform it by blowing on it. Work-hardened (unannealed) foil or a more reinforced design might help with this.

Foil properties, and comparison to paper

The foil is really amazingly flexible for its strength. According to the Wikipedia tensile-strength article, annealed aluminum has a yield strength in the neighborhod of 15–20 MPa, a Young's modulus of 70 GPa, and an ultimate strength ("engineering", I suppose, calculated according to the original material thickness) of 40–50 MPa, from which we can deduce that its plastic strain is about 0.2%–0.3%, while its yield strain is about 0.6%–0.7% (engineering, I suppose). So plastically creasing the foil involves bending it at a radius such that the inner surface is 0.2%–0.3% shorter than the centerline, and the outer surface is 0.2%–0.3% longer, so a bend radius of 1.7 to 2.5 mm.

The same article, though, points out that aluminum alloys are immensely stronger: 414 MPa yield and 483 MPa ultimate for 2014-T6 and 248 MPa ultimate for 6063-T6 (a tempered grade of a general-purpose "wrought" alloy), though their elastic modulus is about the same. The very common 6061-T6 is 275 MPa yield and 310 MPa ultimate, with 69 GPa Young's modulus, roughly the same. These alloys are precipitation-hardened; annealed 6061 ("6061-O")

has only about 55 MPa of yield strength and only about 125 MPa of ultimate tensile strength, but elongation of 25%–30%, so it might make sense to perform the origami on the annealed material and then heat-treat the finished product to precipitation-harden it, gaining an additional factor of 5 in resistance to deformation.

How does it compare to paper? According to UHMWPE clothes could be lightweight and sturdy (p. 3071), the tensile strength of cellulose is in the neighborhood of 40 MPa, but when paper tears, it commonly tears because the different cellulose fibers have come apart, not because all the fibers are failing at the crack, as when you cut it with scissors; the tear tends to rotate into an orientation near parallel to the surface as it propagates in order to break even less fibers, even though this spreads the failure over a larger surface. This is also why cotton paper like that used in dollar bills is harder to tear. Unfortunately, this leaves me little wiser!

Cellulose at room temperature is normally thought of as a brittle material, one which fails without an intermediate plastic-deformation stage, and indeed when you tear paper there isn't a noticeable stretched-paper area at the tip of the tear, nor does the edge formed by the tear crinkle in paper the way it does in aluminum foil. But this would predict that creasing paper should be impossible — like polyester napkins, the paper should just elastically return to its original form unless fibers were actually broken. Fibers *are* actually broken, as evidenced by the tendency of a tear to follow even a simple paper crease, but by itself that doesn't explain the tendency of paper to elastically return to the creased orientation — it would only explain its tendency to be more flexible at the crease.

At first, I thought we couldn't explain paper's tendency to hold creases through some kind of non-cellulose interaction, because cellophane holds creases too, and I thought cellophane was pure cellulose; but it turns out that cellophane also contains glycerin as a plasticizer. Presumably this very plasticity is what allows it to hold creases.

This suggests that paper holds creases by altering the interactions of cellulose fibers. On the inner radius of the crease, there is presumably some thickness of the paper that is crushed into rubble, around which is wrapped a series of layers of cellulose fibers loosely stuck together, either by other materials present in the paper or by interactions among fibers that touch one another. The fibers in these layers slide past one another and dig into inner layers during the creasing process, and perhaps in the outermost layers are broken by the tension. This leaves the natural length of the layers altered, causing the paper to elastically maintain the crease. This hypothesis is Original Research™ and therefore may be entirely wrong, but it gives me a reasonable alternative to supposing that the cellulose is deforming plastically.

Topics

- Flexures (p. 3456) (3 notes)
- Origami
- Aluminum

A sentence-granularity hypertext editor

Kragen Javier Sitaker, 2018-04-27 (4 minutes)

I've been thinking about doing storage and communication in terms of small, atomic "grains", in the range of 64–2048 bytes. In particular I was thinking it would be interesting to structure a hypertext system in terms of such grains, such that you have a few grains on the screen at a time. Grains can include "implicit links" to other grains, and when you change focus to a new grain, the system tries to change the layout so the implicitly-linked grains are also on the screen, maybe hiding grains that have been less recently used in order to make room.

On a cellphone screen you have room for somewhere around 2048–4096 characters of text at a time comfortably, and that text may be referring to equations, graphs, drawings, schematics, computational models, or other parts of the text. If we display 8–16 grains at a time, and each grain has implicit links to the things it refers to (as well as the previous and next grains in the text), we should be able to always have all the referents of the focused grain displayed all the time, as long as that grain doesn't have more than 7–15 referents, plus a maximum amount of context. A simple LRU policy should be adequate to choose which grains to collapse (or scroll off the top, or whatever) to make room.

This suggests somewhere on the order of 128 to 512 displayable characters per textual grain, about a sentence, or a few tens of thousands of pixels (which, for raster graphics, is one impedance mismatch between this UI thing and the packet IPC systems I'd been thinking of).

What would an editor for this structure look like? If you have a petabyte of mostly-ASCII text in your hypertext, broken down into 128-character grains, that's 2^{33} grains, so an unambiguous grain number for each (mutable) grain would require almost exactly 10 decimal digits: "3804448033". Using letters shortens the identifiers to 7–8 characters, or with alternating numbers and letters (as in Augment, sort of) 8–9. A 512-byte grain would have room for about 64 grain identifiers in links, so a full B-tree would be about 5 or 6 layers deep.

You could use this structure directly for your editor's data structures; ideally you'd have keys to merge two consecutive grains and break the current grain at a point (into two grains linked with implicit prev and next links), and maybe some kind of UI affordance to encourage you to press the split key once a grain got uncomfortably large. You really only need to keep in memory the grains that are currently on the screen; you can flush the updated grains out to a transaction log (or sandpile?) as they are evicted. For many purposes, a search function that only finds strings within a single grain would be adequate, as searches for strings that cross sentences are rare, and could perhaps be accommodated with some kind of graph search.

In addition to implicit links, of course, you'd want explicit links

which only activate when clicked on.

A particularly useful kind of enhancement for such a system would be the ability to pass parameters to grains which they use to run some kind of code to render a template for display — potentially the identifiers of other grains. In this way, for example, you could instantiate a pipe flow formula in a hypertext with the particular parameters of a pipe you were considering.

Topics

• Hypertext (p. 3512) (13 notes)
• Editors (p. 3426) (13 notes)
• Granular hypertext (p. 3482) (3 notes)

Academic lineage

Kragen Javier Sitaker, 2016-10-30 (updated 2019-11-24) (15 minutes)

I spent some time tracing academic lineages, helped by the Mathematics Genealogy Project. It traces 132,301 mathematicians [2019-09-05 update: now 246,469], most of whom are still alive, back to a 13th-century astronomer named Shams ad-Din al-Bukhari or Shams al-Dīn al-Bukhārī, who enabled Gregory Chionades to obtain Greek translations of the astronomical handbooks in circulation in the Islamic world.

Mathematical lineages

From al-Bukhārī to Gauss

One path through 22 generations is as follows:

- Shams ad-Din al-Bukhārī
- Gregory Chioniadis
- Manuel Bryennios (aka Μανουήλ Βρυέννιος, Constantinople)
- Theodore Metochites (aka Θεόδωρος Μετοχίτης, 1315, Constantinople)
- Gregory Palamas (aka Γρηγόριος Παλαμάς, born in Constantinople and archbishop of Thessaloniki)
- Nilos Kabasilas (aka Νεῖλος Καβάσιλας, 1363)
- Demetrios Kydones (aka Δημήτριος Κυδώνης, Thessalonica, three-term Mesazon of Byzantium; 6 generations from al-Bukhārī)
- Georgios Plethon Gemistos, (1380; the MGP gives "Nómoi" as his "dissertation") usually called Gemistus Pletho and sometimes called *The Last of the Hellenes*
- Basilios Bessarion (1436; 8 generations from al-Bukhārī)
- Johannes Argyropoulos (Padova, 1444), who also taught Leonardo da Vinci
- Johannes "Kapnion" or "Capnion" Reuchlin (Basel, 1477), a Catholic who campaigned against the burning of the Jewish books
- Philipp Melanchthon (Heidelberg, 1511), the Lutheran theologian who wrote the *Augsburg Confession*
- Johannes Caselius (Halle-Wittenberg, 1560; Leipzig, 1566)
- Georg Calixt (Helmstedt, 1607)
- Johann Andreas Quenstedt (Helmstedt, 1643, De Transsvbstantiatione Contra Pontificios Exercitatio)
- Michael Walther, Jr (Halle-Wittenberg, 1661, Manichaeismi recensio historica; Disputatio theologica inauguralis de Paulina Petri increpatione)
- Johann Pasch (Halle-Wittenberg, 1683, Conjunctiones in genere dissertatione astronomico-theorica)
- Johann Andreas Planer (Halle-Wittenberg, 1686, Gynaeceum Doctum, sive Dissertatio Historico-literaria)
- Christian August Hausen (Halle-Wittenberg, 1713, De corpore scissuris figurisque non cruetando ductu)
- Abraham Gotthelf Kästner (Leipzig, 1739, Theoria radicum in aequationibus)
- Johann Friedrich Pfaff (Göttingen, 1786, Commentatio de ortibus et

occasibus siderum apud auctores classicos commemoratis; 21 generations from al-Bukhārī)
• Carl Friedrich Gauß (Helmstedt, 1799, Demonstratio nova theorematis omnem functionem algebraicam rationalem integram unius variabilis in factores reales primi vel secundi gradus resolvi posse)

Euler's lineage

There's also a path to Euler that diverges in the 14th century via Erasmus from Kydones:

• Kydones (6 generations from al-Bukhārī)
• Manuel Chrysoloras
• Guarino da Verona (1408)
• Vittorino da Feltre (Padova, 1416);
• Theodoros Gazes (Mantova and Constantinople, 1433);
• Rudolf Agricola (Ferrara, 1478);
• Alexander Hegius (1474);
• Desiderius Erasmus (Montaigu, 1497/1506; Turin, 1506; 13 generations from al-Bukhārī);
• Wolfgang Fabricius Capito (Freiburg im Breisgau, 1515);
• Simon Sulzer (Strasbourg, 1531);
• Johann Jacob Grynaeus (Basel, 1559);
• Sebastian Beck (Basel, 1610, Illustre Axioma, Ivstvs Avtem Fide Sva Vivet);
• Theodor Zwinger, Jr. (Basel, 1630, De Illustri Sententia Apostolica Hebr. c. 13. V. 8);
• Peter Werenfels (Basel, 1649, Diatribe In Psalmum S. S. Psalterii Primum. De Vnica Et Vera Hominis Felicitate);
• Jacob Bernoulli (Basel, 1676, Primi et Secundi Adami Collatio);
• Johann Bernoulli (Basel, 1690, Dissertatio de effervescentia et fermentatione; Basel, 1694, Dissertatio Inauguralis Physico-Anatomica de Motu Musculorum);
• Leonhard Euler (Basel, 1726, Dissertatio physica de sono; 21 generations from al-Bukhārī).

Nearly all modern mathematicians can trace their lineage to both Gauss and (weakly) Euler, and indeed a quarter of them can be traced back to Felix Klein, who can be traced back to both Euler (weakly) and Gauss.

Tarski descends from Kant and Huygens

For personal reasons, I'm particularly interested in Tarski's lineage, which does trace back to al-Bukhārī, but not via Gauss or Euler; it is a very distinguished line that runs as follows:

• Erasmus, as above for Euler (13 generations from al-Bukhārī);
• Jakob Milich (Freiburg im Breisgau, 1520, later Wien, 1524);
• Erasmus Reinhold (Halle-Wittenberg, 1535);
• Valentine Naibod (Halle-Wittenberg and Erfurt);
• Rudolph (Snel van Royen) Snellius (Heidelberg and Köln, 1572);
• Willebrord (Snel van Royen) Snellius (Leiden, 1607);
• Jacobus Golius (Leiden, 1621), advisor of Descartes;
• Frans van Schooten, Jr. (Leiden, 1635), also student of Mersenne, Descartes's correspondent;

• Christiaan Huygens (Leiden, 1647);
• Gottfried Wilhelm Leibniz (Leipzig, 1666, Disputatio arithmetica de complexionibus), from whom most living mathematicians descend via his other student Nicolas Malebranche; 22 generations from al-Bukhārī via this path, but see below for a shorter path;
• Christian M. von Wolff (Leipzig, 1703, Philosophia practica universalis, methodo mathematica conscripta);
• Martin Knutzen (Königsberg, 1732);
• Immanuel Kant (Königsberg, 1770, Meditationum quarundam de igne succincta delineatio; Principiorum primorum cognitionis metaphysicae nova dilucidatio; 25 generations from al-Bukhārī);
• Karl Reinhold (Jena, 1787, Briefe über die Kantische Philosophie);
• Friedrich Adolf Trendelenburg (Berlin, 1826, Platonis de ideis et numeris doctrina ex Aristotele illustrata);
• Franz Clemens Brentano (Tübingen, 1862, Von der mannigfachen Bedeutung des Seienden nach Aristoteles);
• Kazimierz Twardowski (Wien, 1891/1892, Idee und Perzeption ("Idea and Perception")—An Epistemological Investigation of Descartes) who also advised Banach;
• Stanislaw Lesniewski (Lwów, 1912, A Contribution To Analysis Of Existential Propositions);
• Alfred Tarski (Warsaw, 1924, O wyrazie pierwotnym logistyki).

This puts Tarski only 31 generations from al-Bukhārī via Kant.

Tarski and Leibniz from Pacioli and Bessarion via Copernicus

I've also found some other paths from Tarski back to al-Bukhārī, but most of the others aren't nearly as spectacular. There's an interesting side path, though:

• Bessarion, as in Gauß's genealogy (8 generations from al-Bukhārī);
• Johannes Müller Regiomontanus (Leipzig and Wien, 1457);
• Domenico Maria Novara da Ferrara (Firenze, 1483) who also studied under Pacioli;
• Nicolaus (Mikołaj Kopernik) Copernicus (Padova and Ferrara, 1499);
• Georg Joachim von Leuchen Rheticus (Halle-Wittenberg, 1535);
• Moritz Valentin Steinmetz (Leipzig, 1567, De Peste Capita Disputationis Ordinariae);
• Christoph Meurer (Leipzig, 1582, De Iride seu Arcu coelesti);
• Philipp Müller (Leipzig, 1604);
• Erhard Weigel (Leipzig, 1650, De ascensionibus et descensionibus astronomicis dissertatio);
• Leibniz (17 generations from al-Bukhārī).

This reduces Tarski to 26 generations from al-Bukhārī.

Sierpiński

Wacław Sierpiński is a particularly interesting node in the graph. He doesn't descend from Gauss or, except via Lagrange, from Euler; but he has a significant number of descendants today (about as many as Euler, discounting Lagrange), and a very distinguished line of descent indeed, one which traces back to Gauss's advisor Pfaff and to d'Alembert. The Pfaff line:

• Pfaff (21 generation from al-Bukhārī);
• Abraham Gotthelf Kästner (Leipzig, 1739, Theoria radicum in aequationibus)
• Georg Christoph Lichtenberg (Göttingen, 1765), with 70,574 descendants;
• Johann Martin Christian Bartels (Jena, 1799, Elementa calculi variationum), who also studied under Kästner and Pfaff;
• Nikolai Ivanovich Lobachevsky (Kazan), Bartels's only known student;
• Nikolai Dmitrievich Brashman (Kazan, also Moscow, 1834);
• Pafnuty Lvovich Chebyshev (St. Petersburg, 1849, On integration by means of logarithms);
• Andrei Andreyevich Markov (St. Petersburg, 1884, On certain applications of continued fractions);
• Georgy Fedoseevich Voronoy (St. Petersburg, 1896, On a generalization of the algorithm of continued fractions (Ob odnom obobshchenii algorifma nepreryvnykh drobei));
• Sierpiński (Jagiellonian University, 1906; 30 generations from al-Bukhārī).

This includes Lobachevsky, who with Riemann revolutionized geometry; Chebyshev, the crippled Tatar who revolutionized probability and polynomial function approximation and who taught Lyapunov; Markov, who created our modern theory of discrete dynamic processes; and Voronoy, the sickly Ukrainian whose "Voronoi diagram" underlies an enormous number of modern geometrical algorithms, and who brought the world-shaking St. Petersburg tradition to Poland.

This gives a path from al-Bukhārī to Sierpiński over 30 generations.

But Voronoy was not Sierpiński's only advisor, and Sierpiński's other lineage is no less distinguished for originating *sui generis* in France and Italy without a known earlier academic line of descent:

• Jean le Rond d'Alembert (Collège Mazarin of the University of Paris, 1735, no known advisor);
• Pierre-Simon Laplace (Caen, 1769, Recherches sur le calcul integral aux differences infiniment petites et aux differences finies);
• Siméon Denis Poisson (École Polytechnique, 1800), who also studied under Lagrange;
• Michel Chasles (École Polytechnique, 1814);
• Gaston Darboux (École Normale Supérieure Paris, 1866);
• Charles Emile Picard (École Normale Supérieure Paris, 1877, Applications des complexes lineaires a l'etude des surfaces et des courbes gauches);
• Stanislav Daremba (Sorbonne, 1889, Sur un probleme concernant l'etat calorifique d'un corps solide homogene indefini), who also studied under Darboux;
• Sierpiński.

We are some 34 generations from al-Bukhārī today

Consider a relatively arbitrary modern scholar, chosen not because she is world-famous but just because I've met her here at the University of Buenos Aires, Sandra Martínez, who descends from both Hilbert and Sierpiński, and is thus 34 generations from

al-Bukhārī:

• Sierpiński, with 5329 descendants (Jagiellonian; 34 generations from al-Bukhārī);
• Stefan Mazurkiewicz (Lwów, same as Lesniewski above);
• Aleksander Michał Rajchman (Warsaw, 1921) (also a student of Hugo Dyonizy Steinhaus; see below);
• Antoni Zygmund (Warsaw, 1923; also directly a student of Mazurkiewicz);
• Eugene Barry Fabes (Chicago, 1965);
• Julio Esteban Bouillet, with 19 descendants (U Minn, 1972);
• Noemí Irene Wolanski (UBA, 1983);
• Sandra Rita Martínez (UBA, 2007).

Rajchman is the link to Hilbert, and thence to Klein and thus Gauß and Euler:

• Gauß (22 generations from al-Bukhārī);
• Christian Ludwig Gerling (Göttingen, 1812);
• Julius Plücker (Marburg, 1823);
• Klein (Bonn, 1868; 25 generations from al-Bukhārī);
• Carl Louis Ferdinand Lindemann (Erlangen–Nürnberg 1873);
• Hilbert, with 2606 descendants (Königsberg, 1885; 27 generations from al-Bukhārī);
• Hugo Dyonizy Steinhaus (Göttingen, 1911; 28 generations from al-Bukhārī).

This, plus the six generations above from Steinhaus, puts Martínez at 34 generations from al-Bukhārī.

The rather weak path from Euler to Klein:

• Euler;
• Lagrange (no degree from Euler, and Euler didn't teach him in person, but in their correspondence they invented the variational calculus, and then Euler got him his position directing mathematics at the Prussian Academy of Sciences and at Frederick's court; he actually studied at Turin);
• Fourier;
• Dirichlet (Bonn, 1827; also studied under Poisson);
• Rudolf Otto Sigismund Lipschitz, the Lipschitz continuity guy; 61,331 descendants (Berlin, 1853);
• Klein (Lipschitz's only known student).

Unfortunatly, Euler was less prolific at training students than he was at engendering children or writing papers; if we discount Lagrange, Euler has only 5835 descendants, mostly in the Netherlands, many alive today.

Scholarchs of Plato's Academy

In ancient times, we can trace the sequence of scholarchs of Plato's Academy for some 300 years, who presumably each were in some sense the academic advisor of their successor:

• Socrates, in some sense, who died in 399 BCE
• Plato (from circa 387 BCE until his death in 348 or 347 BCE)
• Speusippus (347–339 BCE)
• Xenocrates (339–314 BCE)

• Polemo (314–269 BCE)
• Crates (circa 269–266 BCE)
• Arciselaus (circa 266–241 BCE)
• Lacydes of Cyrene (241–215 BCE)
• Evander and Telecles, jointly (215–circa 165 BCE)
• Hegesinus (circa 160 BCE)
• Carneades (circa 155 BCE)
• Clitomachus (129–circa 110 BCE)
• Philo of Larissa (circa 110–84 BCE)

At this point, the Academy was destroyed by Sulla during his siege of Athens, and Antiochus of Ascalon began teaching Stoicism; Cicero studied under him in 79 and 78 BCE and diffused Greek philosophy to the Romans.

This gives us two pieces of the chain connecting us over 2400-odd years to Socrates: one about 280 or 290 years long at the beginning, and another about 800 years long at the end. There's a 1320-year-long gap in the middle which runs through the Macedonian, Western Roman, Byzantine, and Muslim empires, which I don't know much about.

Presumably Archimedes of Syracuse (circa 287–212 BCE: "δῶς μοι πᾶ στῶ καὶ τὰν γᾶν κινάσω", "Transire suum pectus mundoque potiri") was aware of the Academy at Athens; I don't know if he was taught by anyone from the Academy, but he may have studied at Alexandria with Eratosthenes, the third Chief Librarian, shortly after Euclid wrote there.

Going back further, Imhotep ("He who comes in peace"), who designed Djoser's Step Pyramid 2000 years before (circa 2650–2600 BCE), presumably had teachers and students, but they are lost to history; the scribe Ahmes, who wrote the Rhind Papyrus around 1650 BCE, is similarly mysterious. Socrates might have been a follower of Pythagoras (circa 570–495 BCE) who was likely taught mathematics through a line related to that of Ahmes; he is reputed to have traveled to Egypt (and Babylonia, and Chaldea, and maybe India) seeking knowledge.

The Buddhist lineage of dharma transmission

There's another similar academic lineage tradition: the transmission of the Buddha Dharma from one teacher to the next, which connects us personally with Siddhartha Gautama through an unbroken line of Buddhist monks. For example, Stephanie can traced Shunryu Suzuki's dharma transmission lineage back to Bodhidharma, who brought Buddhism form India to China, as follows:

• Bodaidaruma (Bodhidharma, d. 532)
• Taiso Eka (Dazu Huike / Ta-tsu Hui-k'o, 487-593)
• Kanchi Sosan (Jianzhi Sengcan / Chien-chih Seng-ts'an, d. 606)
• Daii Doshin (Dayi Daoxin / Ta-i Tao-hsin, 580-651)
• Daiman Konin (Daman Hongren / Ta-man Hung-jen, 601-74)
• Daikan Eno (Dajian Huineng / Ta-chien Hui-neng, 638-713)
• Seigen Gyoshi (Qingyuan Xingsi / Ch'ing-yuan Hsing-ssu, 660-740)
• Sekito Kisen (Shitou Xiquian / Shih-t'ou Hsi-ch'ien, 700-90)

• Yakusan Igen (Yaoshan Weiyan / Yao-shan Wei-yen, 751–834)
• Ungan Donjo (Yunyan Tansheng / Yun-yen T'an-sheng, 780–841)
• Tozan Ryokai (Dongshan Liangjie / Tung-shan Liang-chieh, 807–69)
• Ungo Doyo (Yunju Daoying / Yun-chu Tao-ying, d. 902)
• Doan Dohi (Tongan Daopi / T'ung-an Tao-p'i, ???)
• Doan Kanshi (Tongan Guanzhi / T'ung-an Kuan-chih, ???)
• Ryozan Enkan (Liangshan Yuanguan / Liang-shan Yuan-kuan, ???)
• Taiyo Kyogen (Dayang Qingxuan / Ta-yang Ching-hsuan, d. 1027)
• Toshi Gisei (Touzi Yiqing / T'ou-tzu I'ch'ing, 1032–83)
• Fuyo Dokai (Furong Daokai / Fu-jung Tao-k'ai, 1043–1118)
• Tanka Shijun (Danxia Zichun / Tan-hsia Tzu-ch'un, d. 1119)
• Choro Seiryo (Zhenxie Qingliao / Chen-hsieh Ch'ing-liao, 1089–1151)
• Tendo Sokaku (Tiantong Zongjue / T'ien-t'ung Tsung-chueh, ???)
• Setcho Chikan (Xuedou Zhijian / Hsueh-tou Chih-chien, 1105–92)

• Tendo Nyojo (Tiantong Rujing / T'ien-t'ung Ju-ching, 1163–1228)
• Eihei Dogen (1200–1253)
• Koun Ejo (1198–1280)
• Tettsu Gikai (1219–1309)
• Keizan Jokin (1264–1325)
• Gasan Joseki (1276–1366)
• Taigen Soshin (d. 1371)
• Baizan Monpon (d. 1417)
• Shingan Doku
• Senso Esai (d. 1475)
• Iyoku Choyu
• Mugai Keigon
• Nenshitsu Yokaku
• Sesso Hoseki
• Taiei Zesho
• Nampo Gentaku
• Zoden Yoko
• Ten'yu Soen
• Ken'an Junsa
• Chokoku Koen
• Senshu Donko
• Fuden Gentotsu
• Daishun Kan'yu
• Tenrin Kanshu
• Sessan Tetsuzen
• Fuzan Shunki
• Jissan Mokuin
• Sengan Bonryo
• Daiki Kyokan
• Eno Gikan
• Shoun Hozui
• Shizan Tokuchu
• Nanso Shinshu
• Kankai Tokuan
• Kosen Baido
• Gyakushitsu Sojun (187?– 1891)

• Butsumon Sogaku (1858–1933)
• Gyokujun So-on (1877–1934)
• Shogaku Shunryu (Suzuki, 1904–1971)

Miscellaneous lineages

A third such academic lineage is the lineage of the rabbis.

Another is that descending from the Great Peacemaker of the Haudenosaunee, around 1200 CE, through Hiawatha, guardians of the Great Law of Peace, which was encoded on wampum belts and may have inspired the Western revival of democracy.

Topics

Dercuano rendering

Kragen Javier Sitaker, 2019-05-11 (updated 2019-05-12) (3 minutes)

As described in Dercuano drawings (p. 64), I want to add illustrations to Dercuano. Some of the cases where illustrations will help most are the shapes of three-dimensional objects, and it occurs to me that in many such cases it might be easier and quicker to specify the three-dimensional geometry of the objects than to sketch them by hand.

The trouble is that I want to make sort of casual sketches, and the semi-photographic quality of normal rendering makes conspicuous any insufficiency in the models being rendered; also, shiny eye-catching rendered graphics might be eye-catching enough to detract from the kind of thoughtful consideration I'd like the reader to be able to apply. So some kind of non-photorealistic rendering might work better.

In particular, I was thinking that maybe by using lines of varying width or varying darkness, I could get a kind of engraving effect, though maybe that's too skeuomorphic. To the extent that the lines run in straight lines (geodesics) along the surface of the 3-D model, they can additionally help by showing the curvature of the surface; to the extent that their orientations aren't determined by the view, the projection will tend to skew them in a way that shows the orientation of the surface even when it isn't flat.

It might be simplest, and perhaps adequate, just to render things in grayscale, though. That might be adequately calm.

Slow, perhaps periodic rotation or nutation might also help with showing 3-D structure; perhaps motion-blurring it would prevent the motion from being too distracting.

A separate question is how to carry out the construction of the 3-D model. Of course this can be almost arbitrarily easy or arbitrarily difficult, but the methods available to date leave a lot to be desired. Listing center-coordinates, radii, and colors of spheres is simple, but there's only so much you can build with spheres, and it's a time-consuming way to build it. CSG is intuitive, for what it can express, but algorithmically it can be very challenging, and it's worthless for modeling tree bark, filleted joints, or smooth curves, and again, it's (often) very time-consuming to use. Teddy 3D is intuitive but difficult to achieve precise results with. The constraint-solving pipeline approach used by FreeCAD, CATIA, and SolidWorks (modeled on the approach taken by Sutherland's SKETCHPAD) is clearly capable of constructing very complex shapes, but these programs' solvers not infrequently fail to converge, and when they almost fail to converge, are quite slow.

Another thing to keep in mind is the possibility of printing. Black and white polygons print very well on laser printers, as do solid lines and polygons of the printer's primaries if it's a color printer; color and grayscale regions require some kind of halftoning, which dramatically reduces the resolution.

Topics

Kernel code generation

Kragen Javier Sitaker, 2019-07-02 (6 minutes)

John Regehr just posted It's Time for a Modern Synthesis Kernel, and I wrote the following comment on the orange website.

This is a wonderful idea, and I hope many people start working on it right away.

Although Massalin has never published her code, according to my memory of her thesis, Synthesis's runtime code generation was *mostly* extremely simple, more like linking than what we think of as "code generation" — it copied a template method into the appropriate slot in the newly-generated quaject, then overwrote specific bytes in the generated code with pointers to the relevant callout (or, in some cases, the initial value of an instance variable for that quaject). Parts of the code that did not benefit from being specialized in this way were factored into ordinary functions the quaject method would just call.

This meant that only a small amount of code was generated for each quaject, and the runtime code generation was very nearly as fast as memcpy(), which meant that it was reasonable to use it on every quaject instantiation.

Massalin *also* talked about applying some optimizations to the generated code, such as the constant-folding and dead-code removal John mentions, but I have the intuition that only a minority of quaject instantiations involved such more aggressive optimizations. Since she never published Synthesis, it's impossible to know for sure. (I'm not questioning her integrity or claiming that the impressive benchmarks reported in her dissertation are faked; I'm saying that we unfortunately can't see the exact mixture of interesting things you need to do to get those kickass benchmarks; so, like an insecure Intel CPU, I'm reduced to speculation.)

Later implementations inspired by Massalin's approach included Engler's VCODE (which, to my knowledge, has also never been published; Engler's PLDI paper cites Massalin in the second sentence of the abstract), which was used to implement Engler's `C, and GNU Lightning (inspired by Engler's published papers *about* VCODE), used in a number of modern JIT compilers.

I suspect that, by contrast, John's idea of using LLVM is inevitably going to have much higher overhead — if only from the CPU cache devastation brought about by any LLVM invocation — so will only be a win for much-longer-lived objects, where the large instantiation overhead can be amortized over a much larger number of invocations. An intermediate approach like Engler's `C might be more broadly applicable.

John suggests this early on in his "for deployment" comment, but I think that it's probably necessary for prototyping too, since the objective of the whole exercise would be to get an order-of-magnitude speedup, and the objective of the prototype would be to find out if that's a plausible result. A prototype that makes all your programs run slower due to LLVM wouldn't provide any useful evidence about that.

I asked Perry what he thought about the above, and he replied with this gem:

So you're correct that the code generation was mostly "template instantiation". I think that was key to having calls like open() function in reasonable time. I also suspect LLVM is a blunt instrument for this work. That said, it would have been difficult for anyone but Massalin to work with the approach in Synthesis. It was very much the product of a person who was both intensely brilliant and completely comfortable with writing "weird code" in the instruction set they were working in.

So there's then the question of how one can take the ideas from Synthesis and make them a practical thing that ordinary programmers could build and contribute to. And that is almost certainly going to involve compiler tooling. As a prototype, making this work by using LLVM is probably a good approach. Ultimately, I think that one is going to have to do the magic at kernel build time and have something fairly lightweight happen at runtime. But to figure out what that is, one needs to play. And the easiest tools right now for playing involve LLVM. If, for example, you can use LLVM successfully to specialize a write call's instruction path down an order of magnitude or more, or to do similar things in the networking code, one can then ask how to do this better.

There are, of course, a number of interesting paths towards playing with this. I suspect that none of them end anywhere near where they start. But the only way to see what might be possible with much better tooling is to start, and you have to start somewhere.

BTW, I think the time is right, or even over-right, for this. Single processor core performance is stalled out, and while in 1992 one could just say "well, we'll have another factor of ten performance improvement in a few years, who needs the trouble", that's no longer the case. Note that this argument also applies, to a considerable extent, to other parts of the modern software ecosystem. When you can't just say "we could spend a couple of months optimizing this, but the next generation of processors will be out by then", things change.

Anyway, not sure if this answers your call for comments, but if you are interested in specific areas around this, I've no shortage of opinions. Many would say I have far too many opinions...

You can quote any subset or the entire thing. So long as you don't distort my intent I have no problem with people using my words that way.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Operating systems (p. 3608) (18 notes)
- Code generation (p. 3381) (2 notes)

What's wrong with ../../?

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

Relative links are great; they let you move your whole tree of HTML files from one place to another and still retain the internal link structure. However, they start to suffer when you have multiple levels of directory structure: is that `href="../../style.css"` or `href="../../../style.css"`? It's a bit confusing, and even if you don't get confused, you still have to modify links when you copy them from one file to another.

What would be more helpful would be the ability to say "up to a directory named foo". Suppose you have this setup:

```
kragen/
  index.html
  resume.html
  style/style.css
  images/
    kragenlogo.png
    headshot.jpg
  blog/
    1.html
    2.html
    archive/
      2008-03.html
```

Now, suppose there's some text in `2008-03.html` that was originally in `2.html` or one of its siblings. It would be nice if that text didn't have to be changed from `<a href="1.html">` to `<a href="../1.html">`. You can write `<a href="/kragen/blog/1.html">`, but in addition to being verbose, that makes it hard to use a tree of HTML that you've downloaded with `wget -r` or something similar.

Suppose you could instead write `<a href="$blog/1.html">`, meaning "go up until you find an ancestor directory named `blog`, then use its children". Now you can write things like `<img src="$kragen/headshot.jpg">` freely, and copy and paste them among all the files.

By itself, this would be a backwards-incompatible change to browsers and the URL spec, but it could degrade gracefully. You could program your web server to generate redirects for backwards-compatibility, while implementing the change in newer browsers. Compatibility problems would only arise if someone had a relative link to a directory whose name began with "$" whose name otherwise duplicated that of a directory higher up in the hierarchy.

Topics

• Browsers (p. 3351) (6 notes)

The uses of introspection, reflection, and personal supercomputers in software testing

Kragen Javier Sitaker, 2019-02-04 (updated 2019-03-11) (12 minutes)

We can do a lot more automated testing now than we could before, but manual testing still requires the same amount of human effort. To some extent our new powers of automated testing are counterbalanced by our new ability to write inefficient software, but not in all cases.

When we have a program, we can run it many times on different inputs to see what it does. "Hypothesis" is David R. MacIver's Python system for doing this automatically ("generative testing" or "property-based testing"), and it has been remarkably effective in my experience in flushing out hidden bugs in programs — including, unfortunately, the Python interpreter!

The number of test cases we can run scales inversely with the amount of work done by each test. Consider this function:

```
void
yp_font_show(yp_font *font, char *text, ypic where)
{
  int x = 0, y = 0;
  for (int i = 0; text[i]; i++) {
    ypic g = yp_font_glyph(font, text[i]);
    if (x + g.size.x > where.size.x) {
      x = 0;
      y += font->max_height;
    }
    if (x + g.size.x > where.size.x) return; /* Glyph is too wide to fit */
    if (y > where.size.y) return;

    yp_copy(yp_sub(where, yp_p(x, y + font->max_height - g.size.y), g.size), g);
    x += g.size.x;
  }
}
```

In a benchmark, this function takes about 36 µs on my laptop to draw the text "hello, world" in a window in a large font, or 5.7 µs in a 12-pixel-tall font.

One core of my four-core laptop executes about 2.4 billion 64-bit instructions per second. A 33-MHz 80386 executed about 4.3 million 32-bit instructions per second; we can extrapolate that it might have run this function some 560 times slower, perhaps requiring 3.2 ms for the 12-pixel font instead of 5.7 µs. So the 386 could test this function (with similar inputs) 300 times per second, while my four-core laptop can test it 700'000 times per second.

Testing hardware: a 40-kW, 100-million-MIPS Pi Beowulf per

programmer?

Moreover, I can potentially take advantage of more than just my laptop for automated testing. A Raspberry Pi 2 is about 5000 MIPS and runs on about 2 to 2.3 watts. If I have a 1-kilowatt testing budget, I can run about 500 Pi 2s and get about 2.5 trillion instructions per second, testing this function roughly 700 million times per second.

(This works out to 400–460 pJ per instruction. The STM32L Cortex-M0 (see files Low-power microcontrollers for a low-power computer (p. 2602) and Notes on the STM32 microcontroller family (p. 3176)) uses about a third of that, but has less memory.)

Is a kilowatt a reasonable budget? Such a rig also costs about US$20k up front and has 500 gigabytes of RAM; at 67×56×11.5 mm each, it would occupy about 21 liters, a box 275 mm on a side, but probably a bit more space is needed for cooling fans and power supplies. A kilowatt costs about US$0.10 per hour, or US$900 per year, which is about a sixth of the depreciation, depending on how you calculate it. This suggests looking for ways to reduce the up-front cost per MIPS even if they increase power consumption.

If we figure that employing a programmer these days costs about US$300k per year (including overhead and benefits), the programmer dominates the expenses until the testing budget gets to 43 kilowatts per programmer, at which point we're spending US$600k per year and getting 30 billion tests of `yp_font_show` and 108 trillion instructions per second. This is more than a cubic meter of computer.

Manual vs. automatic testing cost comparison

By contrast, I might be able to write JUnit-style regression tests of this function at a rate of one per minute or so — which should be downgraded to one per ten minutes if we include the time to maintain those tests, and perhaps one per thirty minutes if we include the time I'm asleep or otherwise not working on the project.

So, to the extent that we can get any benefit at all out of running computer-generated tests, they have a multiplier of around 700 million · 1800 = 1.3 trillion over human-generated tests, or 30 billion · 1800 = 54 trillion with a 43-kilowatt system. Writing a test by hand costs about as much as running 54 trillion randomly-generated tests. In the days of the 386, this multiplier was about half a million, and now it's one to fifty trillion, or nearly a billion without extra hardware†. Each of your manual tests need to be worth more than a billion or a trillion automatic tests to be worthwhile. (This is not as hard to achieve as it sounds, particularly when we're considering tests number eleven trillion to twelve trillion for the same code; remember that a trillion is only a bit under 2^{40}.)

† If you're using Python, automated tests lose a factor of 100 or so because of the interpreter's poor performance.

Extending generative testing

Two obstacles to the usefulness of random testing are reduction and repeatability. The initial randomly-generated test case that reveals the bug may be fairly large and include any number of strange things not related to the failure, while manually-written test cases are simple by

design. And, once the failing test case has been found, it's important to be able to run it again in the future — otherwise it doesn't help you know whether you've fixed the bug (or avoided reintroducing it.) Hypothesis includes an automated test-case reduction which reduces the failing case to a (usually) fairly minimal case before reporting it, and a system for recording previously failed tests to run again — although I haven't tried checking them into Git, and I suspect there might be some impedance mismatches there.

Hypothesis has recently (I think around late 2017) acquired the ability to combine its generative testing approach with statement coverage testing, which is where you measure which lines of code were executed by your tests, so that you can work to focus new tests on lines that haven't been tested yet. In theory, this kind of testing could also give you a reasonable suspicion of which lines of code were responsible for a bug. And you could conceivably augment this with extended kinds of coverage such as loop coverage, multi-condition coverage, and mutation testing; these are just as expensive to apply now by hand as when Marick wrote his paper on it, if not more so, but three orders of magnitude cheaper to apply automatically — Marick was writing in 1991 or 1992, at a time when the 386 was current, though not cutting-edge.

Thus a moderate amount of introspection or reflection, like that needed for statement coverage testing, might enable large gains in test power.

Program search

Earlier I said, "Each of your manual tests need to be worth more than a billion or a trillion automatic tests to be worthwhile." Turning that on its head, what if you just write the tests and not the implementation? In a way, this is the approach behind deep learning: try to generate a program for a very restricted machine that passes some large set of tests.

You could imagine a generative testing system like Hypothesis that suggests code mutations that cause failing test cases to pass; this is a very small version of "what if you write the tests and not the implementation?". Once you have found a failing test case, you can try on the order of 50 million modified versions of the code per second. Perhaps you could expand this to a practical program search system by accelerating the process by writing some very verbose test cases that include some debug messages showing intermediate results while running the algorithm, so that the Kolmogorov complexity of each successive search objective is only 16–32 bits, making the search tractable.

The functional-programming equivalent might be to write test cases for subordinate functions that can be composed to do the desired computation.

Michał Zalewski's American Fuzzy Lop testing system, which provides random input data to programs to test them ("fuzzing"), uses Unix's fork() to virtually "take snapshots" of a process under test at different times during its execution; this allows it to execute many more test cases per second, because new test cases can start from the program state some tens of microseconds before the program exited, rather than from the beginning. Using this facility, he has demonstrated some very impressive results, including constructing a

legal JPEG file by fuzzing a JPEG decoder to find an input file it would accept.

Linux's implementation of snapshotting with fork() is unfortunately rather slow (though faster than other Unixes): some 130 µs on my laptop to fork() and wait() for a dietlibc-linked child process that merely invokes exit(), perhaps on another core. This compares to 0.4 µs for a minor page fault, 0.3 µs to call read(), and probably something like (guessing based on others' measurements) 5 µs to context-switch the core to another already-existing process and (guessing) 0.1–0.2 µs for a one-way IPC under seL4.

This suggests that, even using conventional protection mechanisms, it should be possible to take lazy process memory snapshots an order of magnitude faster than Linux does it, which in turn would make it practical to take them an order of magnitude more often — every 5 µs, for example, rather than every 50 µs. In cases where the the execution time from the last test case input to getting a test result — the feedback lag — is on the order of those 5 µs, this is an order-of-magnitude speedup, but its importance goes far beyond that, because the input search problem is exponential in the feedback lag. 5 µs is about 12000 instructions, which admits an insanely large number of possible executions to search through.

This suggests that unconventional protection mechanisms, such as Valgrind-like machine-code compilation and interpretation, might work better for program search; by recording every change to memory and registers, they can rewind the timeline to any arbitrary instruction execution, at a cost of optimistically 10× to more likely 100× execution time (and also some memory). In cases where the feedback lag can be reduced below, say, 120–1200 instructions, you can thus fail more tests per second with this approach than by using arm's-length mechanisms like virtual memory protection. With the Linux implementation, which costs on the order of 300'000 instructions, the crossover point is around 3000–30'000 instructions of feedback lag; for shorter feedback lags, the Valgrind and interpreter approaches will work better.

Abstract interpretation techniques can yield bigger speedups in some cases, like the miniKANREN system that is capable of finding a quine by searching through possible executions of an interpreter for one whose output is the same as its input.

Topics

Gradient descent beyond machine learning

Kragen Javier Sitaker, 2018-05-18 (2 minutes)

Gradient descent and friends are widely used in machine learning. The basic recipe is that you have some statistical model which contains some parameters, and you use gradient descent to set the parameters to, hopefully, minimize the prediction error.

This is an extremely useful approach, and you could argue that, even by itself, it's revolutionary. But gradient descent is useful for other things that don't fit into this model. Gradient descent is a general continuous optimization procedure; it can be used to approximate a local minimum of any differentiable function, and it scales well to large numbers of dimensions.

Let's unpack this a little bit. In the standard ML approach, you have a MODEL which, given PARAMETERS and INDEPENDENT VARIABLES, produces a PREDICTION of some DEPENDENT VARIABLES with some amount of ERROR. You use gradient descent, or some related algorithm, to try to set the PARAMETERS to minimize the ERROR. And you have to worry about whether you're overfitting, and so you use a training set and a test set to see if the model is starting to overfit your training data.

But gradient descent doesn't know anything about predictions or overfitting or the dependent and independent variables of your model. From gradient descent's point of view, the model and training set (including dependent and independent variables) are just part of a LOSS FUNCTION which, given INPUTS, produces an error or LOSS.

Gradient descent is a somewhat general procedure for finding the inputs that minimize such loss functions.

It turns out that there are a lot of other things in the world that can be represented in this way. There are a lot of things you might want to minimize that are not prediction errors.

Topics

You can't sort a file whose size is cubic in your RAM size in two passes, only quadratic

Kragen Javier Sitaker, 2015-05-28 (5 minutes)

This is a bad idea that has occurred to me before, and I want to write down why it's bad so that it can stop occurring to me.

The problem to solve

The problem to solve is to increase the capacity of two-pass external sorting, in which you ingest a large input file on one pass over the data, storing it on disk, and then emit it, sorted, during a second pass over the data, reading it from the disk. We're counting disk I/O here rather than, say, key comparisons, because the disk I/O is usually the slow part.

The main limitation of two-pass external sorting is that, in two passes, you can only sort files that are about twice the size of your RAM, times the size of your RAM, divided by your disk's bandwidth-latency product. So if your (spinning-rust) disk transfers data at 50 megabytes per second and has 9ms random-access latency, your bandwidth-latency product is 4.5 megabytes. If you have 4.5 gigabytes of RAM, then you can sort files up to about 2000 times the size of your RAM — 9 terabytes — in two passes. If you have 9 gigabytes of RAM, you can sort up to about 8000 times the size of your RAM — 72 terabytes — in two passes. So the critical size is quadratic in the size of your RAM, inversely proportional to the disk's access latency, and apparently perversely, inversely proportional to the disk's bandwidth.

(The apparent perversity here is only apparent; increasing the disk's bandwidth won't actually make two-pass external sorting slower. It just won't make it faster.)

We can model SSDs in this model as disks whose bandwidth-latency product is on the order of 1-4 kibibytes, which is five orders of magnitude better than spinning rust. Unfortunately, this does not increase the amount of data you can sort by five orders of magnitude, because your SSD probably isn't bigger than ten terabytes. It might make more sense to model the CPU-RAM-SSD system as "CPU and RAM" from the point of view of the disk, thus probably enabling you to sort files up to the size of your disk in two passes.

How two-pass sorting works

Simplest method: fill RAM with unsorted data, sort in RAM, write to a new temporary file on disk, repeat until input is consumed; now read these N sorted files and read them all concurrently, merging them together to produce the output.

Optimization: maintain in-RAM data in a bin-heap, and each time you write out a record, read in a new record to replace it, adding it to the bin-heap if possible. This doubles the size of your initial sorted files on average for randomly-ordered input data, and does much

better than that if the input is sorted or nearly so.

Backwards alternative: partition keyspace into N roughly-equal-weight disjoint partitions, and open N temporary output files. Deal input records into the appropriate file as they come in. Once input is exhausted, sort and output each of the N temporary files in order.

The efficiency limitation on the number of concurrently open files is that you need some buffer space in RAM for each open file in order to batch the I/O into operations big enough to use most of the disk's bandwidth. When the buffer space gets smaller than the bandwidth-latency product, the disk spends most of its time seeking.

If you do more passes, you can increase the sortable data size dramatically. Back in the magtape days, apparently many-pass sorts on three or more tape drives were common.

The bad idea

I keep thinking that there should be a way to make two-pass external sorting able to sort a data file *cubic* in the size of your RAM.

The basic idea is that you have N^2 temporary files: you write to N of them at a time during input ("column-wise"), and read from N of them at a time during output ("row-wise").

Why is this idea bad? Well, are these temp files going to be sorted or not? If they're not sorted, then you need to sort them before you can merge N of them together, which means you need to fit N of them into RAM at once during output if you're going to avoid making a third pass over the data. If they are sorted, then they have to get that way somehow, which means that you need to fit N of them into RAM at once during input.

So as far as I can tell, this approach just doesn't work, and there's no way to make it work.

Topics

• Performance (p. 3621) (149 notes)
• Algorithms (p. 3310) (123 notes)
• Sorting (p. 3720) (8 notes)

Alien game challenge

Kragen Javier Sitaker, 2015-09-03 (6 minutes)

"8-bit" or "New Aesthetic" games like Minecraft, Bit Trip Runner, and arguably Dwarf Fortress are increasingly popular, perhaps reflecting a newfound appreciation for gameplay in place of high production values. To some extent this is also a way to disguise lack of production values: if you put crappy models on top of Unreal Engine, the environment they're in makes them look worse by implicitly comparing them to the ones in Counterstrike or Call of Duty [0], and indie devs can avoid suffering such comparisons in part by avoiding the engines that create production-values expectations they will inevitably fail to meet.

[0]: CoD is on Unreal Engine, right? Did I misspell that?

Occasionally there are games programming competitions that focus on some kind of artificial restriction like this, sometimes even using specific old platforms. The problem is that, for obvious reasons, these restrictions tend to resemble restrictions we have actually suffered in the past, so the aesthetic of the games tends to be somewhat predictable: square pixels, small color palettes, 2.5D, 7-segment displays, FM or square-wave or triangle-wave synthesis.

But what if, instead of creating video games that recall the age of Space Invaders, we tried to create video games that recalled the analogous stage of technological development on an alien planet, or perhaps an alternate-history Earth? We could have been laboring under different restrictions and produced different kinds of games.

A number of things have happened recently in games that suggest some possibilities. Flappy Bird could easily have been written in 1985 [1], but there were no similar games. 3-D wireframe games were feasible in 1985 (Maze War, the first first-person shooter, was from the 1970s, and the Star Wars game was a popular arcade game of the time) but they greatly benefited from vector displays, which went out of style at the time.

[1] I think James Hague pointed this out but I'm offline at the moment.

So, some example alternative histories:

- As I said, what if vector displays?
- What if hexagonal pixels had been the standard? They have more consistent spacing and thus reduce sampling error for the same number of pixels, which is why tabletop strategy games (arguably including Settlers) use hexagonal tiles.
- What if Tek-4014-style storage tubes had been the mainstream display technology? You can't erase part of the screen; you have to erase the whole screen.
- What if touchscreens? But we're already exploring that with game dynamics like Flappy Bird and Angry Birds.
- What if timesharing?
- Music synthesis with electronic difference engines, sort of like vector-display display lists, but for waveforms?

a. Music synthesis with circulating bit sequences in shift registers (like the Apple]['s NTSC colors) rather than analog waveforms?

• Vector displays with higher-order display lists? You could display quadratic splines as easily as points, but maybe only a small number of new ones per second.
• What if the CPU itself were fundamentally different in some way?
  a. If most of your memory were write-once read-many, so that every time you played a game, you irreversibly used up some of the memory?
  b. If most of your memory were fundamentally sequential-access, like bubble memory or magnetic tapes? (Arguably we're suffering that now with disks, and of course videodisc games of the 1980s explored some of this too.)
  c. If the CPU were a combinator-graph reduction machine like the SKIM rather than a RAM machine?
• Character-cell displays created an aesthetic that survives in Dwarf Fortress and Nethack, but also that influences any number of modern "8-bit" games, because they use fixed-width fonts. But to a great extent those displays were very culturally contingent, and while perhaps Greek and Latin alphabets are among the simplest systems to draw this way, it seems like it's probably a coincidence that the cultures that invented computers happened to use alphabets rather than other writing systems. So consider, what if your character display were for a language that was:
  a. Top-to-bottom?
  b. A syllable-block system like Korean hangul?
  c. One of those where accents are essential to understandability? Maybe, like APL, your display would need to support arbitrary overstrikes, or at least two or three overstruck characters. How could this reduce the size of the character set?
  d. Necessarily variable-width, or necessarily supported character overlaps? I don't know if Devanagari, for example, is actually this way, but I've never seen non-proportional Devanagari. Proportional font rendering isn't actually that much harder than fixed-width font rendering. Historically it came along with framebuffers, which certainly make it easier, but are framebuffers really necessary?
  e. Ideographic? Would you necessarily move to a small syllabary for computers, like 1980s Japanese PCs used katakana, or could you perhaps handle some kind of large ideographic character set in reasonably simple hardware?
• Hardware sprites and hardware scrolling were a big deal in 1980s games, since you didn't have time to redraw the entire screen each frame. Similarly, for fractals, hardware palette color-cycling was a big deal, a technique which also permits fade-to-black, flashing a certain color index, and so on, without having to redraw the entire screen; and a bit earlier, XORing objects into the framebuffer allowed you to move them around the screen without having to save the background they passed over in separate memory. But suppose the available display hardware had supported other interesting operations instead? Of course, there's an infinite variety of possible operations, most of which (like XOR) are mostly visually uninteresting. But there are a wide variety that seem like they would have been interesting but were never implemented in hardware:
  a. Displacement mapping, which is kind of a generalization of sprites: you have a number that you add to the pixel coordinates generated by the hardware counter to get the address in the

framebuffer (or tiles, or whatever) to draw the pixel.

Topics

2016 outlook for automated fabrication and 3-D printing

Kragen Javier Sitaker, 2016-08-11 (20 minutes)

I think other kinds of automated fabrication might serve as well as 3D printing or better for general-purpose home use.

There are several kinds of automated fabrication. CNC machining is pretty important. Lithography is pretty important; it's how both ICs and printed circuit boards are made, and in one or another form it's how almost all text and pictures get onto manufactured goods. Robot welding has been a major part of fabricating many kinds of metal goods since the 1980s. The first versions of CNC machining were cam-controlled, in the late 19th century; they were called "screw machines". After setup, you would feed them a piece of bar stock and they would conduct a programmable series of lathing operations on them, repeatedly, under the control of the cams, which is how screws have been made ever since then and why screws are cheap.

The promise of 3D printing was, when 3D Systems was founded in 1986, that product designers could rapidly produce prototypes to experiment with, even if the cost was far too high to compete with mass-produced products. My math teacher told the class about it in 1993; he'd seen a plastic apple produced by stereolithography.

In 2005, Adrian Bowyer founded RepRap as a project to make automated fabrication accessible to the masses. Its goal was to make "rapid-prototyping machines" so cheap that it's cheaper to make things with them than with injection molding, by making them self-replicate. But it failed at that goal.

Problems with RepRap-style 3-D printing

RepRap-style 3D printing — 3-axis gantry open-loop single-material FDM without a heated chamber — is really limited in what it can achieve for a variety of reasons.

Most of these are problems that would not be serious if the printer could self-replicate at a low cost, as was the original goal — it could print a new version of itself that solved the problem.

Imprecision

RepRap-style 3D printers typically have 100-micron errors even in the X and Y directions, and due to the nature of FDM, the Z-axis of each piece is quantized to typically between 250 and 500 microns, resulting in a stairstep appearance.

This doesn't sound like a lot, but it's common for any ordinary subtractive machining operation to be carried out to 20-micron or 10-micron precision, with 1-micron precision in areas that need it. An ordinary 600-dpi laser printer has 40-micron precision; an expensive 2400-dpi printer has 10-micron precision. 1-micron feedback sensors are easily available, but RepRap-style printers don't use them. Why? I think it's a design error, but I think the intention was to lower costs.

The big problem with imprecision is that it makes the properties of the material unpredictable, because you get unpredictable layer

adhesion. Poor layer adhesion gives you a tiny fraction of the potential tensile strength of your material in the Z direction.

Two of the main sources of poor layer adhesion are actually bed-leveling problems and filament diameter miscalibration.

Recent RepRap-family printers have ceded somewhat on the open-loop-control front to do automatic bed leveling by touching down on each corner of the bed before starting the print, then twiddling the Z axis dynamically as the extruder moves around in X and Y to compensate for bed tilt. This ensures that the first few layers deposited have a uniform thickness, greatly reducing layer adhesion problems. Without bed leveling (either manual, using screws, or automatic as described above) it's easy to have a bottom layer that's squished flat on one side of a piece and barely touching the bed on the other, with lots of roundness in the deposited filament.

Filament diameter miscalibration also affects layer thickness and dimensional precision; if the filament diameter is 10% smaller than expected (for example, 180 microns if your filament is 1.75 mm diameter, the most popular size), then the amount of plastic per linear millimeter is 21% smaller, and so the width of extruded traces will be 21% low, and you may get poor layer adhesion. Many commercial filaments have significant diameter variations even within a single spool, and they are rarely perfectly round, making precise measurement difficult.

Imprecision puts a severe limit on volumetric resolution. I'm particularly interested in volumetric resolution because, for mechanical computation, that's what counts — if a certain multiplier design requires 4 million voxels, then it will be 10.5 milliliters on a RepRap-style machine or 0.5 cubic millimeters (half a microliter) on a hypothetical 5-micron-resolution 3-D printer.

Material limitations

With FDM without a heated chamber, your materials are basically limited to PLA, ABS, and nylon, and filled or colored versions of these. (Except see below about PVA, elastomers, and PETG.)

These materials are not all bad. They come out of the machine sterilized, which is nice, and PLA and nylon are biocompatible; and the most interesting fillers I've seen are brass and fine sawdust, which give you the ability to produce objects that are mostly brass or wood. PLA is very dimensionally stable, ABS is very strong, and nylon is very tough and easily accepts very large deformations (30% elongation at break is typical).

But they do have many disadvantages.

PLA is weak and brittle.

But ABS and nylon offgas toxic fumes during printing, so you likely don't want to print in them indoors without a good ventilation system. ABS also leaches toxic styrene when in contact with acid, so you can't use it for food-contact applications. You can't use PLA for most food-contact applications, where you would expect it to shine due to its nontoxic nature; the problem is that it softens at a very low temperature, as low as 60°, even though you need at least 180° to extrude it.

PLA is hygroscopic and aquadegradable, so by leaving a PLA spool exposed to air, you can run into big problems printing after a few weeks or months. The water in the plastic can boil and make bubbles

as it goes through the nozzle, or it can just degrade the plastic, making it even weaker and more brittle than it already is.

ABS and nylon require high temperatures, like 210° and 240° respectively, which PLA can't withstand; this is a problem if you switch from printing in nylon to printing in PLA, because you have to print at nylon temperatures until all the nylon is out of the nozzle, because it's frozen solid at the usual PLA temperatures.

ABS, because of its larger thermal shrinkage and high stiffness, has a terrible tendency to curl as it cools without an enclosed build chamber; worse, this depends on the ambient temperature, which means your ABS printing may be going fine all day and then fail during the night. If it curls up off the bed, it can collide with the nozzle and be knocked entirely out of place, but even if that doesn't happen, you can get delaminations which totally vitiate the strength of your print.

PLA is vulnerable to glass-style fatigue; a piece of PLA held in a stressed position may crack through after weeks or months with no motion. Glass does this too, but mostly only if it's underwater. In both cases there seem to be critical stresses below which the fatigue cracks don't propagate. Probably whether this happens to a given specimen of PLA depends in part on the amorphousness (glassiness) of the PLA, which in turn depends on the composition and cooling rate.

I don't have any experience printing in nylon, but it's very soft, which limits its applications somewhat. And the higher temperature makes the hotend fail more often.

FDM prints unavoidably (?) come out with a stairstep surface because of Z-axis quantization. If they are ABS, they can be smoothed by brief exposure to acetone vapor; this is harder to achieve with PLA, because PLA is invulnerable to almost all organic solvents. People have had success using dichloromethane, which is outrageously toxic, and reportedly ethyl acetate, which is several times less toxic than table salt. But the ethyl-acetate-based nail polish remover I tried didn't do the job, it just swelled the plastic. Other people just recommend painting the piece.

Thermoplastic elastomers like "Filaflex", "Ninjaflex", and "Flexifil" are a promising new addition that I don't have any real knowledge of; they don't have the level of superelasticity that rubbers do. Ninjaflex is a thermoplastic polyurethane; I don't know what the others are made of. Supposedly Filaflex has 70% elongation at break, which is quite a bit more than nylon's 30%, and Ninjaflex 660%, but neither equals an ordinary rubber band, which can easily hit 1000%.

PETG filament, like Form Futura's HDglass, is another newish product that claims to let you use FDM to fabricate transparent objects. I don't have any experience with it. Being PETG, it ought to be food-safe, too.

Single-extruder limitations

Standard RepRap-style machines have a single extruder nozzle, fed by a single filament, in order to lower the cost and improve reliability. That means it can only print in a single material at a time, and changing between materials is far from instant. This creates a number of problems.

Most obviously, unless you paint the piece, it's a single solid color,

or it's the uncontrolled sequence of colors that the filament had — like when you knit with variegated yarn.

I think the single-material limitation is the main thing that prevents RepRap replication from really taking off. The single most failure-prone part of a RepRap-style printer is the hotend, and that's the part it can't print, because it's made of brass, Teflon, Kapton, and a cartridge resistance heating element, which don't melt at the temperatures it's exposed to.

(Somewhat less obviously, the hotend has a dimensional tolerance that's tighter than RepRaps are usually capable of achieving: the diameter of the nozzle aperture.)

Dual-extruder FDM printers are capable of doing several things that single-extruder models can't manage, aside from being theoretically better suited to self-replication. One is printing objects that have controllable visible patterns on their surfaces. Another advantage is the ability to print support material in a soluble form: polyvinyl alcohol (PVA), a nontoxic water-soluble plastic that melts at a conveniently low temperature, is a favorite choice. But ABS, which easily dissolves in acetone, is a reasonable alternative for supporting PLA, which is invulnerable to acetone.

Dissolvable support material is an enormous help in making preassembled mechanisms. The usual RepRap-style support material has to be cut away from the finished piece using a knife, which requires manual intervention, leaves crap on the surface, and is difficult to impossible inside enclosed spaces. Also, it's easy to cut your hands that way.

Here's an example of using a dual-extruder FDM machine for dissolvable PVA support material.

Finally, there are a lot of kinds of machinery that inherently require two or more different materials to be deposited in intimate contact. Circuitry requires both conductors and insulators; PLA filled with graphite, lead, or more expensively, brass, silver, or gold, is totally adequate as a conductor for many circuits, but you need insulators too. Electric motors could conceivably be 3-D printed, but as far as I can tell, practical motors need many thin layers of conductors, insulators, and high-magnetic-permeability materials. (Those are typically a poorly-conducting kind of steel, deposited in layers with insulating epoxy in between to reduce eddy-current losses.)

Many of the more interesting possible applications of 3D printers are in metamaterials; by careful design of a repeating 3-D structure, for example, you can combime two materials with different thermal coefficients of expansion to get a bulk "metamaterial" with a TCE that's orders of magnitude lower than either of the two "phases" or materials that compose it. That's super important for precision measurement, feedback, and control.

Self-replication problems

As I said above, the single-extruder RepRap design is one of the major obstacles to self-replication. You could imagine an alternative RepRap design that used two different materials for two different nozzles, plasticized using two separate mechanisms, such that either nozzle could print the other nozzle — for example, one material plasticized using heat, and the other plasticized with a solvent.

But multiple extruders is not the only possible solution.

You could also imagine a multi-step manufacturing process, where for example you first use FDM to fabricate a shape in PLA, then make a plaster cast around the PLA and use it to cast, for example, brass. You can find videos of people doing this on YouTube ("lost PLA process").

Or first you fabricate a shape in stainless-steel-filled PLA, and then you sinter it in an oven, or something. There are 3-D printing companies that do this with selective laser melting of stainless-filled thermoplastic.

Or you could use a material whose cradle-to-cradle life cycle includes more than two steps like PLA's melting and hardening. Plaster of Paris, for example, which you can deposit as a thixotropic liquid freshly mixed from powder and water, then allow it to harden. The full cradle-to-cradle process for plaster of Paris involves calcining the resulting hydrated gypsum at 150° to 180° and then grinding the resulting calcium sulfate hemihydrate to a fine powder. As long as the "nozzle" isn't doing the calcining, it won't have any trouble making an alternate nozzle. Which is a good thing, because once you let plaster harden in the nozzle, you'll need to replace it.

A fully autonomous self-replicating system will probably need to be able to make ceramics from clay and fire them, because essentially all industrial processes depend on ceramics. This might be a good place to start.

Open-loop stepper-motor actuation

RepRap-style printers use open-loop stepper-motor control. This results in three problems: imprecision, cost, and unreliability.

Open-loop stepper motors have poor precision (or, let's say, poor bandwidth) for reasons I don't fully understand. This is one of the sources of the problems in the "Imprecision" section above.

The beefy NEMA 23 or NEMA 17 stepper motors used in a RepRap are rare to find at salvage and quite expensive to buy new, because they're mostly used in industrial machinery, not consumer goods. Older 2-D printers used steppers, but much wimpier ones; newer ones use DC motors with closed-loop control.

The printer uses such large motors because the motors have to be strong enough to move the dynamic loads of the printer with basically no effort, because they cannot adjust the effort to the load without feedback. (A certain limited amount of feedback is provided by the variation in slip in the stepper.)

Finally, steppers add unreliability, because if the printer ever encounters resistance and misses a step, all subsequent movements will be offset by the missed step, resulting in a failed print, typically spaghetti.

That stupid gantry geometry

The gantry construction of RepRap-style printers amplifies the problem of imprecision.

Traditional CNC machine tools use a "gantry" geometry in which high-rigidity linear actuators at precisely aligned right angles determine the relative positioning of the workpiece and the cutting tool. With a gantry, you can make precisely parallel and right-angled cuts by hand, without doing a lot of numerical calculation. But gantries require immensely stiff beams and joints to achieve reasonable

precision, linear slides (which are difficult to make and even more difficult to make precise), and linear actuation, which typically adds backlash and therefore imprecision.

Fortunately, less stupid geometries are coming into fashion, including deltabots, parallel SCARA topologies, and polar robots.

Gantry construction synergistically causes problems with open-loop control, because the precision of open-loop control depends critically on rigidity, since the open-loop control can't correct for mechanical deformation any more than it can correct for missed steps.

My favorite robot geometry, the Stewart platform, has not become popular.

Emerging alternatives to RepRap-style 3-D printing

CNC laser cutters are actually far more accessible than 3D printers today, in the sense that there is likely a CNC laser cutter physically closer to you that you can use, and if you make something on it, it will cost you a lot less than doing the same thing on a 3D printer. It's just that it's limited to planar fabrication (as are CNC plasma torch tables, CNC waterjet cutters, and lithography per se.

Things like the Othermill at US$2200 are only a little more expensive than RepRap-descended 3D printers; the PocketNC FR4 is a fully-funded Kickstarter for a US$449 three-axis CNC milling machine, which is cheaper than many 3-D printers. The FR4 is milled from epoxy-glass-fiber composite circuit boards, which are very rigid, light, and cheap, compensating for its stupid gantry geometry (see below for the problems with gantries).

You can see that this FR4 is capable of reproducing its own circuit boards and the parts of its own structure, and has 5-micron precision in all three axes. That is about 20×20×50 = 20000 times better volumetric resolution than any open-loop-control FDM 3D printer.

Norbert Heinz built his 100-micron-resolution CNC machine entirely out of recycled junk, including closed-loop control of the underpowered DC motors he used. In that video you can see him using it to engrave glass with a diamond cutter, cut Styrofoam with a soldering iron, and mill gears from acrylic or aluminum, all of which are materials that are out of reach with FDM.

With FDM, you can't make anything transparent, anything as strong as glass or acrylic or aluminum, or anything as light as Styrofoam. And, at the 100µm precision achieved by both Norbert's junk heap and a thousand-dollar FDM 3-D printer using open-loop control, your gear surfaces suck, and you need a lot of extra clearance to print preassembled gear assemblies. (Although, of course, with CNC cutting, you can't print anything preassembled.)

Norbert's machine is 500×500×?? mm rather than the 200×200×150 of a typical RepRap-derived 3D printer, so it can make things that are about ten times as big.

Michal Zalewski has written is a superb introductory guide to CNC machining and resin casting. Remember what I said about a multi-step manufacturing process? His multi-step process is that he cuts a mold or a positive, casts a mold if necessary, and then casts the part. He gets 2-micron accuracy.

Topics

Error Reporting is Part of the Programmer's User Interface

Kragen Javier Sitaker, 2007 to 2009 (18 minutes)

(This is about the user interface provided to the application programmer, not the user interface the application programmer provides to the non-programmer.)

I have a quibble with the user interface design approach of web.py. Its user interface design slogan says:

"Think about the ideal way to write a web app. Write the code to make it happen."

And in a February 2006 post on the web.py mailing list:

The goal of web.py is to build the ideal way to make web apps. If reinventing old things with only small differences were necessary to achieve this goal, I would defend reinventing them. The difference between the ideal way and the almost-ideal way is, as Mark Twain suggested, the difference between the lighting [sic] and the lightning bug.

This is a very respectable user-interface-design philosophy: start from a conception of the user interface, and then proceed to write the code to make the user interface work. I totally agree with this point of view.

Unfortunately, the user interface for a library doesn't just include the code you have to write to use the library; it also includes implicit coupling created between parts of that code, and the error messages produced when your code is wrong. And web.py doesn't do very well on those latter items, because of hyper-optimizing the code you have to write --- to the point of being deceptive.

The fundamental merit that is somewhat lacking is that of explicitness; explicit is better than implicit, because explicitly stating the relationships in your program makes them both comprehensible to a reader and changeable by a maintainer. Implicit relationships are neither. Also, explicitly stating relationships allows errors in these relationships to be reported in the terms of the user's model (the green code) rather than the implementation model (the yellow code).

An Example of the Problem

I'm developing on top of web.py 0.3, which hasn't been released yet. Here I'm trying to reproduce an example from the docstring of `web.application.request`, but with `web.config.debug` turned on.

```
kragen@thrifty:~/devel/watchdog-git$ python
Python 2.4.4 (#2, Apr  5 2007, 20:11:18)
[GCC 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import web
>>> web.config.debug = True
>>> urls = ("/hello", "hello")
>>> class hello:
...     def GET(self):
...         web.header('Content-Type', 'text/plain')
...         return "hello"
...
```

```
>>> app = web.application(urls, globals())
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/home/kragen/devel/webpy.dev/web/application.py", line 62, in __init__
    module_name = modname(fvars)
  File "/home/kragen/devel/webpy.dev/web/application.py", line 54, in modname
    file, name = fvars['__file__'], fvars['__name__']
KeyError: '__file__'
>>>
```

This error leads me to wonder, "WTF?" This is not an error phrased in terms of anything I passed in to web.py. And it only occurs when `web.config.debug` is turned on.

It turns out that the problem is that, when `autoreload` is true, `web.application.__init__` tries to reload the module that contains the `urls` and `mapping` arguments I'm passing in, and it does so by assuming that the `mapping` is the `globals()` of some module loaded from a file and therefore will contains a `__file__` key that tells where to find that file.

Here's another test case that shows a related problem, this time in a file:

```
#!/usr/bin/python
import web

class hello:
    def GET(self):
        web.header('Content-Type', 'text/plain')
        return "hello"

def main():
    web.config.debug = True
    urls = ("/hello", "hello")
    app = web.application(urls, globals())
    app.request("/hello")

if __name__ == '__main__': main()
```

This gives the following error, which you'll note is several times longer than the program itself --- actually two errors:

```
Traceback (most recent call last):
  File "/home/kragen/devel/webpy.dev/web/application.py", line 186, in wsgi
    result = self.handle_with_processors()

  File "/home/kragen/devel/webpy.dev/web/application.py", line 158, in handle_wito
oh_processors
    return process(self.processors)
  File "/home/kragen/devel/webpy.dev/web/application.py", line 153, in process
    return p(lambda: process(processors))
  File "/home/kragen/devel/webpy.dev/web/application.py", line 457, in processor
    return handler()
  File "/home/kragen/devel/webpy.dev/web/application.py", line 153, in <lambda>
    return p(lambda: process(processors))
  File "/home/kragen/devel/webpy.dev/web/application.py", line 153, in process
    return p(lambda: process(processors))
```

```
  File "/home/kragen/devel/webpy.dev/web/application.py", line 456, in processor
    h()

  File "/home/kragen/devel/webpy.dev/web/application.py", line 68, in reload_mappo
oing
    self.mapping = getattr(mod, mapping_name)
TypeError: getattr(): attribute name must be string

Traceback (most recent call last):
  File "../webpybug.py", line 15, in ?
    if __name__ == '__main__': main()
  File "../webpybug.py", line 13, in main
    app.request("/hello")
  File "/home/kragen/devel/webpy.dev/web/application.py", line 142, in request
    response.data = "".join(self.wsgifunc()(env, start_response))
  File "/home/kragen/devel/webpy.dev/web/application.py", line 196, in wsgi
    result = self.internalerror()

  File "/home/kragen/devel/webpy.dev/web/application.py", line 227, in internalerro
oror
    return debugerror.debugerror()
  File "/home/kragen/devel/webpy.dev/web/debugerror.py", line 305, in debugerror
    return djangoerror()
  File "/home/kragen/devel/webpy.dev/web/debugerror.py", line 293, in djangoerror
    return t(exception_type, exception_value, frames)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 499, in __call__
    return f.go()
  File "/home/kragen/devel/webpy.dev/web/template.py", line 761, in go
    self.output._str = ''.join(map(self.h, self.parsetree))
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 729, in h_for
    out.extend(self.h_lines(i[BODY]))
  File "/home/kragen/devel/webpy.dev/web/template.py", line 757, in h_lines
    return map(self.h, lines)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 700, in h_if
    expr = self.h(i[CLAUSE])
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 621, in h_expr
    item = self.h(i[THING])
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 651, in h_test
    return e(ox) and e(oy)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 621, in h_expr
    item = self.h(i[THING])
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 588, in h_paren
    return self.h(item)
```

```
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 621, in h_expr
    item = self.h(i[THING])
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 635, in h_test
    return e(ox) in e(oy)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 621, in h_expr
    item = self.h(i[THING])
  File "/home/kragen/devel/webpy.dev/web/template.py", line 553, in h
    return getattr(self, 'h_' + item[WHAT])(item)
  File "/home/kragen/devel/webpy.dev/web/template.py", line 676, in h_var
    raise NameError, 'could not find %s (line %s)' % (repr(i[NAME]), i[LINENO])
NameError: could not find 'x' (line 178)
```

Again, I didn't pass in anything called "x"; I never asked web.py anything about "x"; and I didn't pass in any non-string things that could reasonably be used as dict keys. It turns out that in order to get the auto-reloading logic to work, web.py is rooting through the globals of the specified module in order to find the `mapping` argument I passed in, so that it can reload `mapping` from that module after it reloads the module.

This is an example of what is wrong with "magic". The error is rather confusing for the user to debug, at least if they're kind of dumb like me, because the web.py API being called makes all sorts of hidden assumptions about the arguments and the relationships between them. The API leads you to believe that it is called with some arbitrary mapping and a dictionary to look up keys from the mapping, but that's a lie. At least when you have `debug` turned on, it really needs a module and a global variable name in that module, in order to support auto-reloading. It's just sneaky about it, and produces incomprehensible error messages.

The solution, for what it's worth, is to instantiate the `web.application` with `autoreload` turned off:

```
app = web.application(urls, globals(), autoreload=False)
```

The Cost In The Yellow Code

It's not just that this magic makes it harder to write and debug applications on top of web.py. It also makes web.py itself harder to understand. Here's the code in `web.application.__init__` to do this magic:

```
        if autoreload:
            def modname(fvars):
                """find name of the module name from fvars."""
                file, name = fvars['__file__'], fvars['__name__']
                if name == '__main__':

                    # Since the __main__ module can't be reloaded, the module has
```

```
                # to be imported using its file name.
                name = os.path.splitext(os.path.basename(file))[0]
            return name

        mapping_name = utils.dictfind(fvars, mapping)
        module_name = modname(fvars)

        def reload_mapping():
            """loadhook to reload mapping and fvars."""
            mod = __import__(module_name)
            self.fvars = mod.__dict__
            self.mapping = getattr(mod, mapping_name)

        # to reload modified modules
        self.add_processor(loadhook(Reloader()))

        # to update mapping and fvars
        self.add_processor(loadhook(reload_mapping))
```

Here's what I think it would look like if you passed in `module_name` and `mapping_name` directly:

```
    if autoreload:
        def reload_mapping():
            """loadhook to reload mapping and fvars."""
            mod = __import__(module_name)
            self.fvars = mod.__dict__
            self.mapping = getattr(mod, mapping_name)

        # to reload modified modules
        self.add_processor(loadhook(Reloader()))

        # to update mapping and fvars
        self.add_processor(loadhook(reload_mapping))
```

Additionally there's copy-and-pasted code in `web.webpyfunc`. Here's what it looks like now:

```
if not hasattr(inp, '__call__'):
    if autoreload:
        def modname():
            """find name of the module name from fvars."""
            file, name = fvars['__file__'], fvars['__name__']
            if name == '__main__':

                # Since the __main__ module can't be reloaded, the module has
o
o
                # to be imported using its file name.
                name = os.path.splitext(os.path.basename(file))[0]
            return name

        mod = __import__(modname(), None, None, [""])
        #@@probably should replace this with some inspect magic
        name = utils.dictfind(fvars, inp)
        func = lambda: handle(getattr(mod, name), mod)
```

```
        else:
            func = lambda: handle(inp, fvars)
    else:
        func = inp
```

I don't completely understand this, but I think that with a more honest interface, all that code would look more like this:

```
func = lambda: handle(getattr(__import__(modname, None, None, [""]), name),
                      inp)
```

This would enable the removal of `web.utils.dictfind`:

```
def dictfind(dictionary, element):
    """
    Returns a key whose value in `dictionary` is `element`
    or, if none exists, None.

        >>> d = {1:2, 3:4}
        >>> dictfind(d, 4)
        3
        >>> dictfind(d, 5)
    """
    for (key, value) in dictionary.iteritems():
        if element is value:
            return key
```

That's about 30-50 lines of code to implement this single misfeature, which could be removed with a simple interface change.

The (Lack of) Benefit in the Green Code

But how much does all that extra complexity above shorten the code you have to write to use the library? (I've argued above that it actually makes the application code harder to write, although shorter, because it renders important dependencies invisible.)

If the interface were changed as I suggest, the cost to user code would be that instead of writing this:

```
app = web.application(urls, globals())
```

You would write this:

```
import mywebapp
app = web.appreloader(mywebapp, 'urls')
```

Or, if you standardized on the name `'urls'` (which is what most web.py apps call the mapping) it would just be:

```
import mywebapp
app = web.appreloader(mywebapp)
```

But I think that, in this case, explicit is better than implicit.

Global Variables: Another Example

I originally ran into the above problem not because I was manually

setting `web.config.debug` to `True` in Python's REPL, but rather because I was writing post-hoc unit tests for a web app that did (http://watchdog.net/). That app set `web.config.debug` to `True` when it was imported. This violates the normal Python principle that you should be able to import a module without fear, but there doesn't seem to be another solution with web.py. In order to work around some of the bad effects of storing state in global variables, web.py uses `ThreadedDict`s so that HTTP requests in different threads don't interfere with each other.

Reporting Problems in the Terms of the User's Model

Above I said that being explicit instead of implicit makes it possible to "report problems in the terms of the user's model". Here's an example of what that means, from another context, where the user is the programmer in a language, not the caller of a library. Here's a small Python program that forgets to check its inputs:

```
kragen@thrifty:~/devel$ cat > len.py
import sys
print len(sys.argv[1])    # print length of first command-line argument
kragen@thrifty:~/devel$ python len.py  # note: no command-line arguments
Traceback (most recent call last):
  File "len.py", line 2, in ?
    print len(sys.argv[1])
IndexError: list index out of range
```

That's pretty straightforward. It shows you the broken line of code and tells you that what it's doing is trying to index into a list (`sys.argv`), but the supplied list index is off the end of that list. Both the list and the index are objects in the world of the Python programmer, the user of Python; they appear in the program.

How about in C?

In C, we don't have lists; instead we have mostly machine words with associated types. Any connection between the starting address of an array and the number of items in the array is necessarily implicit. (There are things like `valgrind` that can sometimes tell you if you violate array bounds, but the ANSI standard requires the compiler to allow some kinds of pointer arithmetic outside of array bounds.) Consequently the C runtime can't report such an error in the terms of the source C program (that is, the terms of the user of the C compiler); it has to report it in terms of the implementation of C.

So here's the same program in C:

```
kragen@thrifty:~/devel$ cat > len.c
#include <string.h>
#include <stdio.h>

int main(int argc, char **argv) {
  printf("%d\n", strlen(argv[1]));
  return 0;
}
kragen@thrifty:~/devel$ gcc -Wall len.c -o len
kragen@thrifty:~/devel$ ./len
```

```
Segmentation fault
```

That's not a very helpful error message, but we can do better.

```
kragen@thrifty:~/devel$ gcc -Wall -g -O0 len.c -o len
kragen@thrifty:~/devel$ ./len
Segmentation fault
kragen@thrifty:~/devel$ gdb len
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.

This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/lo
oib/tls/i686/cmov/libthread_db.so.1".

(gdb) r
Starting program: /home/kragen/devel/len
Failed to read a valid object file image from memory.

Program received signal SIGSEGV, Segmentation fault.
0x0804837c in main (argc=Cannot access memory at address 0xffffffff
) at len.c:5
5     printf("%d\n", strlen(argv[1]));
```

So now we know what line the error happened at (although this is not always possible in C) but it still isn't telling us that it was a pointer arithmetic error. To find that, we have to descend to the implementation level, which happens to be x86 assembly. Here's `main`:

```
(gdb) disassemble
Dump of assembler code for function main:
0x08048354 <main+0>:    lea    0x4(%esp),%ecx
0x08048358 <main+4>:    and    $0xfffffff0,%esp
0x0804835b <main+7>:    pushl  0xfffffffc(%ecx)
0x0804835e <main+10>:   push   %ebp
0x0804835f <main+11>:   mov    %esp,%ebp
0x08048361 <main+13>:   push   %edi
0x08048362 <main+14>:   push   %ecx
0x08048363 <main+15>:   sub    $0x10,%esp
0x08048366 <main+18>:   mov    0x4(%ecx),%eax
0x08048369 <main+21>:   add    $0x4,%eax
0x0804836c <main+24>:   mov    (%eax),%eax
0x0804836e <main+26>:   mov    $0xffffffff,%ecx
0x08048373 <main+31>:   mov    %eax,0xfffffff4(%ebp)
0x08048376 <main+34>:   mov    $0x0,%al
0x08048378 <main+36>:   cld
0x08048379 <main+37>:   mov    0xfffffff4(%ebp),%edi
0x0804837c <main+40>:   repnz scas %es:(%edi),%al
0x0804837e <main+42>:   mov    %ecx,%eax
0x08048380 <main+44>:   not    %eax
0x08048382 <main+46>:   dec    %eax
0x08048383 <main+47>:   mov    %eax,0x4(%esp)
```

```
0x08048387 <main+51>:   movl   $0x80484b8,(%esp)
0x0804838e <main+58>:   call   0x8048290 <printf@plt>
0x08048393 <main+63>:   mov    $0x0,%eax
0x08048398 <main+68>:   add    $0x10,%esp
0x0804839b <main+71>:   pop    %ecx
0x0804839c <main+72>:   pop    %edi
0x0804839d <main+73>:   pop    %ebp
0x0804839e <main+74>:   lea    0xfffffffc(%ecx),%esp
0x080483a1 <main+77>:   ret
End of assembler dump.
```

OK, but where are we?

```
(gdb) x/i $pc
0x804837c <main+40>:    repnz scas %es:(%edi),%al
```

`repnz scas` is what `gcc` compiles `strlen` to, so the problem is the argument of `strlen`. It reads from where `%edi` points. So `%edi` must be pointing somewhere bogus.

```
(gdb) x/a $edi
0x0:    Cannot access memory at address 0x0
```

Aha, `%edi` is a null pointer. So we probably passed a null pointer to `strlen`. Where did that come from?

```
(gdb) x/a argv
Cannot access memory at address 0x3
(gdb) p argv
Cannot access memory at address 0x3
(gdb) info locals
No locals.
(gdb) info args
argc = Cannot access memory at address 0xffffffff
```

Looks like our `argv` got horked. Oh well, too bad. We know the problem is that `argv[1]` was null.

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

There is a striking family resemblance between the debugger session above and the process required to comprehend the web.py error:

```
TypeError: getattr(): attribute name must be string
```

It took me something like half an hour to figure out what was going on there. Probably if I were smart, it would have been more like five minutes, but in either case, the entire class of such hard-to-diagnose errors could be avoided by interface design that's explicit enough to report errors in the terms of the user's model.

References

In Economizing can be Penny-Wise and Pound-Foolish, Reginald

Braithwaite describes how you can divide code into "red code", which you don't understand, "yellow code" which you understand but which belongs to the solution domain rather than the problem domain, and "green code" which is purely problem-domain code.

Explicit is better than implicit is one of the design principles of Python. It's not an absolute law, but a design heuristic, helpful for the reasons explained here. You can read a little bit of Paul Prescod trying to explain it to Paul Graham at http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg01587.oohtml.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Programming languages (p. 3656) (47 notes)
- C (p. 3359) (28 notes)
- Python (p. 3671) (27 notes)

IMGUI programming compared to Tcl/Tk

Kragen Javier Sitaker, 2018-12-24 (updated 2018-12-31) (8 minutes)

I wrote this example Tk program last night as an example showing how easy Tk makes GUI development:

```
scale .a -variable x -orient horizontal
entry .b -textvariable x
label .c
pack .a .b .c -side top
proc updatec {a b c} {global x; .c configure -text [expr 1.0 / $x]}
trace add variable x write updatec
```

(This leaves out the long and winding incremental road to getting it to work, which is a strong point of Tcl: it's interactive, everything is a string, and everything gives you helpful error messages. Just that the language semantics is a shambles.)

I think you could quite reasonably do this same application with an IMGUI framework in C, and maybe it would even be less code, because you don't have separate component states.

Something like this, maybe:

```
static int x;
ig_scale(ig_int_model(&x)); ig_horizontal();
ig_entry(ig_int_text_model(&x));
float r = 1.0 / x;
ig_label(ig_float_text_model(&r));
```

And yeah, it's actually about 25% less code than the Tk version.

The idea for the layout is that the framework does a pass over the UI definition function before the drawing pass to find out what sizes everything is requesting and how they're connected together. (It also does a pass over the UI definition function for each event, using the layout from the previous frame.) Then it computes the new layout and does a final pass to paint everything.

Tk lets you pack things on all four sides of your window, which allows you to avoid introducing frames in many cases. I don't think that's necessary here; you just need horizontal boxes and vertical boxes, for which you need to be able to `ig_hbox()` or `ig_vbox()` to start a new nested box, or `ig_end()` to end one of them. Additionally we can provide a default tabular setup, where adjacent sibling hboxes or vboxes have their inner items aligned by default — you can add an extra level of nesting to avoid this if necessary.

The `ig_*_model` functions wrap a raw pointer to the relevant type in a model struct that is passed by value to the widget function in question; presumably it contains a getter, a setter, and a userdata, although you could imagine that for strings it might have slicing functions or something.

Things like layout options can be provided with things like the `ig_horizontal()` call above; at runtime it will error if the current object doesn't have an orientation (like scales and scrollbars do).

The full set of configuration options for a Tk scale widget is as follows:

activeBackground background bigIncrement borderWidth command cursor digits font foreground from highlightBackground highlightColor highlightThickness label length orient relief repeatDelay repeatInterval resolution showValue sliderLength sliderRelief state takeFocus tickInterval to troughColor variable width

You could imagine providing these with named member initializers in a struct, but that requires an extra line of code to declare the struct and doesn't work well for options like `scale -to`, whose default is 100 but for which 0 is a valid value. So probably supplying them with functions (some of which perhaps take struct or multiple arguments) is better.

However, if those functions are to come after the widget name, as they should, then the actual layout or painting must happen after the widget function itself returns — it must be deferred. That also means that these option functions can't change the function's return value, which is relevant for, e.g., tab property pages or menu items, which really benefit from being able to return a boolean.

How Dear ImGui does it

For that matter, buttons also benefit from being able to return a boolean:

```
if (ImGui::Button("Button"))
    clicked++;
```

Dear ImGui has a `SameLine()` function which lays out the following text (or presumably other widget) on the same line, rather than a separate line, as per default. A possible hbox-oriented alternative would be a `ig_endl()` function which ends the current hbox and starts a new one. This is still more modeful, but it avoids having multiple mechanisms for the same purpose, and it avoids needing 5 `SameLine` calls to get 6 things into an hbox.

(Dear ImGui also has a horizontal mode.)

This treenode thing is a thing I really like about Dear ImGui:

```
if (ImGui::CollapsingHeader("Configuration")) {
    if (ImGui::TreeNode("Configuration##2")) {

        ImGui::CheckboxFlags("io.ConfigFlags: NavEnableKeyboard", (unsigned int *o
o)&io.ConfigFlags, ImGuiConfigFlags_NavEnableKeyboard);
```

Radio buttons share a model and specify a value:

```
static int e = 0;
ImGui::RadioButton("radio a", &e, 0); ImGui::SameLine();
ImGui::RadioButton("radio b", &e, 1); ImGui::SameLine();
ImGui::RadioButton("radio c", &e, 2);
```

Dear ImGui requires a PushID/PopID call in loops:

```
for (int i = 0; i < 7; i++)
{
    ImGui::PushID(i);
```

```
            …
            ImGui::PopID();
        }
```

It identifies clickable widgets with an "ID stack", which I guess is sort of like a pathname; windows and tree nodes push onto the ID stack. Within a window it normally uses the button (or whatever) label and hashes it, so there are hacks you have to use if you want to animate the label or whatever; the `"Configuration##2"` in the example above is one such hack — the 2 isn't displayed, but forms part of the ID. As a result, you need to call `TreePop` to end a treenode.

Within the implementation of treenodes, to find out if the treenode is open, TreeNodeBehaviorIsOpen fetches from `window->DC.StateStorage`:

```
    is_open = storage->GetInt(id, (flags & ImGuiTreeNodeFlags_DefaultOpen) ? 1 : 0) != 0;

    …

    if (toggled)
    {
        is_open = !is_open;
        window->DC.StateStorage->SetInt(id, is_open);
    }
```

StateStorage is a per-window sorted ImVector of key-value pairs, where the values are unions. It's amusing to me that the id is hashed with a CRC32 of the string, but then they don't bother to use a hash table to store it, instead resorting to binary search; but they expect to do insertions, as opposed to updates, quite rarely. Still, you'd think that would favor cuckoo hash tables rather than binary search.

The ID construct results in weird things where clicking on one button will activate another one, and so forth. Presumably it could result in a case where this happened even if the buttons had different IDs just because of a hash collision.

A convenient thing about Dear ImGui is that most (all?) of the widgets take printf format strings and varargs.

Dear Imgui labels everything with its ID string by default, since for usability you probably want to know what you're setting anyway.

Perhaps the other thing to beat in this space is REBOL's Visual Interface Dialect: http://rebol.com/docs/view-guide.html. An example from https://en.wikipedia.org/wiki/Rebol:

```
view layout [text "Hello world!" button "Quit" [quit]]
```

Or in REBOL R3-GUI:

```
view [text "Hello world!" button "Quit" on-action [quit]]
```

Examples from http://rebol.com/docs/easy-vid.html:

```
style yell tt 220 bold underline yellow font-size 16
yell "Hello"
```

```
yell "This is big old text."
yell "Goodbye"

vtext bold "Wild Thing" effect [gradient 200.0.0 0.0.200]
```

Here %. is the current directory:

```
vh2 "File List:"
text-list data read %.
button "Great!"
```

Here a "pair" specifies geometry:

```
button 200 "Big Button"
button 200x100 "Huge Button"
image %palms.jpg 50x50
image %palms.jpg 150x50
```

A file filtered to purple provides the backdrop for a button, behind the "Button" text:

```
button "Button" %palms.jpg purple
```

This action assigns `value` to a refinement of another widget and then invokes `show` on the widget:

```
slider 200x16 [p1/data: value show p1]
p1: progress
```

At least in R3-GUI, the layout is tabular: http://www.rebol.com/r3/docs/gui/panels.html explains that this has four columns:

```
view [
    panel 4 [
        button "First"
        button "Second"
        button "Third"
        button "Fourth"
        button "Fifth"
        button "Sixth"
    ]
]
```

Topics

- Programming (p. 3658) (286 notes)
- Programming languages (p. 3656) (47 notes)
- Syntax (p. 3738) (28 notes)
- Graphical user interfaces (p. 3489) (23 notes)
- BubbleOS (p. 3352) (17 notes)
- Immediate-mode GUIs (p. 3515) (8 notes)
- Tcl/Tk (2 notes)

Parallel DFA execution

Kragen Javier Sitaker, 2017-04-18 (9 minutes)

I was reading Raph Levien's notes on "rope science" in his "xi" editor. He's talking a lot about monoid cached trees for applications like maximum line width computation, height computation, parenthesis matching, detection, and word wrapping; the objectives are to enable these computations to be done in parallel (across multiple cores), incrementally, and lazily.

It occurs to me that most of these are kind of special cases of a generalized DFA monoid, and using the standard parallel prefix-sum algorithm on the DFA monoid might provide a more convenient way to express these computations.

Consider this simple example of a comment-tagging NFA, in Perl regexp syntax extended with `@x` to tag the just-matched character with `x`, using `N` for non-comment and `C` for comment:

```
([^#]@N|#@C([^\n]@C)*(\Z|\n@N))*
```

This works out to a fully deterministic regexp, in the sense that you never have more than one state live after a given character. If we unpack it from the regexp syntax into traditional programming outline syntax, it looks like this:

```
repeat:                 # 1
    either:
        match [^#]
        tag N
    or:
        match "#"
        tag C
        repeat:         # 2
            match [^\n]
            tag C
        either:
            match EOF
        or:
            match "\n"
            tag N
```

Despite being 15 lines of code, and not counting the final state, this FSM only has two states, marked above with the comments "# 1" and "# 2". The state diagram looks like this (pipe to `dot -Tx11` from the graphviz package to see):

```
digraph nc {
    rankdir=LR;
    LR_1 [ shape=circle, label="1" ]; LR_2 [ shape=circle, label="2" ];
    EOF [ shape=doublecircle, label="" ];

    LR_1 -> LR_1 [ label="[^#]\nN" ];   LR_1 -> LR_2 [ label="#\nC" ];
    LR_2 -> LR_2 [ label="[^\\n]\nC" ]; LR_2 -> LR_1 [ label="\\n\nN" ];
    { LR_1 LR_2 } -> EOF [ label="EOF" ];
```

```
}
```

XXX why is state 2's EOF transition sort of explicit while state 1's isn't?

Now, obviously, you can run this on a text sequentially, starting from a known initial state, tagging each of the characters with either N or C as you go. At any given point in the text, your state is either 1 or 2.

However, you can *also* run it from an *unknown* initial state. In this case, your state at any given point in the text is a *function* from the initial state to the current state. Initially this function is { 1: 1, 2: 2 }. And as you go, you tag each character, not with N or C, but with a *function* from the initial state to the tag. For example, if the first character is "q", then you tag it with { 1: "N", 2: "C" }, and your state doesn't change. But if then you match a "#", your state function changes to { 1: 2, 2: 2 }, which is to say, always 2, and you tag it with { 1: "C", 2: "C" }, which is to say, always "C".

If you run this algorithm over a block of data taken from the middle of a long file, you will end up with some final state function at the end of the block of data. For this DFA, it will be either { 1: 1, 2: 2 }, { 1: 2, 2: 2 }, or { 1: 1, 2: 1 }.

(If you do this with an NFA instead of a DFA, your result will be a binary relation rather than a function.)

If you break the file up into many blocks and run it in parallel over each block, you will compute such a function for each block. By composing these functions, you can compute such a function for longer runs of the file. For example, if block 39 comes out to { 1: 1, 2: 2 } and block 40 comes out to { 1: 2, 2: 2 }, then the concatenation of blocks 39 and 40 comes out to { 1: 1, 2: 2 }.

If you compose these blocks into a balanced binary tree, you can then compute the function for the whole file by propagating these functions up the tree; functions are, of course, a monoid under composition. Also, though, this allows you to take a known initial state and then efficiently propagate it back *down* the tree, left to right, to every character in the file.

This is the standard parallel prefix sum algorithm, applied to DFA execution.

If you were doing a similar kind of tagging in an NFA, you would probably want to only apply the tags that didn't belong to branches of the possibility tree that were ultimately discarded. This involves propagating information *backwards* as well in the down-the-tree stage. I think this simply involves rolling back the things that led to intermediate states that ultimately mapped to a null set of states.

You could think of this transformation of the DFA from a computation on states into a computation on functions from states to states as a kind of abstract interpretation with non-standard semantics of the DFA. You can do the same kind of abstract-interpretation trick with computational models more powerful than a DFA, although you lose the bounded-space guarantee the DFA gives you. For example, consider this parenthesis-counting automaton:

```
repeat:
    either:
        match "("
```

```
        n++
    or:
        match ")"
        n--
        either:
            n >= 0
        or:
            n < 0
            tag X
    or:
        match [^()]
```

It tags "X" whenever there's an unmatched ")". You can run it in the standard way with an initial concrete value of `n` such as 0, and it will tell you if there are mismatched parentheses in your text, its state at any given position being some concrete value of `n` such as 3. Or you can run it with an initial abstract value such as n_0, and its values at different positions will be further abstract values such as $n_0 + 3$ — or, you could say, $\lambda n_0.n_0 + 3$.

```
digraph pp {
    LR_1 [ shape=circle, label="" ];  EOF [ shape=doublecircle, label="" ];

    LR_1 -> LR_1 [ label="(\nn++" ];
    LR_1 -> LR_1 [ label=")\nn--\nn>=0" ];
    LR_1 -> LR_1 [ label=")\nn--\nn<0\nX" ];
    LR_1 -> LR_1 [ label="[^()]" ];
    LR_1 -> EOF [ label="EOF" ];
}
```

XXX graphviz is sucking at laying out those edges and labels

Although this automaton has an infinite set of possible states, as well as possible mappings from state at the beginning of a block and state at the end of a block, and representing one of them could in principle consume an arbitrarily large amount of space, you can apply exactly the same approach to composing those functions into a monoid tree. And, as it happens, these particular mappings have a relatively compact representation, all fitting the schema $\lambda n_0.n_0 + k$.

It seems to me that to make the abstract interpretation tractable, you need some kind of limitation on the power of the language — you don't want to have to solve the Halting Problem for one of the blocks. I suspect that it's probably sufficient to forbid loops that don't advance through any input, but I'm not sure.

The objective here is to have a scripting language in which you can conveniently express any of the monoid computations Raph talked about in "rope science", including word-wrap and the like, and that also provides some kind of evaluation efficiency guarantee. You'd like to be able to compute, for example, that if at character 512 the carriage position was between 110 and 170 pixels, then the word breaks thereafter will be at characters 547, 610, 665, ..., and that at character 1024 the carriage position will be 212; while if it was between 170 and 220, they will be at 542, 604, 665, ... and the carriage position at character 1024 will still be 212. And you'd like to be able to compute that automatically from a word-wrapping script

written in a backtracking language with numerical variables.

Parenthesis matching, however, is harder — the case where you want to not just count parentheses, but also blink the matching parenthesis. You need a stack, and the effect of running a paren-matching script over a section of the file will be to pop some set of parens from the stack (possibly requiring them to be of the right type) and push some others. Such functions can still be composed into a monoid tree, of course. (If you want the parenthesis locations to be integer offsets from the beginning of the file, maybe one state variable should be the current offset.)

Bjoern Hoehrmann has been working on an algorithm called "parselov" which compiles a context-free grammar to a finite-state automaton that approximates the CFG using a limited-depth parse stack (one stack item, I think).

Topics

Logarithmic maintainability and coupling

Kragen Javier Sitaker, 2015-11-23 (7 minutes)

In theory, with ideal design, the functionality of a piece of software grows exponentially with the amount of code in it: each new piece of code can take advantage of the entire existing set of capabilities, and so adding a line of code to a system adds capabilities proportional to those of the previous codebase.

In practice, this never happens. In a few hundred or a few thousand lines of code, we can write a hypertext system, or a ray tracer, or a full-text search engine, or a filesystem, or an operating-system kernel, or an interpreter or compiler for a high-level language, or a GUI, or a networking protocol, and so on. Naively extrapolating, we would expect that in a hundred thousand lines of code, you could go far beyond these capabilities, into things that seem like utterly alien technology — let alone what you could do in a million or ten million lines of code.

You could surely argue that this is somewhat ill-founded: it isn't clear how to reduce "amount of functionality" to a scalar quantity. But in fact, regardless of how you choose to define it, it's clear that this isn't happening: our existing computing systems routinely contain millions of lines of code, and rather than seeming like alien technology from the future, they often fail to work at all, and when they do, they often barely work.

Also in theory, the difficulty of writing a piece of software incrementally, piece by piece, should be proportional to the square of the amount of code in it, because the number of potential pairwise interactions between lines of code increases proportional to that square. This is often cited when hackers try to convince one another of the merits of simplicity.

In practice, however, the observed exponent seems to be around 1.05 per thousand lines of code: the basic COCOMO model estimates development effort at 2.4 person-months per KSLOC**1.05. So:

- a thousand-line project should take 2.4 person-months (410 lines of code per person-month), while
- a ten-thousand-line project should take 27 person-months (370 lines of code per person-month),
- a hundred-thousand-line project should take 302 person-months (330 lines of code per person-month),
- a million-line project should take 3400 person-months (290 lines of code per person-month),
- a ten-million-line project should take 38k person-months (260 lines of code per person-month), and
- a hundred-million-line project should take 430k person-months (230 lines of code per person-month).

These two departures from theory are connected! They have to do with the density of connectivity in the codebase. To escape the trap of quadratic complexity, only a tiny fraction of the possible

interactions in the codebase can exist — a line of code invoked by tens of thousands of callers probably cannot change its behavior, even for the better, without introducing bugs. By the same token, a subroutine cannot depend on too much about the behavior of too many other subroutines: it becomes too likely to break when one of the others changes.

So, just in order to be able to keep writing those 10 lines of code per day in that hundred-million-line project, you end up duplicating facilities that exist elsewhere, perhaps in a restricted or more efficient form. You may not even know they exist; certainly you can't depend on them to retain their existing behavior and performance, or not to have behavior or performance in corner cases that conflicts with your needs. Even if not, they may be in a language you can't invoke conveniently or contain subtle dependencies on control structure, runtime facilities, or resource usage that you can't afford and perhaps don't even know about.

And thus it is that the promise of exponential growth in software capability with growing effort is lost.

But what if we were to bite the bullet? What if we were willing to accept that adding a line of code to a hundred-thousand-line system was going to take 100 times as long as adding it to a ten-thousand-line system? If we could get back the exponential growth, would it be worth it?

In terms of flexibility for experimentation, *no*. If it takes you a month to write a piece of code that ends up being five lines, *and then you throw it away*, you surely would have been better served by writing that throwaway code in some other way. Even if it worked out to be 200 lines of code instead of five, that would have been a much better bargain.

But in terms of building the most powerful system, *yes*. If a thousand-line fully-coupled system takes you, say, 100 days and has functionality of, say, One OTCC†, then we can predict the following:

- a two-thousand-line system will take you 400 days and have a functionality of, say, two OTCCs;
- a five-thousand-line system will take you 2500 days (7 years) and have 16 OTCCs of functionality (at which point you're still behind where you would have been by just writing 25 single-OTCC systems);
- a ten-thousand-line system will take you 27 years and have 512 OTCCs of functionality, at which point you are beating where you would have been with 100 single-OTCC systems by a factor of five;
- a twenty-thousand-line system will take you 110 years and have 500k OTCCs of functionality, at which point you are beating where you would have been by writing 400 single-OTCC systems by a factor of 11;
- a fifty-thousand-line system will take you 680 years and have 560 tera-OTCCs of functionality, at which point you are beating where you would have been by writing 2500 single-OTCC systems by a factor of 200 billion. Yet you have been writing, on average, about a line of code per week.

† OTCC is a small C compiler written by Fabrice Bellard for the IOCCC, and it's an exemplar of powerful functionality condensed into a small package. In the

absence of any kind of reasonable unit for software power, it's my best approximation of the maximum amount you can get done in a thousand lines of code.

Even though I sort of pulled them out of my butt, these numbers are sort of discouraging, even though they suggest that we can do dramatically better at software development by improving reuse. The problem is that this level of reuse has such a high price tag that, under these assumptions, you don't start to break even for decades. It's hard to imagine that the correct numbers will make this look much better.

Topics

- Programming (p. 3658) (286 notes)
- Small is beautiful (p. 3714) (40 notes)
- Utopias: proposals unlikely to be realized for improving things (p. 3767) (19 notes)
- Program design (p. 3654) (11 notes)
- Project management

Cycle sort

Kragen Javier Sitaker, 2013-05-17 (1 minute)

Cycle sort is an $O(N^2)$ in-place sorting algorithm that only performs N writes to the original array in the worst case, handy if that array is stored somewhere very costly such as Flash. It's $O(N^2)$ because it eventually compares every element to every other element, twice, to find that element's position.

It seems to me that it's usually possible to do better, even under the constraint of only performing N writes. Of course, if you have an unlimited amount of RAM, you can just read the items into RAM, sort them, and write them back in order. That's why the "in-place" qualifier on cycle sort is important: it uses only O(1) space, while doing the RAM sort uses O(N) space.

What about the middle ground? What if you can afford O(log N) or O(sqrt(N)) memory? Is there a sorting algorithm that does better than the $2N^2$ comparisons of cycle sort, while maintaining the O(N) writes to the original array?

Topics

lattices, powersets, bitstrings, and efficient OLAP

Kragen Javier Sitaker, 2014-04-24 (17 minutes)

Some thoughts that occurred to me on the bus home about using bit-twiddling tricks to speed up lattice operations. The original genesis of the idea was the old idea that it's a shame that Unix doesn't have "sub-users" that have the same relationship to users that the users have to `root`, whose name is suggestive of the idea of a hierarchy of users, something that was never added to Unix. To implement such a thing, you'd ideally want to substitute a "user is equal to or has power over" check for the usual "user is equal to or is superuser" check, and you'd like it to be efficient enough that it doesn't slow down the operations it needs to guard.

As it happens, the uid of `root`, 0, has a special bitwise relationship to other user IDs: it is, in some sense, the AND of all the other possible user IDs. The AND of any uid with `root`'s uid will equal `root`'s uid. It's as if each 1 bit in your uid represented some restriction, and `root` is the uid with no restrictions. (And `nobody`, traditionally uid -1, is the uid with all possible restrictions, which is nicely symmetrical. Although in this case this would mean `nobody` was subject to the whims of any user at all, which might not actually be what you want, but whatever.)

So you'd substitute the check `(actor_uid & actee_uid) == actor_uid` for the check `!actor_uid || actor_uid == actee_uid`, which would be exactly as efficient; but you'd have to have some kind of magical way to assign uids to regular users so that you didn't accidentally end up with one regular user being superuser over another one, or over some sub-user of another one.

Is there such a magical way to assign uids? That's what this essay explores. I haven't found a universal solution, but I've found some solutions for some interesting cases.

This probably isn't actually practical for Unix kernel security policy, but it has other possible applications; for example, in databases, often the last step of query evaluation is to sift through a potentially large number of candidate records produced in some inconvenient order by index-traversal operations, filtering out the small number of records that actually match the query. In particular, in OLAP applications, there is often no index that can reduce the number of records adequately.

This technique is not related to bitmap indexing or compressed bitmap indexing.

Lattices and powersets

Any arbitrary finite lattice L is a sublattice of the powerset of some finite set S, with intersection and union of the powerset lattice being almost the meet and join operations (or vice versa, but I'll be using almost-intersection for meet and almost-union for join). The constructive proof is something like this: optionally, take the transitive reduction of L, removing the redundant edges; then take the remaining edges to be S. Represent each element E as the set of

edges through which E is reachable from the minimum, that is, the union of all paths from the minimum to E.

Clearly S might need to be about the size of L --- consider the worst-case scenario where your lattice is a totally ordered set. You'll need one new element of S for each element of L other than its minimum.

Are there cases where S needs to be larger than L? I'm not sure.

This construction is clearly not optimal, in the sense of finding a smallest possible S for any L; if you have, for example, a min, a max, and 6 incomparable items in between, you can represent represent these 6 items as {0, 2}, {0, 3}, {0, 4}, {1, 2}, {1, 3}, and {1, 4}, needing only 5 items in S rather than 6. In cases like this, the construction above gives O(N) elements, while O(log N) elements is possible, as I'll explain below.

I said "almost union" and "almost intersection". The meet and join operations are not quite intersection and union, because the intersection and union operations may produce a powerset element that doesn't correspond to an element of L; in this example, intersecting {0, 2} and {0, 3} gives you {0}, not {} as it should. I assert without even the handwaviest sketch of a proof that you can find a unique largest subset or smallest superset in any such case, or at least that you can choose your representation such that this is true, but I don't really care that much because mostly I'm interested in the poset operations rather than the lattice operations.

Bitstrings and powersets

The bitstrings of length N represent the powerset of a set of N items, with a 1 bit at position i representing that item i is present, 0 representing the empty set, bitwise complement representing set complement, bitwise AND representing intersection, and bitwise OR representing union.

The great advantage of bitstrings on computers is that you get a lot of parallelism for free; since bit-serial computers went out of style in the late 1950s (excepting the Clock of the Long Now) and until we move to FPGAs with arbitrary bit-length operands, normal computing hardware can take the AND, OR, or NOT of an entire word of bits as quickly as a single bit; depending on the hardware, this gives 8, 16, 32, 64, 128, or even 256-way parallelism, on top of whatever you get out of multicore, which has been exploited to good effect in bitwise DES key cracking programs, among others. (And constant-time AES-CTR implementations!)

But you still have the problem of encoding your desired lattice into bitstrings that encode the desired properties.

One-hot encoding

The simplest kind of lattice is one where all the elements, except the min and max, are incomparable; that is, the meet or join of any two elements that are not the min or max is the min or max. You can encode this as in the naïve version above: assign each of those elements a unique bit position that gets 1. For example, you can represent five such incomparable elements as 00001, 00010, 00100, 01000, and 10000. Electronics people call this "one-hot" encoding.

Note that this follows the usual lexical ordering of bits.

This is okay as far as it goes, but it's wasteful of space once you get

past five elements, because it takes linearly as many bits as you have elements. But for a logarithmic encoding, you can use the Cartesian product.

Cartesian product encoding

If you have a a bunch of such incomparable elements, then instead of using a single one-hot bitstring for all of them, you can chop the bitstring up into chunks, two to four bits per chunk, and use one-hot encoding independently in each chunk. With six bits, say, you have two three-bit chunks, or fields, each of which can one-hot encode three possibilities, so you have nine possibilities in all:

```
001001 001010 001100
010001 010010 010100
100001 100010 100100
```

instead of the six you'd get with straight one-hot encoding, or the eight you'd get with three two-bit fields, or one four-bit field and one two-bit field. It turns out that three bits per field is optimal in this sense, because log 3 / 3 > log 2 / 2, a little. This way you can get 2 * 3^10 = 118 098 unrelated elements, plus max and min, out of a 32-bit word, rather than 32. 118098 is bigger than 32.

In the limit as the number of bits gets big, this gives you log 3 / 3 bits-or-nats per bit, or 0.53 bits per bit; that is, 47% of the space is a tax imposed by the requirements of the representation. We'll see later that there's a more efficient alternative.

Cartesian hierarchies

But that's not all! If you fill one of those fields with all 0s or all 1s, you get an extra min or max element for just that one field. In particular, if you wanted to give Unix users the ability to create subusers, you could assign, say, the first 18 bits to the normal userid, while leaving the other 14 bits set to zero. This would give you 3^6 = 729 independent users, each of whom could create 2 * 3^4 = 162 sub-users.

In general, by setting all but the first N fields to all-zeroes, you get a sort of "Nth-level `root` user", who has authority over all the users whose uids start with its non-zero fields. In an OLAP context, this could correspond to the first N levels of a hierarchical dimension: for a datetime field, for example, perhaps the first 4 bits indicate a year, the next 4 a month, the next 10 a date within the month, the next 7 an hour, the next 12 a minute, and the next 12 a second, 49 bits in all.

Orthogonal dimensions

As the datetime example suggests, though, once you have separate fields, it makes sense to query on any subset of them: perhaps you're more interested in what happens at 8 AM each day, rather than 8 AM on a particular day. So the Cartesian-product approach gives you not just hierarchy but orthogonality.

Groups, in the Unix security context

In some cases, you could replace the notion of a "group" with a user that is subordinate to *multiple* users, so that any of those users can

write to its files (a common use of groups), but it can't affect any of those users. But making that work may require knowing about it ahead of time when you're assigning uids; unless you assign one bit position to each distinct user (quite doable these days, now that we usually have less than 64 people using a single computer) you wouldn't be able to construct groups containing arbitrary sets of people without changing their uids.

In particular, you could reasonably partition the uid space into one or more cross-cutting hierarchies of grouping, and each group (or intersection of groups) would have its own "administrator" with a bunch of zero bits, as well as its own "group" with the corresponding 1 bits.

Gray code

Consider, instead, the case of searching for other users of a smartphone app in geographical proximity to you. If you tile the geographical area in question with imaginary tiles, you mostly want to find people in the same tile as you; but if you're close to a tile boundary, you may also want to find people in up to three neighboring tiles. It would be nice to be able to do this query efficiently.

If you represent the tile coordinates in Gray code, then as you move along either axis, at most one bit changes as you cross a tile boundary. And you can calculate which one it is. If you simply take the AND of the coordinates of the up-to-four tiles in your neighborhood, you obtain a set of bits that you can then compare to others' candidate coordinates.

You can do Gray code in arbitrary bases, such as base 3.

A similar approach can be used to query for temporal proximity.

Binomial identifier assignment

As I mentioned earlier, assigning unrelated identifiers in 32 bits using cartesian product of one-hot fields only gets you 118098 unrelated identifiers. But you can get 16c32 = 601 080 390 unrelated identifiers by taking all the 32-bit values that have 16 1 bits and 16 0 bits: a binomial coefficient. If you use this approach to assign unrelated uids, you can use 12 bits to get 924 unrelated uids, rather than needing 18 bits to get 729 of them.

The binomial approach thus has a dramatically lower tax than the one-hot and Cartesian approaches (it costs you 36% of your effective bits at 4 bits, 23% at 8, 18% at 12, 15% at 16, 11% at 24, 9% at 32, 6.5% at 48, and 5.3% at 64), but at the cost of eliminating orthogonality and hierarchy. It seems like the tax may asymptote to zero with large numbers of bits, but I'm not sure. It definitely gets below 0.3% at 1928 bits, but that's already big enough to be sort of irrelevant.

It's already more efficient at 4 bits; you can represent 6 incomparable elements in 4 bits as 0011, 0101, 0110, 1001, 1010, and 1100, rather than the 5 needed for a Cartesian product of one-hot encodings.

All-or-nothing edge families

The efficient encoding schemes mentioned above take advantage of a particular property of families of edges in the transitive-reduced

lattice to encode them more efficiently: that for any element of the lattice, within some subset of the edges, either none, exactly one, or all of the edges are on a path between that element and the infimum. That is, those edges are mutually exclusive --- within a certain sublattice, except for its supremum; and each is paired with another edge whose representation would be redundant. This is what allows us to safely use such schemes.

This suggests an approach for finding more efficient representations of arbitrary lattices: begin with the basic construction, then look for families of edges with this property. Will it work? Is it optimal? I don't know.

Hashing

In database-query cases, it may be adequate to reject *most* of the candidate records from the index operations, rather than *all* of them. In this case, you may be able to hash a larger identifier space into a smaller one, which you then represent with approaches such as the one-hot, cartesian-product, and binomial approaches mentioned earlier.

There's a curious relationship here between binomial assignment, bloom filters, and approximate homomorphic processing, that I haven't fully explored. For example, if many-bit binomial names are uncorrelated, the AND or OR of two or three of these names is likely to have several bits still, respectively, set or cleared; and so you should be able to use that combined value as a first-pass to sift for candidate matches. The expected number of candidate matches is probably dominated by the improbable case that the names cancel badly, in which case you get an exponential number of false positives.

Query criticism

You could argue that this technique is not appropriate for query processing because it inflates the size of your index data: simple bitfields give you 1 bit per bit, while this gives you 0.53 bits (with Cartesian product of one-hot encodings) or between 0.64 and 0.95 bits per bit (with binomial assignment), all in order to allow any item to potentially represent, in effect, a simultaneous bitmask and value: a combination of a selection of a set of fields, with required values for those fields.

But in the context of querying a database, the records, and especially the records' index entries, do not need to waste space (whether it's 47% of your space, 36%, or only 5%) on something that is only useful in a query term. It would make more sense, instead of testing `(a & b) == a`, to test `(am & b) == ar`, with a separate bitmask and expected result.

This approach makes more sense in applications where the objects you're comparing really are the same kind of object, like when you have two Unix processes one of which wants to `ptrace` the other.

CAMs

Content-addressable memories, which are mostly used in routers' routing tables and virtual memory translation lookaside buffers, include the `(a & b) == c` query operation as its fundamental operation, returning all the `b` that satisfy it.

NTRU

Could this be useful for implementing NTRU? I have no idea because I don't know how NTRU works.

Credits

Thanks to Darius Bacon for illuminating discussions on this essay!

Topics

Hall-effect Wheatstone bridges for impractical steampunk electronic logic gates

Kragen Javier Sitaker, 2019-04-24 (2 minutes)

Some years ago I thought I had found a way to build electrical logic gates through magnetoresistance using 19th-century materials science. The phenomenon in question is the altered resistance of ferromagnetic materials when magnetized, by about 5% depending on the orientation of the field, as demonstrated by Kelvin in 1856. I concluded that, by using a Wheatstone bridge, you could get large amplification and inversion from this effect. What I didn't realize at the time was that, as my calculations later showed, the extremely high self-inductance of the ferromagnetic conductors would limit it to sub-hertz speeds.

A Hall-effect alternative might solve the problem. The Hall effect, demonstrated by Edwin Hall in 1879 in gold leaf, produces a few millivolts of voltage from one side to the other of a conducting ribbon with a magnetic field at right angles to it. The voltage is proportional to the current through the ribbon and the magnetic field and inversely proportional to the thickness of the ribbon and the charge carrier density.

The energy of whatever current is driven by the Hall-effect voltage, importantly, does not come from the applied magnetic field; it comes from the sensing current, which does not diminish that field. In theory, this means that you could use that voltage to drive another coil controlling another set of Hall-effect ribbons. In the absence of better means of amplification, you could drive the sensing current through many parallel layers of gold leaf with insulators between them from many isolated voltage sources, such as separate windings of a transformer, and put their Hall voltages in series, thus controlling an arbitrarily large amount of energy with an arbitrarily small magnetic field. This series Hall voltage could then drive a load whose resistance was not too high compared to the (ordinary, not Hall) resistance of the sensing ribbons. (Although, wait, aren't those sensing ribbons in series? Is that in fact a limit on the amplification you can achieve without superconducting ribbons?)

Such a device could, in theory, operate even at high frequencies. In practice, though, I think it might require an unreasonably large amount of apparatus for even a single logic gate.

Topics

- Electronics (p. 3430) (138 notes)
- Physical computation (p. 3631) (26 notes)
- Alternate history (p. 3316) (10 notes)

Induction kiln

Kragen Javier Sitaker, 2019-06-02 (19 minutes)

I'm thinking that, with modern electronics, the most expedient way to heat up a small pottery kiln to a controllable temperature might be induction heating of iron inside the kiln, using coils outside the kiln driven by a simple 300W ultrasonic inverter; this avoids the need for any exotic materials or components.

The basic design

A small cylindrical space is the kiln proper, as in An electric furnace the size of a sake cup (p. 666), perhaps 150 mm across and 100 mm tall; it is enclosed by insulating refractory material. This is divided into a bottom plate with a raised rim and a sort of inverted bucket that fits on top of it; lining the upper part of the bucket is a steel or cast-iron pipe (## in the cutaway diagram below) heated by coils of copper wire (oo in the diagram) wound around the *outside* of the refractory.

```
 ----------------------------------------
 |                                      |
 |     ____________________________     |
 |_  ##                            ##  _|
oo|  ##                            ##  |oo
oo|  ##                            ##  |oo
oo|  ##                            ##  |oo
 |   ##                            ##   |
 |    |                            |    |
 |    |                            |    |
 |    |                            |    |
 |_   |                            |   _|
   \  |                            |  /
    \_|                            |_/


 --                                    --
 | \                                  / |
 |  \________________________________/  |
 |                                      |
 |______________________________________|
```

With this arrangement, when the contents overheat and melt into the bottom of the kiln, they destroy only the bottom plate, which is the easiest part to replace; and, of the electrical components, only the inexpensive steel pipe is exposed to the high temperatures inside the kiln. The copper windings, in particular, are insulated from the pipe by a significant thickness of refractory material.

The raised border around the edge of the plate also serves to lengthen and crookedify the air path between the inside and the outside of the kiln, thus reducing thermal leakage, though probably at the cost of increasing wear when you open and close it.

The power delivered to the pipe can be on the order of 100 to 500 watts, and there isn't any particular advantage (and maybe a bit of

disadvantage) to its tendency to deposit primarily in the outer diameter of the pipe, due to the skin effect, so we can use frequencies as low as is convenient.

(As explained below in "materials and thermodynamics", the walls and floor probably need to be proportionally thicker than is shown here.)

Temperature feedback

We can use the type-K thermocouples used for safety cutoffs for gas ovens and pilot lights; these are available at any hardware store and cost about US$3. To avoid creating a low-thermal-resistance path through the kiln wall, the lead can be passed through the wall in a spiral path from the inside to the outside. Accurately measuring the 50-millivolt DC signal from the thermocouple is somewhat challenging, as explored in Inductor thermocouple sensing (p. 2037), but also a well-understood electronics problem with off-the-shelf designs to solve it. We might have to turn the induction heater off when we're taking temperature measurements.

Materials and thermodynamics

As described in An electric furnace the size of a sake cup (p. 666), refractory firebrick has thermal conductance in the range 0.2–0.5 W/m/K. We can use ghetto firebrick made by mixing used yerba mate with a ball-clay-based clay body; as noted in file `ceramics-notes`, my previous experiments firing it at 1020° burned out all the yerba cleanly (making a terrible stench) and left solid ceramic results at ratios of 1:1, 3:2, 1:2, 1:3, 1:4, 1:6, and 1:8 in favor of the yerba; but the 1:6 and 1:8 results were noticeably friable and porous, so 1:2 (probably about 0.8 g/cc) or 1:3 (probably about 0.6 g/cc) is probably a safe bet. These probably conduct about 0.3 W/m/K.

The surface area of the inner cylinder described above is 100 mm · 2 π 75 mm + 2 π (75 mm)² = 0.082 m², so it will lose 100 W at ΔT = 1000 K if the 0.3 W/m/K refractory averages 246 mm thick; this suggests that 300 W and 82 mm thick is a reasonable goal. You probably want to be able to burst above 300 W in order to hit temperatures at or above 1020 K in a reasonable time.

This gives an outer diameter of 314 mm and an outer height of 264 mm, for a total outer volume of 20.4 liters, 1.8 of which are taken up by the inner chamber; this leaves 18.6 liters of refractory material, about 11.2 kg of fired clay (and 13 kg of used yerba or coffee grounds or sawdust or whatever) if we assume 0.6 g/cc.

It's probably worthwhile to put 5 mm or so of solid clay on the top of the plate to make it resist impacts and abrasion a little better.

Although you would clearly get better results if you fired the kiln in a larger kiln before using it, I think it will probably be usable even if you don't do this — the steel pipe will reach a high enough temperature to burn out the yerba in its near vicinity, which will provide enough insulation to allow it to start heating up the inner chamber and burn out the yerba there. The ball clay I was using really needs to be sintered at 1020° to reach full strength, but full strength may not be necessary for this application. The yerba should eventually burn to charcoal at even 250° or 300°, which is probably adequate insulation.

It would be nice to prevent the yerba from stinking before the

thing is dry, and maybe a small amount of copper sulfate or something would work as a biocide for this. This depends in part on how tolerant the neighbors are, though.

Nickel-plating the steel pipe might extend its life — steel doesn't melt until at least 1130° and usually 1300° or more, and pure iron won't melt until 1492° (and the mild steel we can most easily get will be close to that), but iron or steel in direct contact with air oxidizes rapidly above 200° or so. Nickel-plating isn't totally trivial (nickel salts are fairly toxic) but can be done successfully with nothing more than salt, vinegar, and electricity.

Induction heating

Rudnev et al.'s *Handbook of Induction Heating, 2nd Ed.* explains that it's possible to heat pretty much any metal (and also graphite) by induction heating, but there are significant differences; in particular, ferromagnetic metals have a much thinner skin depth δ in which the heat is deposited than non-ferromagnetic metals. For good heat transfer, the metal must be thick enough to be "electromagnetically thick" (p. 65), at least six times the skin depth δ.

This is substantially complicated when you are heating steel, because its skin depth increases by about a factor of 15 when it passes its Curie point (p. 64), which is around 720–770°, and also because the skin depth in ferromagnetic materials depends on the the magnetic permeability of the material, which is not constant but varies with the magnetic field intensity — perhaps $300\mu_0$ at room temperature and low fields, it may drop to $150\mu_0$ at room temperature and 20% of the saturation field intensity (Fig. 3.10, p. 60).

Now, in theory, we might not have to worry about this, because we don't really care where the heat gets deposited inside the kiln; it will eventually conduct to where it's needed. But at 60 Hz the skin depth in SAE 1040 steel is already 2.5 mm at low field intensity (Table 3.5 on p. 64). Six times that (to be "electromagnetically thick" and absorb nearly all the field) would be 15 mm, and 15 times *that* (when we pass 700°) would be 225 mm. Since I think it's going to be hard to find or work with steel pipes that are more than 5 mm thick, it's probably worthwhile to use a higher frequency; if we want a skin depth of a 90th of 5 mm, we need 100 kHz (0.06 mm in SAE 1040 at 10 A/mm and 21°, according to the same table).

(However, the penetration depth scales as the square root of the frequency, and the power delivered scales as a negative exponential of the number of skin depths, so even two or three skin depths would already be pretty good, so lower frequencies might be adequate. Probably not powerline 50 Hz, though.)

On p. 97 et seq. we have expressions for the electrical efficiency η_{el} and related quantities. It points out that you have losses from the resistance of the coil itself and losses from heating up nearby random metal objects, and it gives this approximate formula for heating a solid cylinder in an electromagnetically long solenoid coil made from electromagnetically thick copper tubing:

$$\eta_{el} = 1/(1 + (D'_1/D'_2)\sqrt{(\rho_1/(\mu_r\rho_2))})$$

Here D'_1 and D'_2 are the inside diameter of the coil and the outside diameter of the cylinder, offset by their respective skin thicknesses, and ρ_1 and ρ_2 are their respective resistivities.

Now, the pipe isn't a solid cylinder, and the coil here isn't electromagnetically long or electromagnetically thick, but let's assume that the situation is close enough. We have the major disadvantage of having 82 mm of refractory between the coil and the pipe, so $D'_1/D'_2 \approx 2.1$, which would give us $\eta_{el} \approx 0.3$ absent other considerations; but our resistivity ratio ρ_1/ρ_2 is about $0.017\mu\Omega m/0.16\mu\Omega m \approx 0.1$ (Table 3.1, p. 54), and, at least at first, the μ_r of the steel pipe ≈ 300, so our $\sqrt{(\rho_1/(\mu_r\rho_2))} \approx \sqrt{(0.1/300)} = \sqrt{.00033} = 0.018$, so $\eta_{el} \approx 1/(1 + 2.1 \cdot 0.018) \approx 96\%$, though it might decline to 83% as the steel heats up and stops being ferromagnetic. On the other hand, the hotter steel will be even more resistive, so the situation might improve.

An interesting thing about this equation is that it doesn't include any explicit dependence on the number of turns or the current, except indirectly through the lower μ_r that comes with a higher field. I don't yet understand what the implications of more or less turns are, except that more turns means more inductance and thus more impedance — easier to drive at low frequencies, harder to drive at high frequencies.

Electronics

Suppose we decide on 20 kHz and 50 V as being relatively friendly numbers. To get 300 W out of 50 V we need 6 A, and thus about 8 Ω of impedance; with a 50% duty cycle we'd need 4 Ω. A MOSFET like the IRF540N (see My attempt to learn about jellybean electronic components (p. 1974)) should work well for this if we have a source for the 50 V, but it can handle 33 amps; maybe a more interesting approach is to drive the low-impedance induction coil through a step-up transformer, perhaps stepping up voltage 4:1 from 12 V to 48 V and stepping current down from 24 A to 6 A. Perhaps even a higher induction coil voltage could work.

Okay, but what does that imply about our coil? Presumably we want its inductance to be high enough that, if the pipe weren't conductive (maybe we replace it with laminated steel that blocks the eddy currents), it has a much higher impedance than 4 Ω at 20 kHz, maybe 40 Ω, and we want its dc *resistance* to be much lower than 4 Ω, maybe 0.2 Ω. AWG20 wire can reasonably carry 5 A and is 0.812 mm in diameter and 33 mΩ/m, so you could reasonably use 6 m of it to get 0.2 Ω, which would be just about 6 turns around the outside of the kiln; this would make it a sort of 6× stepdown transformer to the steel pipe, so you'd see 36× the resistance of the (skin depth of the) steel at the induction-coil terminals.

I'm going to hazard a guess, though, that even with 6000 mA going through it at 20 kHz, six turns of AWG20 wire isn't going to be able to induce detectable heating in a 5-mm-thick steel pipe, especially once it passes its Curie temperature. Let's suppose we're starting up and so our skin depth in the steel is 0.14 mm, and take that as a reasonable approximation of how much steel is actually on our one-turn transformer secondary dissipating the eddy currents (to avoid having to do the integral). So we have a strip of steel 140 μm thick, maybe 50 mm wide (half the height of the inner chamber), and 470 mm in circumference, with steel's resistivity of 0.16 μΩm. This comes out to 10.7 milliohms, so with only a 36× boost from the 6:1 turns ratio, we're still at only 0.39 Ω. So if we want to use only 50 volts ac on the inductor to heat up the steel, we are going to need

more turns.

Not as many more turns as you'd think, though — only about 19 turns, 18.7 m of wire, for which we need less than 20 mΩ/m to hit 0.2 Ω, thus AWG17 or bigger. But that's for heating up the cold steel. Once it warms up and the skin depth increases 15×, we'll need more turns to hit the same resistance — though at 700° carbon steel's resistivity has climbed from 0.16 μΩm to about 1.1 μΩm, so only 2½× more turns. These extra turns would be detrimental at lower temperatures if we're using a constant-voltage power supply, since the higher impedance means lower current and thus lower power, but maybe it's best to optimize for low power at low temperatures, where heat leakage is low, and high power at high temperature, so 40 or 60 turns might be worthwhile. 60 turns is 59 m of wire, so to keep the resistance below 0.2 Ω, we need 3.4 mΩ/m, so AWG10 (2.6-mm diameter) or less.

(Alternatively you could complicate things and use two separate coils or a variable frequency.)

So at startup we have 10.7 mΩ in the skin layer of the steel, which looks like a 39 Ω resistance in the 60 turns of the winding at ac, although their dc resistance is 0.2 Ω. We're putting 50 volts across it, although it's really only 25 volts rms (a square wave has the same p-t-p and rms), so we drive 640 mA rms through the winding, producing 16 W, 99.5% of which gets dissipated in the steel inside the kiln. So maybe 60 turns is going a bit overboard with favoring the higher temperatures, or maybe I calculated something wrong; the power is low by a factor of 20.

Let's try 30 turns: now we see 9.6 Ω in the winding at ac, so we drive 2.6 A rms through the winding — 65 W, which is probably reasonable. If the skin depth increases so that we're using all 5 mm of thickness of the steel, its resistance would drop to 0.3 mΩ, except that the compensating resistivity increase gets us to 2 mΩ or so, which looks like 1.8 Ω in the winding at ac, thus 28 A rms (!!) and 1400 W (!!!) of which 10% is dissipated in the winding itself if it's still 0.2 Ω (!!!!).

(Hmm, I think I'm still calculating something wrong, because the eddy-current EMF should be 30·50 V = 1500 V both when it's cold and when it's hot, so V^2/R should only increase by a factor of 5.3, not 22.)

So we probably need to take some kind of measure against that. Just using thinner steel might help some, but increasing the frequency is another possibility. Also, you could adjust the duty cycle.

The leakage inductance of the coil presumably also imposes significant high-frequency reactance, which will limit the current you can push into it.

(Because of skin effect, it might be necessary to use thinner wire, perhaps two or more parallel windings.)

(15× may be understating it — this skin depth is atypically shallow because we're using an atypically weak field and thus have atypically high permeability, so we will have an atypically extreme increase in skin depth when we smash into Curie.)

Scaling laws

What scaling laws govern the dimensions of such a kiln?

Well, at a fixed inner volume, the necessary wall thickness is

inversely proportional to the power. So if we could increase the power from 300 W to 600 W, we could use 41 mm of refractory instead of 82 mm of refractory, reducing the weight of the thing by more than half and its volume by a third. Voltages and currents would increase. Lower powers would require using proportionally thicker walls (and, perhaps, proportionally smaller inner chambers) which would eventually make the pipe too short to pick up much current.

The inner volume is proportional to the cube of the characteristic dimension, and the leakage surface area to its square. So, for example, a 10% increase in width, height, and depth would increase leakage surface by 21% and volume by 33%. Maintaining the same power would thus require increasing the wall thickness by that same 21%, which very quickly gets us to absurd dimensions. It might be more reasonable to increase the wall thickness proportionally (by 10%) and the power proportionally (by 10%) as well.

Electrical efficiency η_{el} stays the same as long as the wall thickness and inner diameter increase proportionally. Using the proportionally thicker metal we could accommodate more easily in a larger chamber would permit proportionally lower frequencies due to proportionally thicker skin depths.

So a small prototype, like the sake-cup-sized thing suggested in An electric furnace the size of a sake cup (p. 666), might be a good thing to start with.

Topics

- Electronics (p. 3430) (138 notes)
- Materials (p. 3560) (112 notes)
- Manufacturing (p. 3558) (50 notes)
- Ceramic (p. 3371) (17 notes)
- Kilns (p. 3538) (8 notes)
- Induction (p. 3523) (3 notes)

Flying spot reilluminatable stage

Kragen Javier Sitaker, 2017-05-15 (1 minute)

One of the key cost drivers in television filming today is the single-camera technique, in which each scene is filmed several times with a single camera in different camera angles, in order to get the lighting optimal.

You can use a flying-spot camera to do much of this in a single shot using multiple photosensors, each one corresponding to a different virtual light source, all filmed from the point of view of the flying-spot illumination source. This allows you to make a weighted sum of the signals from the various light sensors after the fact in order to compute a version of the scene with the proper "lighting".

This doesn't allow you to change camera angles (without multiple flying-spot sources and the attendant difficulties with higher-speed signals), but in many cases all that is needed is to crop a higher-resolution shot down to the area of interest.

The standard approach since the 1950s to making flying-spot stages livable for the actors is to illuminate them during the vertical blanking interval with a strobe light.

Topics

- Optics (p. 3609) (34 notes)
- Cameras (p. 3364) (8 notes)
- Video (p. 3768) (7 notes)

Barcode receipts

Kragen Javier Sitaker, 2007 to 2009 (6 minutes)

So it's kind of a pain to track my supermarket purchases, in part because the receipt isn't really machine-readable. The store often has all the data in machine-readable form in their cash register; could they give it to me in machine-readable form?

It's Technically Feasible, Inexpensively

The bare minimum information for the receipt would be a little bit of header information (store location, date and time, currency of transaction), and for each item on the receipt, a UPC code, a quantity, and a price. UPC codes seem to be 13 digits (abbreviatable to 8), quantities are usually either three digits (of weight) or one (of units), and prices are usually two to four digits, including price. You could prefix prices and quantities with a length digit or terminated them with a non-digit code (say, if you're using BCD). (Say you bought 2.30 kilograms of avocados for $12.40; the quantity-and-price field would then read 323041240, and the UPC number would have to indicate that the unit of measure was 10 grams). In this scheme, quantity 1 would be represented as "11"; you could special-case that as "0".

So the typical item would have five digits of price and quantity data, for a total of 18 self-delimiting decimal digits. In BCD that would be 72 bits, or 9 bytes. The per-receipt header might be 30 bytes. So a 30-item receipt might be 300 bytes. (In binary instead of BCD, you would need about 60 bits per item, but that's probably not worth the extra complexity.)

PDF-417 stacked barcodes hold up to 1108 bytes of binary data per symbol; DataMatrix/Semacode holds up to 1556 bytes per symbol; QR Code holds up to 2953 bytes per symbol (at 177x177 pixels, which is about one-third redundancy). So 300 bytes is small enough that you could print a small barcode directly on the receipt, probably even with old dot-matrix receipt printers, given appropriate driver software.

Minimally, with 1-bit pixels, you'd need 2400 pixels to represent 300 bytes, which is about 50x50 pixels. In traditional 5x9 dot-matrix fonts, that's less than 54 characters. I don't know if existing barcodes are sufficiently robust against the kinds of errors dot-matrix printers add (round dots, row-to-row misregistration) but there's plenty of headroom here for error correction. I vaguely seem to remember one of the matrix barcode systems recommending printing each pixel of the barcode symbol with at least 4x4 of the underlying pixels, so with that and the redundancy, you might need 16 * 4/3 * 300 * 8 = 51200 dot-matrix dots, 226 pixels square, or 1137 5x9 character cells. By my count, my latest dot-matrix receipt is 34 characters wide, so that would be 33 lines. On the face of it that sounds like an impracticably large amount to add to a 30-item receipt, but I've seen much worse, so it might already be reasonable. But you could presumably design a "barcode" whose overhead was close to a factor of 1.33 instead of a factor of 21, and then you'd be down to 72 characters instead of 1137, which obviously adds only a trivial amount of cost to the receipt.

So you could use a less obtuse data format, too, instead of the horrible all-decimal format suggested above, where "323041240" means "2.30kg $12.40".

How Could It Be Bootstrapped?

It may seem a little impractical to expect supermarkets and the like to upgrade their cash-register systems for the convenience of customers who want to itemize all their grocery purchases, which is something hardly any of them do. But here's a slightly plausible deployment path.

A few people, some of the time, have to itemize all their purchases and turn in receipts: businessmen, academics, and NGO workers on travel on expense accounts, mostly. They already do this even though it's a pain, and a lot of them do it when they're on travel without their secretaries. A lot of them would be delighted to avoid the hassle.

So if a major retailer of some expense-account-able commodity announced this kind of barcoded-receipt program and shipped free software to import your receipts into a few of the most popular ways of tracking expense accounts, it would attract these travelers. Maybe Ruth's Chris, or Avis, or Hyatt, or toward the lower end of the market, maybe T.G.I.Friday's, Chili's, Days Inn, and the like.

It would have to offer some substantial advantage over monthly credit-card bills to get adopted, since that's what a lot of these folks use now. I have a couple of ideas: - It costs the receipt issuer much less to print a barcoded receipt than to process a credit-card transaction, so small-transaction merchants who aren't willing to accept credit cards could therefore become expense-account options. - Issuing a company credit card to someone puts the company at some financial risk, and perhaps for this reason, academics on travel and workers for small NGOs rarely use company credit cards. Accepting barcoded receipt entries does not entail any extra risk to the company. - Credit-card bills lose any convenience advantage when only part of a purchase is expensable.

Similarly, self-employed USians who deduct business expenses on their federal tax returns are obligated to itemize those business expenses: travel expenses, raw materials, equipment, and the like. This suggests a broader group of retailers: Home Depot, OfficeMax, CompUSA, Best Buy, Kinko's, Ace Hardware, FedEx, Ikea, the Gap, Borders.

Once the receipt-importing software was out there, other companies could compete for these customers by barcoding their receipts too, if the software to do so is relatively easily available.

The path from Hyatt to Carrefour is pretty dubious, though. So is there anyone already in a position of routinely itemizing their supermarket purchases? Maybe servants who go grocery shopping for their masters?

Thanks

To Beatrice, for very helpful discussions on the subject.

Topics

Caching screen contents

Kragen Javier Sitaker, 2017-06-14 (2 minutes)

Suppose I have a 1920×1080 screen that doesn't always update, and I'd like to cache tiles of it. I could do it with quadtrees (11 levels down to the pixel level, or 7 levels down to the traditional 16×16 tile size) or I could do it by simply dividing the screen into a flat grid of tiles — tiles of 64×64 pixels, say, dividing the screen into a 30×17 grid.

Then, to redraw the screen in a kind of RESTful system, I could make 510 requests to a tile-rendering service, providing it with information about what was on the screen. For many uses, it would be okay for these requests to take too long for the screen to fully update in a single frame time. You could imagine a usable system, for example, where each 16.7ms frame only has time to refetch 50 of the tiles: 340μs to merely revalidate the cache of a tile.

(As a point of comparison, httpdito on my laptop takes about 50 μs per HTTP request, which would be not quite enough to redraw the screen.)

This would suck for watching a movie or mouse tracking, though, since you'd have randomly 0–100 ms of latency (0–6 frames) for screen updates, making your mouse pointer jitter all over the fucking place, making movies unwatchable, and making real-time games utterly unplayable. But if you have some kind of cache invalidation protocol that eagerly recalculates the tiles that have actually changed, you could reliably keep your latency to a fraction of a frame and track the mouse accurately.

Suppose we can improve somewhat on httpdito by using a binary protocol and not forking and accepting new connections and get down to 10 μs per request. (0MQ can normally handle messages in under 1 μs.)

Topics

Bitsliced operations with a hypercube of shuffle operations

Kragen Javier Sitaker, 2016-11-30 (2 minutes)

Doing bit-parallel operations with bit-parallel instructions on big bitvectors is a useful way to do computations with very high efficiency when you have enough independent computations. Bringing together bits from different bitplanes requires some kind of bit-shuffling or transposition operations, but the question remains which of the involute number of possible permutations should be provided. For example, if each of the 64 bits of a 64-bit output word can be a copy of any of the 64 bits of the input word, there are 64^{64} possible operations; specifying one of them would require 384 bits.

In essence this is a new face of the "topomania" problem faced in parallel computer design in the 1980s and early 1990s, with bit positions in all the register taking the role of processors. The typical early solution was a hypercube topology, in which, say, a 1024-processor CM-1 would have 10 communication links per processor.

This seems like a perfectly adequate solution to me. Given a 256-bit register, 16 bit-shuffling operations suffice to construct a useful hypercube:

```
rot(0) exchanges even and odd bits, taking ...ABCDEFGH to ...BADCFEHG
rot(1) exchanges even and odd bitpairs: ...ABCDEFGH → ...CDABGHEF
rot(2) exchanges even and odd nibbles: ...ABCDEFGH → ...EFGHABCD
rot(3) exchanges even and odd bytes;
rot(4) exchanges even and odd wydes;
rot(5) exchanges even and odd 32-bit words;
rot(6) exchanges even and odd 64-bit words;
rot(7) exchanges the two 128-bit halves of the register.
```

To these rot(n) operations we add copy(n) operations which discard half of the bits (say, the high half) instead of exchanging them:

```
copy(0): ...ABCDEFGH → ...BBDDFFHH
copy(1): ...ABCDEFGH → ...CDCDGHGH
copy(2): ...ABCDEFGH → ...EFGHEFGH
```

etc.

It's not very important, I think, which direction the copy operations copy; copy(n) ∘ rot(n) will copy in the other direction instead.

I think this provides adequate routing facilities for many aggregate computations. Together with an analogous set of instructions for filling registers with index bit patterns (…10101010, …11001100, …11110000, …) it should enable fairly general and efficient bit-sliced programming.

Topics

Gardening machines

Kragen Javier Sitaker, 2019-04-02 (updated 2019-04-24) (32 minutes)

Exponential rosemary accounting

I have a couple of small potted rosemary bushes on my balcony, with a third that I'm trying to coax to grow out some roots in a cup of water inside. All three of these grew from a cutting Beatrice's husband Santiago gave me when he and Beatrice moved to Holland late last year; I have perhaps quintupled his investment in me in five months, if we measure in rosemary biomass. If I can continue this rate of exponential growth for the rest of the year, by way of judicious propagation, in 7 more months I will have multiplied my current rosemary holdings by a factor of 10, a factor of 50 for the whole year, a 4900% internal rate of return on this project, a doubling time of about 66 days.

Of course, we don't usually account for agriculture this way. And I probably won't actually have 20 rosemary bushes in August, nor four rosemary bushes each five times the size of the ones I have. The exponential growth of the plants will eventually outstrip my balcony space, pots, and watering time; if I don't attend to them, they may die of dehydration, but also, they face other dangers which themselves grow exponentially: they may be choked out by faster-growing weeds, or they may suffer attacks by insects, fungus, or root nematodes. In any case, the last five months have been summer months, with temperatures around 25°, while the next few months will be winter months, with temperatures around 15°, so the Arrhenius law predicts that the plants will grow about half as fast. Ultimately, the limiting factor for my rosemary production will probably not be the capital equipment of rosemary plants, nor my labor or mineral resources, but the energy I have available: rosemary will not grow in the dark.

I also have an Aloe vera plant I found discarded in a gutter, which has rooted nicely after a few weeks of early failure, and another aloe that was a gift from my girlfriend Alejandra. In a third pot, I'm trying to root a leaf I accidentally knocked off the gutter plant. The aloe plants, too, are growing exponentially, though apparently at a slower rate.

Although these growth processes are exponential, they are also slow. Even on the rosemary plants, the difference from one day to the next is very hard to see; calculation suggests that it averages about 1%. Several days can go by without a noticeable change.

(Post scriptum: the above was written 2019-03-26. The cutting finally grew out roots (one root of 15mm and two roots of 5mm) on 2019-04-02, one week later. I'm not sure how long the whole rooting process took, since I'm not sure when I made the cutting, but it must have been a couple of weeks or so. As of 2019-04-05, the roots had increased to seven in number, including a tiny fork from one of the oldest ones, which itself had lengthened to some 40mm; this quantity of daily root growth is also on the order of 1% of the volume of the cutting per day, which is some 150 mm long. On 2019-04-07, the root tips had increased to, I think, 20 in number; on

2019-04-08 a root had sprouted from a node higher up the stem as well, though now I count only 15 tips.)

Ruderals

Rosemary and aloe are perennial plants adapted to somewhat harsh conditions: their ecological niche does not demand that they grow especially fast or reproduce abundantly, but rather that they successfully survive a variety of temporary adversities. Ruderal annual plants, on the other hand, survive by growing rapidly in soils that have been rendered temporarily plantless by plowing or some other catastrophe, ending their lives in a riotous explosion of sex called "going to seed"; the champion among cultivated ruderal annuals is corn ("maize"), which can grow several millimeters per day. Each corn plant — a corn kernel's way of making more corn kernels — normally produces several ears of corn, each bearing 200 to 400 kernels, in a single growing season, which normally only occurs once per year. But this means that a single kernel of corn can multiply into several thousand kernels within a year, not just the factor of 50 or so I'm estimating for the rosemary.

This means that corn's rate of capital growth is on the order of 100,000% per year.

Yogurt bacteria

I'm eating a small jar of yogurt I made the night before last. In the refrigerator right now I have another liter or so of it. The yogurt, like poop, consists mostly of dead bacteria, in this case mostly Streptococcus salivarius subspecies thermophilus and Lactobacillus delbrueckii subspecies bulgaricus, mixed with chemicals the bacteria couldn't digest, mainly a casein gel in this case (though the bacteria do digest some of the protein), and bacterial waste chemicals such as lactic acid and acetaldehyde. The jar contained perhaps 100 mℓ of yogurt, which was made from about 200 mℓ of milk through about 24 hours of fermentation after being seeded with about 1 mℓ of living yogurt culture. The bacteria are, say, about three cubic microns each, and occupy over 50% of the yogurt volume, so I just ate tens of trillions of bacteria which I produced overnight from only hundreds of billions of bacteria.

Probably, under better-controlled conditions, I could have gotten the yogurt to finish in under 8 hours; most of the jars were already relatively thick after 12 hours, though some had not fermented at all — perhaps I had inoculated them with the starter culture when they were still too hot, so I added more starter culture to them at this point. The 24-hour timescale suggests an average rate of bacterial reproduction of, conservatively, about 20% per hour, which amounts to doubling every three or four hours.

(More aggressive bacterial cultures like Escherichia coli can reproduce much faster, doubling every half-hour or so under ideal conditions, around 37° with all the nutrients it needs.)

Doubling every four hours would give you 2191.4 doublings per year, which works out to a multiplication by about 4.8×10^{651}. But only about 1×10^{98} of these yogurt bacteria would fit into the observable universe. So if these yogurt bacteria were to double 325 times, which would take less than two months at four hours per doubling, they would fill the observable universe, out to the farthest

galaxies and quasars. After about the first month, the yogurt sphere would have to be expanding outward faster than the speed of light, which is probably impossible. Even if you could find a way to defeat relativity, where would you get the milk?

Flour beetles

Some time ago I was the proud owner of a number of confused flour beetles. Confused flour beetles are marvelous little self-reproducing automata, about three millimeters long, which can acquire sufficient water by metabolizing the dry flour they eat to survive and reproduce, although if the flour is very dry, they reproduce more slowly. They can also survive a thousand grays of radioactivity (in about 10% of cases) and thus probably more than a thousand sieverts, more than cockroaches can. Five grays will usually kill a person; radiation therapy usually kills tumors with 20–80 grays. Confused flour beetles are commonly raised for use as laboratory animals, particularly in studies of genetics.

I didn't measure the rate of growth of my capital holdings of confused flour beetles, but Wikipedia's marvelous "Home Stored Product Entomology" article tells me that a female flour beetle can lay 300–400 eggs over a 5–8 month lifetime. So a mating pair of beetles can produce, conservatively, 150 mating pairs of beetles in 8 months, which amounts to a doubling time of 34 days, once exponential growth kicks in after an initial linear ramp-up period, and a multiplication factor of some 1800 per year, a return on investment of 180,000% per annum — nothing like the universe-consuming rate of growth of yogurt, but comparable to that of corn.

Rather than submitting my fortune to appraisal by counting my flour beetles, I took them, the grain they were eating, and the basket they were eating it in, and threw them all together into a dumpster, in hopes that they would cease their self-reproducing in my house. Thus I preserved my holdings of flour, beans, and other grains. This is because I could not program the beetles; they came preprogrammed from an optimization process which did not, for instance, offer me a quiescent mode in which a stock of flour beetles I already judged as more than sufficient would cease to increase, nor a practical way to employ the beetles to construct mosaics, murals, or sculptures directed by an STL file.

So far this measure has apparently been successful, though I live in fear of the beetles' return. Every new bag of flour I get spends some time in my freezer to kill any insects inside.

Mere capitalism

Capitalism, of course, is based on the exponential growth of productively employed capital. Wikipedia clarifies the terminology thus:

> In economics, "capital", "capital goods", or "real capital" refers to already-produced durable goods used in production of goods and services. The capital goods are not significantly consumed, though they may depreciate in the production process. "Capital" is distinct from "land" in that capital must itself be produced by human labor before it can be a factor of production. At any moment in time, total physical capital may be referred to as the "capital stock", a usage different from the same term applied to a business entity. In a fundamental sense, capital consists of any produced thing that can enhance a person's power to perform economically useful work — a stone or an arrow is capital for a caveman who can

use it as a hunting instrument, and roads are capital for inhabitants of a city. Capital is an input in the production function. Homes and personal autos are not capital but are instead durable goods because they are not used in a production effort.

The name "capital" refers originally to the heads of herds of livestock, which feature this same exponential growth through biological processes, except when limited by available feed. But, compared to an internal rate of return of 4900% or $4.8\% \times 10^{653}$, the traditional rates of return on capital investments, about 3%–10% per annum, seem fairly pitiful. The outrageous rates of exponential capital growth we see in gardening or herding invariably slam quickly into other resource limits: the sunshine on my balcony, for example, or the milk I have on hand to grow yogurt in, or the difficulty in controlling exponential-growth processes to produce the desired outcomes.

One of the rosemary plants, for example, has a blade of ruderal grass coming up in its pot. Grass is a relative of corn (more properly, corn is a sort of ruderal grass) and it grows much faster than the rosemary does. If I don't want a pot full of grass shading the rosemary, sooner or later I need to pluck out the weed. And this same rosemary plant had a close call with death when I was still rooting the cutting it in water: fungus started to grow on its bark, and if the process had continued, might have started to digest the living plant. I lowered the water level in its jar so that its bark could dry out, and the fungus disappeared and hasn't returned. This latest batch of yogurt all turned out without problems, but the previous batch had a couple of jars that grew some fungus after the bacteria, causing an off flavor.

Historically, the main determinant of power in agricultural and pastoral societies — those whose most important means of production grows by these insane exponential biological growth rates, rather than being human-designed machinery — has been control over land, particularly arable land. Landowners had power, and nobles and gentry were distinguished by their land holdings, not their wisdom, learning, machinery, or industry, nor even martial virtues like their honor and bravery, and certainly not by the amount of grain stored in their granaries. This was the rule in Europe from the time of the late Roman Republic, when Senators derived their income from the latifundia they owned, and of course in every feudal society too.

Consider, also, for example, Imperial China's 士 "shi"; until the Warring States Period they were noble-born charioteers. The 秦 "Qin" dynasty and its successor 汉 "Han", 隋 "Sui", 唐 "Tang", and 宋 "Song" dynasties needed to reduce the power held by the feudal nobility, for which purpose they established an empire-wide civil-service bureaucracy, redefined the 士 "shi" class to include it, and staffed it increasingly with commoner scholar-bureaucrats they hired according to their performance on 科举 standardized tests. But these scholar-bureaucrats also became agrarian landlords, and were generally landed gentry even before sitting for the examination.

Thus, even though Ricardo and Smith's classical economics was born at the moment and center of England's transition from such a feudal society to the new capitalist society, they do not consider plants or plant seeds to be capital, precisely because they multiply too rapidly for their scarcity to be a major limiting reagent in the production function in an economy with functioning agriculture. Only in

extreme cases, like when the Lykov family's millet crop was killed by a late frost, leaving them hungry until seven forgotten seeds sprouted the following year, or in Israel's destruction of generations-old olive trees in the Palestinian occupied territories, are we forced to confront the crucial nature of self-replicating automata to the economic production process. Under normal circumstances, they are either nonexistent or superabundant, and in either case their stock is not a relevant variable.

So, what will happen to human society when we develop programmable self-replicating robots that can work from inorganic feedstock? No longer will we measure the assets of a company by its stock of capital goods — indeed, apart from a few exceptions like railroads, it's been quite passé for a company's book value to be based on tangible assets like railroad cars since at least the 1980s, much less its market cap. Nobody would think to argue that Microsoft has a competitive advantage against software startup firms because Microsoft owns a lot of computers to produce software with, while the startups would have to buy them, though it seems reasonable to think that Ford, GM, and Chrysler had an edge over late-20th-century startup car firms because the Big Three already owned factories. (Tesla is the first successful automobile startup in the US in decades, although China's success at bringing new brands to market since 2005 suggests that US industrial policy has as much to do with the late-20th-century oligopoly in the US automotive market as raw economic factors did.)

But what happens when we can grow factories like rosemary plants, or even yogurt? How do we think about the effects on human society and future economic prospects?

Perhaps as we learn to garden programmable machines we will find ourselves thinking more in biological terms than in mechanical terms. We'll be constantly watching out for something or other going wrong, hoping to notice problems that could grow exponentially and stop them while they're small. We'll accept some degree of parasitism and predation on our production as the cost of having a system that works at all. We'll try to manipulate conditions to disfavor the growth of the elements we don't want, even at the cost of those we do — as I let the roots of my rosemary dry out a bit to get rid of the fungus on its bark. To a great extent, these sound like the skills of a turn-of-the-millennium business manager, drawing on the best of 20th-century management practice and Taoist practice.

Stability, size, durability, and repairs

Industrial-age machinery is prized in part by its stability, size, and durability; and it is repaired. But self-reproducing machinery need not work this way, and its logic will invert industrial-age logic in interesting ways.

Stability

A lathe which, left idle overnight, can no longer cut in the morning is hardly a lathe at all. Great efforts go into ensuring that form materials are dimensionally stable so that changes in humidity will not drive them out of tolerance. It is common for a machine left unused for decades to remain usable, perhaps with a few minor repairs.

But many bacteria will die if left without food for a few hours.

Suppose some strain can reproduce itself by a factor of a thousand in ten hours. A left-behind cell that lacked resources to reproduce itself during that time can only, at best, result in a single cell at the end of it. Even if it happens to fall back into the nutrient broth, its descendants will be outnumbered a thousandfold by those who multiplied the whole time. So, many such bacteria spend no resources on surviving lean times.

Similarly, a self-reproducing machine that requires continuous homeostatic control to remain viable, and "dies" irreversibly if that control fails for a significant length of time, may be a perfectly reasonable replicator. So stability is not necessary to self-replication if homeostasis can replace it.

Size

Machine size was the distinguishing feature of the mass-production era. A steam shovel that carried 100 tonnes per shovel-load would move rock 10 times as fast as a shovel that only carried 10 tonnes, and might require only 5 times as much material, and the same number of fabrication and assembly operations, though each handling pieces 5 times as big. Moreover, a 20-tonne rock could be moved by the 100-tonne shovel without being broken first. A bigger lathe could make bigger parts, and due to its greater rigidity and power, it could make the same parts faster and to tighter tolerances. So, bigger machines made more efficient use of material, tooling, and labor, and could do things smaller machines could not.

But replicators need not follow this logic. There is every reason to believe that a one-kilogram object can be built as easily by a thousand tiny replicators each processing a gram of material, followed by a final assembly, as by processing the kilogram in one operation with a large replicator. Indeed, it seems likely that the smaller replicators will work faster, working as they do in parallel. So the size pressure on replicators may indeed be opposite: toward the smallest practical dimension, to maximize flexibility.

Biological life certainly has followed this design. Despite the existence of large single-celled creatures like foraminifera and slime molds, and the occasional large cell like the squid's neuron with its giant axon, the typical size of a eukaryotic replicator — a cell — is close to a nanogram in complex organisms. You could argue that, for example, a human neuron is not a replicator because most of them die without ever producing a new neuron, and they are rather specialized to their climate-controlled environment; on the other hand, nearly all human neurons are produced by the replication through mitosis of a neuron, and they can be coaxed to grow and even replicate in vitro.

This composition from such a large number of quasi-independent replicators certainly has major disadvantages, including cancer, but it seems to have sufficient compensatory advantages that the cells of multicelled creatures are hardly ever big enough to see without a microscope. So we should expect artificial replicators to find significant advantages in smallness, as well.

Durability

Conventional machinery is considered better when it is durable. A lathe whose chuck or gears or ways wear out after a few hours of use is a very poor lathe. Often we go to great effort to improve the

durability of our machinery so that it can continue operating reliably for long periods of time, including measures such as jewel bearings in wristwatches, hardened steel gears, hardened steel calipers, all manner of bearings, and many others. One of the great benefits of vacuum tubes over relays in the history of computing was that they lasted for many more operations than relays did.

This industrial-age logic has come under assault by disposable products over the last few decades, with some degree of rationality: if money you invest has a 10% rate of return on investment, you can quite reasonably use a 9.1% discount rate to devalue the future use-value of your machinery. To be more concrete, suppose you spend US$1000 (of resources, energy, and labor) to build a machine that will last for 20 years. At the end of these 20 years, you will need to spend another US$1000 (inflation-adjusted, of course) to replace it. To be prepared for that eventuality, you can invest US$149 today into something productive, which will have grown to that US$1000 in 20 years. So if you had the option to extend the machine's life to 40 years by building it in a more expensive way that cost US$1200, it would be a poor economic choice. Indeed, by investing US$175 today, you can obtain enough investment income to replace the machine every 20 years for eternity; even increasing its lifetime to millennia for the extra US$200 cannot compete.

But replicators, as described above, have a rate of return many orders of magnitude greater than that of industrial-age capital. Their dynamics have a phase transition when they last long enough, on average, to produce more than one offspring: at 0.99 offspring before death, their population would gradually exponentially decrease, while at 1.01 offspring before death, it would gradually increase.

But suppose you have an autotrophic clanking replicator that can self-replicate in a week, and which lasts only three weeks before irreversibly failing. For the first week, you have one replicator; in the second week, you have it and the one it produced during the first week, each producing a child replicator. In the third week, you have four replicators, each producing a twin, but at the end of this week, your original replicator dies, so you have seven replicators in the fourth week; they produce seven new replicators, but the two oldest die, so you have 12; and so the process continues, with 20, 33, 54, and so on, with an exponential growth rate of $\varphi \approx 1.618$, the golden ratio, per week.

This means that, once you have reached an adequate population of replicators, you can use $\varphi - 1 \approx 61.8\%$ of the total to produce non-replicator products, while the remaining 38% are occupied with replacing the dying replicators. This 38% is the "overhead cost" of this limited durability; if some redesign would extend the replicator's lifespan to dozens or hundreds of offspring, it would increase the end-product productivity of the replicator "biomass" only by eliminating up to that 38%, working out to a 61.8% improvement.

The same logic applies regardless of the generation time: if your replicator needs an hour to self-replicate and dies after three hours, you still have a growth rate of 61.8% per generation. Three hours is a very poor lifetime for conventional industrial machinery; industrial products that are consumed this quickly, such as arc-welding electrodes and TIG-welding cups, are categorized as "consumables" rather than machinery.

Steel cutting as an example

As a specific example, suppose one of the cycles in our clanking replicator's fabrication process graph cuts steel.

The standard industrial process for this nowadays cuts the steel ceramic inserts made of tungsten carbide cemented with cobalt, pressed to shape in molds.

The previous standard industrial process for this, still in use in places, used a special "high-speed" steel alloy, sufficiently harder at high temperatures than ordinary steel to cut it rapidly and last a long time, cut to shape by abrasive grinding with a grinding wheel. The grinding wheel was made of, for example, crystals aluminum oxide or silicon carbide cemented with, for example, magnesium oxychloride, cast into the shape of a grinding wheel, and then trimmed to an accurately circular shape using star wheels, made of hardened steel. (Older grinding wheels might have been cut from stone.)

The process before that used cutting tools made from high-carbon steel, annealed and cut to shape using either steel tools or a grinding wheel, and then heat-treated to harden them.

The ancient process used cutting tools made from wrought iron, hot-forged, then cut to final shape using either hardened steel tools or a grinding wheel, and then case-hardened to harden them.

But abrasive cutting with, for example, a grinding wheel or abrasive sawblade, works fine for cutting steel. It's often used for cleaning up after welds, where it easily copes with oxide deposits, and it's often used as the most expedient alternative at construction sites. It's also used, in the form of "surface grinding", to cut surfaces more accurate than those that can be cut with steel tools, due to lower temperatures and deflection forces. The litany of other steel-cutting tools described above is purely a result of optimizing the cutting process; a high-speed steel or especially ceramic tool can cut much deeper into the workpiece with each cut than the microscopic spark removed by a grain of abrasive, and it also lasts much longer. Even the use of super-hard abrasives like silicon carbide and aluminum oxide is an optimization to cut faster and wear slower.

A clanking replicator might quite reasonably use only abrasive cutting or only case-hardening, thus dramatically shortening the time per generation by reducing the number of processes in the cycle; and, in the abrasive case, it might quite reasonably use only low-grade abrasives barely hard enough to cut the steel, such as quartz, soda-lime glass, feldspar (orthoclase feldspar being the defining mineral of Mohs hardness 6) or even apatite (Mohs hardness 5). Glass is especially promising here because of its isotropy, low-temperature thermoplasticity, and lack of need of time for crystallization.

This example also illustrates why small size could be a big advantage: the diffusion process of case-hardening, the crystallization processes of the crystalline abrasives, and the heating and cooling processes of a mass of glass all take time proportional to the characteristic dimension of the thing. So a replicator of one centimeter in diameter could reproduce its steel-cutting tools, all other things being equal, in one hundredth the time of a replicator of one meter in diameter.

Repairs

When my bicycle malfunctions, I debug it; I compare my mental

model of the ideal bicycle to the actual observed bicycle to discover where the reality departs from the ideal, meanwhile updating my conception of the ideal in accordance with the things I learn from the observed bicycle, and modifying the observed bicycle with wrenches to sharpen my observations. Once I have found the root cause of the problem — a spalled bearing ball, for example — I repair the bicycle, reconstructing the deviant parts in accordance with the ideal, perhaps an ideal I have improvised in the moment rather than my original conception. For example, when the derailleur bent irreparably, I removed it and shortened the chain, converting it into a fixed-gear bicycle for the time being.

In this way, my behavior provides the bicycle with the homeostatic processes it needs to continue working, which it is too simple to provide for itself.

When my rosemary malfunctions, I usually do not repair it; usually the malfunctioning part dies on its own, but sometimes I prune it. Self-replication takes care of replacing the damaged rosemary more easily than figuring out that, for example, some xylem channels are blocked in this branch due to excessive bark growth on the inside of a tight bend compressing the inner bark, and carrying out microsurgery to widen the xylem channels. The same kind of process usually handles malfunctions in my own body, and it will apply to autotrophic mechanical replicators as well: it will commonly be more practical to recycle broken replicators than to repair them.

However, this is not the case for "malfunctions" that happen so frequently that they would reduce machine fertility below the replacement rate; those must be repaired by either internal or external homeostatic processes, such as a dude with a wrench. In the initial stages of replicator bootstrapping, we can expect to have to repair many faults manually at first, until we have improved the replicators to intervention-free replacement-level fertility.

Also, malfunctions that affect the overall system of replicators, rather than an individual replicator, require a different approach. If the rosemary comes under attack from caterpillars, pruning it may not be adequate — the caterpillars will just eat the healthy part I didn't prune. Malfunctions that are themselves replicators, such as viruses, parasites, and cancer, may be able to outrun mere apoptosis-based approaches. Debugging such system problems promises to remain as challenging as in gardening, and presumably the future of humanity will be decided in this way, as different people or groups struggle for the power to program replicator populations.

Topics

• Ubicomp (p. 3761) (12 notes)
• Post-scarcity things (p. 3642) (6 notes)
• Gardening (p. 3469) (2 notes)
• China (p. 3375) (2 notes)
• Capitalism
• Abrasives

Rhythm codes

Kragen Javier Sitaker, 2015-09-03 (4 minutes)

Suppose your codeword is a common-time measure; the symbols available for use within it are a quarter-note, two eighth notes, and a rest; you're using two pitches simultaneously, separated by a perfect fifth; the first beat, for timing, cannot be two rests or two eighth notes, nor can the second beat be two rests; and if only one of the pitches appears in the measure, it must be the low one (since without hearing the other one, it's hard to guess whether it's meant to be the low one or the high one). How many possible codewords do you have?

Well, a normal beat (the third or fourth) has 9 possibilities available. The first beat can be 7 of those 9 possibilities, and the second can be 8 of them. That gives us $7 \cdot 8 \cdot 9 \cdot 9 = 4536$ possibilities, but of these, a few have only rests at the lower pitch: $2 \cdot 2 \cdot 3 \cdot 3 = 36$. So the total number of these very simple biphonic melodies is 4500, so each one can encode just over 12 bits, which is roughly a single English word, depending on how sophisticated a model you use to measure the entropy.

I chose a perfect fifth because, although the two pitches are assonant, their harmonics will be largely distinct, potentially enabling decoding even if large swaths of the spectrum are lost. The fundamentals of the two of them are the second and third harmonic of a fundamental an octave below the lower pitch, and so the divisible-by-three harmonics of the low note will be the even harmonics of the high note. This means that half of the harmonics of the high note will not be found in the low note's harmonics, and two thirds of the low note's harmonics will not be found in the harmonics of the high note.

If, for example, we choose 220Hz (A, I don't know, is that A below middle C?) for the low note, then 330Hz (E?) will be the high note. The 220Hz note has about 68 harmonics in the easily-human-audible range, of which 46 are not among the harmonics of E; and E has 46 harmonics in the easily-human-audible range, of which 23 are not among the harmonics of A. You should be able to distinguish between these notes by finding the 9.09ms-lag autocorrelation peak and comparing it to the 4.55ms-lag one and the 3.03ms-lag one (they should be higher if there is a note there and lower if there isn't?), and that should work even if big parts of the spectrum (including the fundamentals) are strongly suppressed. And you should be able to find the note onset by differentiating these autocorrelations over time.

If you play the melody at 180bpm, which is pretty durn fast for music, you will get through one measure every 1333ms, for a total of 9 bits per second. This is slower than people talk, so people might be able to learn it. Your eighth notes will run for 167ms each, which is about 37 cycles of A's fundamental, which is plenty.

A 30ms window is 1323 samples at 44.1kHz, and brute-forcing its ACF is thus about 0.9 million multiplications, which seems like a lot. A 1024-sample Fourier transform should be something like 0.02 million, and you can derive the ACF by DFT | take magnitude |

DFT. But if you're only looking to compare autocorrelation at three particular lags, you can do that much more simply, with only three multiply-accumulates per sample and without explicit windowing.

(A thought here for pitch tracking: if you're trying to track a changing pitch using autocorrelation, maybe you could hill-climbingly adjust a pair of lags bracketing the true frequency, like nostrils, and get by with only two multiply-accumulates per sample? That's maybe cheaper even than the Goertzel algorithm or a software PLL, although it does require a multiply.)

Topics

Balcony battery

Kragen Javier Sitaker, 2019-02-11 (updated 2019-12-06) (6 minutes)

It's summer in Buenos Aires, and of course that means we have regular power outages, during which not even the electric fans work. I now have an apartment with a two-meter balcony; I could definitely put a bench on it. And the bench could have marine deep-cycle batteries in it. To be comfortable to sit on, it should have 500 mm from the floor to the seat height; it could be, say, 1750 mm long; and it could probably productively be 500 mm wide, too. (I could add backrests if that's too wide for comfortable sitting, but I think it may actually be a bit scanty.)

A properly designed bench could conceal a lead-acid battery bank within it, and its placement on the balcony could provide it with good ventilation to prevent hydrogen buildup, as well as safe drainage in the case of a containment failure. If we have 50 mm on each face of the cuboid devoted to supports and the like, we still have 400 mm × 400 mm × 1650 mm. A 1650 mm × 400 mm bottom area is ⅔m². A typical car battery (BCI group 58, 58R, or 59) might be 255 mm × 183 mm × 177 mm high, occupying a base area of 0.0467 m², 14 times smaller. In practice I think you'd have to line the batteries up inside a bench of this size in two rows of 6 batteries, with their long axis parallel to the bench's, so you'd only get 12 of them in there.

Let's be more specific, though: there is a large D-BAT DB180N 12V deep-cycle nautical battery which claims 12 V, 235 mm × 235 mm × 520 mm, and 180 amp-hours (7.8 MJ), for sale for AR$6500 (US$176). If you made the bench just a little wider, you could fit seven of these motherfuckers (US$1232, 54 MJ at US$23/MJ) in there: 520 mm × 1645 mm long.

Or, take a more typical car battery like a Bosch S4 55 D. It sells for AR$2799 (US$76) and is 12 V, 170 mm × 170 mm × 240 mm, and 65 amp-hours (2.8 MJ). If you put the batteries at right angles to the bench, you could get 18 of them (US$1368, 50.4 MJ at US$27/MJ) into a 480 mm × 1530 mm floor area. You'd probably be better off with a deep-cycle type and with fewer electrical connections to go wrong, but at least this shows that the pricing and energy density on the other battery are not too far off.

Amazon has a 35-amp-hour (1.51 MJ) deep cycle battery for US$110 (US$73/MJ) which is 127 mm × 167 mm × 196 mm and weighs, in archaic units, "25 pounds", which means 11.3 kg. This works out to an energy density (probably fairly invariant for lead-acid batteries) of 133 kJ/kg, so a 50-MJ setup like those described above should weigh about 370 kg.

So with 50 megajoules in your pocket, so to speak, how long could you run things?

This laptop is using 9 watts. It would drain the battery pack in 1500 hours. It would drain a single 2.8 MJ car battery in 86 hours. That's a pretty long power outage.

There are little 3-watt USB fans, but a strong electric fan might be 75 watts. It would drain the large battery pack in 185 hours (almost 8 days) or a single 2.8 MJ car battery in 10 hours.

If you were willing to limit your total power drain to, say, 250

watts, you could hook up the batteries internally with fuse wire to get some added safety against electrical faults. If you put the seven deep-cycle batteries in series to get 84 volts, 250 watts would be just over three amps; you could use four-amp or five-amp fuse wire, which would be about AWG28 (0.32 mm) for iron wire, AWG24 (0.51 mm) for tin wire, AWG33 (0.180 mm; hard to find!) for aluminum wire (also hard to find), or AWG35 (0.142 mm) for copper wire.

For the limited but extremely important use of cooling things to 0° or above, a potentially better approach is to bank cool in the form of ice rather than batteries. 370 kg of batteries holds 50 MJ of energy, which can remove about 100 MJ of heat from your house. By contrast, 370 kg of ice with its enthalpy of fusion of 333 kJ/kg can remove about 123 MJ. Moreover, water costs several orders of magnitude less than batteries do, kilogram for kilogram; it's just a matter of the refrigerative apparatus to freeze it with and then, possibly, some kind of automated ice-cube handling apparatus to allow you to handle large quantities of ice without running delicate pipes through all of it. But the crossover in cost probably isn't until a GJ or so.

(With 2 kW of sun streaming in through the window, 100 MJ is only 14 hours of cooling; surviving a week-long heat wave would require several times that.)

Lithium-ion batteries

BloombergNEF reports:

Shanghai and London, December 3, 2019 – Battery prices, which were above $1,100 per kilowatt-hour in 2010, have fallen 87% in real terms to $156/kWh in 2019. By 2023, average prices will be close to $100/kWh, according to the latest forecast from research company BloombergNEF (BNEF).

If we remove the Sumerian units, that works out to US$305/MJ in 2010, US$43.3/MJ in 2019, and US$27.8/MJ in 2023. Why are these prices so much higher than the prices I observe *in the retail market* here in Argentina in 2019? Because Bloomberg is specifically talking about battery packs for electric vehicles, although they don't mention this until later in the article, and lead-acid batteries are far too heavy for any but short-range or aquatic electric vehicles.

But it's very interesting that BNEF is predicting that lithium-ion batteries will get into the lead-acid price range.

Topics

- Pricing (p. 3646) (89 notes)
- Energy (p. 3438) (63 notes)
- Household management and home economics (p. 3504) (44 notes)
- Batteries (p. 3340) (7 notes)

Usability of scientific calculators

Kragen Javier Sitaker, 2016-09-29 (19 minutes)

I just bought a scientific calculator (for US$27) with the purpose of cannibalizing its solar cell (it can also run off a button cell) and maybe display and keyboard. But I got distracted with its user interface, which is kind of amazing given the constraints it's under.

First, I want to be clear that the calculator is unfortunately rather limited in its power, and there are a lot of things that are clumsy about its user interface. And I've crashed into quite a number of usability problems. Despite all this, it does achieve many amazing things.

This is a Cifra SC-9100 sold by Unitronic SA here in Argentina, but likely they didn't design the hardware or write the software, both of which are probably from a white-label company in China, where the calculator is made. In part I base this inference on the fact that the error messages are in English.

It has a large 5×4 keypad for the most basic functionality (arithmetic, entering numbers, clearing), a 6×4 area above it with two keys missing for most of the functions, and four navigational/shift and four directional keys, for a total of 50 keys.

This calculator is not comparable to the TI-83 or HP-48 lines of graphing calculators with computer algebra systems onboard. But then, they can't run off a tiny solar cell, either. Evidence suggests that it has on the order of 2K of memory.

Feature list

To some extent this calculator seems to have been designed to fill out a feature checklist, because many of the features don't work together properly.

- 79-step infix algebraic expression evaluation with 9 variables with letter names.
- Interactive editing of expressions before evaluating them on a 12-character alphanumeric line with 5×6 pixels per glyph.
- When you have a syntax error, you can jump the cursor to it.
- Stored history of the last few (6–8 usually) calculations and results.
- A direction pad on the keyboard, which makes the history editing pretty easy.
- Decimal floating-point, d°m's", complex, and rational numeric types.
- The usual complement of standard numerical functions: square root, cube root, square, cube, exponent, nth root, log, 10^n, ln, e^n, sin, cos, tan, $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, sinh, cosh, tanh, $\sinh^{-1}$, $\cosh^{-1}$, $\tanh^{-1}$, nPr, nCr, +, -, ×, ÷, factorial, reciprocal, polar to rectangular, and rectangular to polar.
- Newton's-method-based solver for (single) arbitrary general equations; you can supply values for some of the variables and ask it to solve for another one, although this may not always converge.
- Special solvers for quadratic and cubic equations in one variable and for systems of linear equations in 3 or 4 unknowns.
- A sort of forms-based user interface for the solvers where you can

navigate around the fields.
• Display modes including fixed-point, scientific notation, engineering notation, which can be selected after the fact and then used to review previous results.
• A very easy but limited programming facility reminiscent of BASIC without control structures.
• Twenty predetermined unit conversions and their inverses, including things like horsepower, acres, mmHg (but not psi!), and parsecs (but not lightyears).
• 40 physical constants, including things like the Stefan-Boltzmann constant and the permittivity of free space.
• Hexadecimal, octal, and binary bases and bitwise operations.
• Statistics: mean, standard deviation, and regressions (linear, logarithmic, exponential, power, inverse (reciprocal), and quadratic).
• Numerical integration using Simpson's rule; approximation of numerical differentiation using finite differencing.
• Matrices, but only up to 3×3.

Limited BASIC-like programming facility

Here's a version of Minsky's circle program in a 3-place fixed-point display mode.

I type 1 SHIFT STO X to set X to 1; this results in the display

```
1→X
    1.000
```

Then 0 SHIFT STO Y:

```
0→Y
    0.000
```

Now I enter the program:

```
X=X-.1Y:Y=Y+.1X:X²+Y²
```

And I press the = key (the other = key) repeatedly to single-step the program as an endless loop:

```
X=X-.1Y
    1.000
Y=Y+.1X
    0.100
X²+Y²
    1.010
X=X-.1Y
    0.990
Y=Y+.1X
    0.199
X²+Y²
    1.020
X=X-.1Y
    0.970
Y=Y+.1X
    0.296
X²+Y²
```

```
    1.029
X=X-.1Y
    0.940
Y=Y+.1X
    0.390
X²+Y²
    1.037
```

Continuing in this vein, after 11 more iterations (33 presses of the = key) we get to X = 0.020, Y = 1.001, X^2+Y^2 = 1.002. (The third expression is just a kind of a sanity check.)

To me, this is an incredible kind of immediacy: in 8 keystrokes I can set up the initial state of the program, in 33 more keystrokes I can implement Minsky's circle algorithm with a debugging statement, and then I can immediately start single-stepping it. A further additional keystroke — the left or right direction on the direction pad — takes me back into editing the program.

You can similarly easily program it to, for example, tabulate the values of a function:

```
0→X
X(1-X):X=X+.1
```

This gives us the sequence 0.000, 0.090, 0.160, 0.210, 0.240, 0.250, 0.240, 0.210, 0.160, 0.090, 0.000 (-0.110, ...) just as it should, with two keystrokes per output number. Or, to tabulate the logit at intervals of 0.05:

```
.05→X
ln (X÷(1-X:X=X+.05
```

(Note the unterminated parentheses before the :. Also note that `ln` was one keystroke.) In 27 keystrokes, plus two per answer, this gives -2.944, -2.197, -1.735, -1.386, -1.099, -0.847, -0.619, -0.405, -0.201, 0, and so on, with two keystrokes per output number. This agrees with the results of `scipy.special.logit(numpy.linspace(0.05, 0.95, 19))`.

Excel's user interface is probably the most optimized for doing such iterative numerical calculations in the history of the universe. Doing the same version of Minsky's algorithm in Gnumeric, which uses Excel's user interface, is `1 TAB 0 ↵ =↑-.1*↦ TAB =↑+.1*↓↤ TAB =↤ SHIFT ^ 2 + ↤ SHIFT ^ 2 ↵ ↑ SHIFT ↤ ^C↓↤ SHIFT PgDn ^V`, which is 45 keystrokes as I count it — almost 50% more! (And that's not even charging extra for the fact that my netbook doesn't have a keypad, so even some basic arithmetic operators need shift.)

Of course, Gnumeric (or Excel) doesn't require three more keystrokes to get each new answer, doesn't forget the previous results, can automatically recalculate, can graph, and so on. And in my Gnumeric instance, that `SHIFT PgDn ^V` at the end pasted 16 iterations, which would be 48 keystrokes on the calculator.

But I feel that these issues of the accessibility, manipulability, and inspectability of the state machine's execution history are nearly orthogonal to the issues of specifying the state machine's transition function. Yes, the two sets of functionality do compete for the same keyboard keys. But specifying the same state machine took 33

keystrokes on 50 keys on the calculator (about 186 bits), and about 45 keystrokes on 82 keys on my netbook (about 286 bits).

The good news kind of ends here, though. As far as I can tell, there are no conditionals, no way to loop without manually single-stepping, no way to assign to multiple variables in a step, no way to store a vector in a variable, no way to (programmatically) access previous values of a variable, and no way to loop implicitly rather than explicitly. Worse, your program is lost if you press ↑ or ↓ on the direction pad (although you can access single steps of it, the colon-separated form is gone, and you have no copy and paste to reconstruct it with), if the calculator automatically powers off from inactivity, or if you start entering another expression.

It has a random number generator, but with only three digits, and I haven't found a way yet to round or truncate to an integer in a formula. (There is a Rnd key for rounding, but what it does is round the currently displayed result to the displayed precision, so to a first approximation it appears to do nothing. You can't use it in a formula.) The closest I've come up with is $2\sqrt{(2\sqrt{(2\sqrt{(2\sqrt{(\text{Ran\#}-.5)^2}-.5)^2}-.5)^2}-.5)^2}$; my theory was that this would eliminate one bit of precision each time. However, iterating the formula $2\sqrt{(\text{Ans}-.5)^2}$ (here Ans being the result of the previous formula) starting from .555 gives me the orbit .11, .78, .56, .12, .76, .52, .04, .92, .84, .68, .36, .28, .44, .12; so it's definitely lost some information, including after the first three iterations, but then it enters a period-10 orbit, so it doesn't keep losing information. .133 enters a different orbit.

The above does illustrate that you can use the command-line history facility to supply different inputs to an iterative algorithm over time.

So this level of programmability of this calculator is inspiring, but ultimately very limiting.

Newton's-method-based solver

In a way this is the most amazing part to me, because it's close to a general interactive constraint system. If I enter

```
Y=X^6-2X+1 [SHIFT] [SOLVE]
```

(which is a bit harder than it sounds because the "=" and each of the occurrences of a variable requires pressing the [ALPHA] key first) it drops me into a "form" (one line displayed at a time!) where I can supply values for Y and X. Immediately it asks me:

```
Y?
    28.000
```

If I answer by typing 2π=, it calculates that result, invisibly stores it in Y, and continues:

```
X?
     5.000
```

I can scroll up and down between the two lines of the form with the direction pad, but if I respond to the X? with a second use of

[SHIFT] [SOLVE], the display goes blank for about 6 seconds and I get the result:

```
X=
    1.418
```

(I happen to have the display mode set to 3-place fixed-point, which isn't quite enough for displaying the solutions of sixth-order equations.)

If instead I supply the value for X, move back up to Y, and press [SHIFT] [SOLVE], it calculates Y rather more quickly.

This isn't quite modeless — I can supply the value for Y by an algebraic expression, but not by specifying a different equation to solve, and I can't apply the solver to systems of equations with a : separating them (that gives a syntax error).

A related but sort of redundant facility is its CALC key; given an algebraic expression like

```
√(X²+Y²)
```

pressing CALC will query you to update X and Y in the same way (but mysteriously without the ability to scroll to the other fields), then display the result of evaluating the expression.

Cubic solver

To solve $x^3 - 4x^2 - 5x + 2 = 0$, I have a couple of different options; I can try the Newton solver, or I can use its cubic solver. The cubic solver is like this:

```
(MODE MODE MODE 3)
 Unknowns?   →
    2   3
(→)
↤Degree?
    2   3
(3)
 a?
(1=)
 b?
(-4=)
 c?
(-5=)
 d?
(2=)
```

(half-second delay)

```
X₁=          ↓
 4.93163002
X₂=          ↕
 0.323190091
X₃=          ↑
-1.254820111
```

These answers were quickly obtained, and they are correct as far as

they are displayed, although at first I mistakenly thought they weren't.

Obtaining a result using the general solver turned out to be a little trickier; I entered this expression:

```
0=X³-4X²-5X+2
```

and used SHIFT SOLVE twice, with X initially at 0. This ran about as fast and got the .323 solution. X initially at 2 also yields the same solution; with X initially at 3, I instead get the -1.25 solution. To try to find the third solution if we didn't already know it, we could try dividing out the other solutions; by using the left-arrow on the solution we can get back to edit the equation into

```
0=(X³-4X²-5X+2)÷(X-.323190091)÷(X+1.254820111)
```

With this version — which unfortunately I couldn't copy and paste the numbers into — I still get a solution at "-1.254820111" if X starts there, but now if X starts at 0, I get the 4.93 solution.

This is considerably more effort, but it's applicable to equations of any degree. It's not guaranteed to work, due to roundoff error, but it will often work.

It turns out that the 0= I'd stuck in there at the beginning isn't necessary. This works too:

```
X^6-2X+1-2π [SHIFT] [SOLVE]
```

From some starting point, it comes up with the answer X=-1.193859 or so. Given instead (X^6-2X+1-2π)÷(X+1.1939) and a start of X=-1, in about three seconds, it comes up with another solution at X=1.417685166. If we divide that factor out too, it comes back with an error message, "Can't solve", after about 45 seconds or a minute; and indeed (upon graphing the equation elsewhere) those seem to be the only two real roots of this polynomial. (In a sense it should be obvious that there are only two real roots from its form.)

Numerical integration and differentiation

To calculate $\int\sqrt{(1-x^2)}dx$ from -1 to 1 with 512 partitions using Simpson's rule, I type

```
∫√(1-X²),-1,1,9=
```

and wait about 13 seconds. It yields 1.5707, which is accurate almost as far as it goes, but that isn't very far. (The exact answer is $\pi/2$, close to 1.570796.) If instead of 9 I specify 7 (for $2^7 = 128$ partitions rather than 512) it runs in about three seconds and produces 1.57 as the result.

This manages to be simultaneously counterintuitive (I never would have guessed either the comma, the logarithmic notation for the number of partitions, or the limit of 512 partitions), opaque (it's no easier to write than to read), and impressive (that the calculator can do this at all).

As a crude approximation, this speed suggests that it's running a few hundred floating-point operations per second.

Numerical differentiation is faster, of course. It uses a finite

difference, because apparently the numbskulls who implemented the calculator's software haven't heard of automatic differentiation yet. To tabulate the derivative of the logit, for example:

```
.05→Y
d/dx(ln (X÷(1-X,Y,1e-9:Y=Y+.05
```

This gives 21.052, 11.111, 7.844, 6.250, 5.333, 4.762(166667), 4.395, 4.167, 4.041, 4.001, 4.040, and so on. These are accurate to nearly four places, which is fairly underwhelming given that I specified using $\Delta x=10^{-9}$. The extra digits on the 4.762 number suggest that it's not actually bothering to calculate the intermediate results to more than about five places.

Base-N and logical operations

In base-N mode, the trigonometric keys become hexadecimal digits, and the reciprocal key opens a menu of bitwise operations. This allows you to explore mappings like `Ans and(Ans-1)` or `Ans xor(Ans-1)`, which last generates a Sierpinski triangle in binary, one row at a time, if you seed it with a power of 2. Unfortunately in this mode you apparently cannot assign to variables, not even X, Y, and M.

However, I did manage to get it to tabulate reflected-binary Gray code values using two keystrokes each. First I set M to 0 using `RCL M SHIFT M-`, and then I entered and ran the program `1M+:Mxor(M÷2`, getting the hexadecimal sequence 1, 3, 2, 6, 7, 5, 4, C, D, F, E, A, B, 9, 8, 18, 19, 1B, 1A, 1E, 1F, 1D, 1C, 14, 15, 17, 16, 12, 13, 11, 10, 30, ... sequence A003188. (The program also works in decimal mode in base-N.)

Modefulness

This calculator suffers terribly from modefulness. You can't take absolute values except in complex-number mode; you can't use trigonometric functions in base-N mode (although you can use arithmetic) because the keys are remapped to hex digits. You can't use square-root in base-N mode for no apparent reason; the key just doesn't do anything. You can't numerically integrate in complex mode. You can't assign to variables in base-N mode or complex mode. And so on.

Worse, in some cases the meaning of programs is dependent on the mode. In base-N mode, any numeric constant in your program will vary depending on base — and most give a syntax error in base 2. Worse, this happens when you merely try to scroll back in history to the formula.

Usability lessons

Except for when asked to do significant amounts of computation or when dealing with a key that maps to nothing, the calculator is always instantly responsive. Its silent keyboard, despite presumably being rubber carbon-bottomed domes under hard keys, works reliably. These are big pluses.

Bigger still is the immediacy of most operations. Doing base conversion with a single keypress in base-N mode is really nice. This doesn't excuse the absence of features like variable assignment in base-N.

The form-based UI for off-the-shelf models (e.g. cubic equations)

really eases using them, but makes composing them difficult to impossible.

Writing the transition function of a numerical finite state machine as a series of colon-separated assignments works surprisingly, even amazingly, well. Being able to nest assignments and to compute and display two or three results per keystroke (rather than just one) would help. Being able to run the iteration out of user control (with conditional and iteration operations) would make an even bigger difference. The single-step-and-see-labeled-result one-keystroke flow is magical, though.

After my initial astonishing experiences with the device, I was starting to become attached to it and doubt my tentative plans to rebrain it with a low-power high-performance microcontroller; my further investigation, however, shows that it's actually kind of an unusable piece of shit. Current mass-market low-power ARM microcontrollers like Atmel's picoPower line run at 12 milliwatts at 48 million 32-bit instructions per second, which is 250 pJ/insn; my best estimate is that in full sun this calculator's solar cell can produce 29 milliwatts.

TI's current Nspire line of graphing calculators run on 150MHz ARM microcontrollers with 64MB of RAM. But they don't run off a solar cell.

I can't find the information on the ARM picoPower microcontrollers I was thinking of.

Topics

A two-operand calculator supporting programming by demonstration

Kragen Javier Sitaker, 2018-12-11 (22 minutes)

Historical programmable calculators have mostly either used BASIC-style infix expressions and jumps, or FORTH-style stack operations. When I'm programming, I find FORTH code more bug-prone than register-to-register code. Could you design a usable programmable calculator using a non-stack-machine machine code? I think so, and in fact, I think you could make it more usable than traditional calculators by taking advantage of programming by demonstration.

Or, to put it another way, how easy could you make programming in raw machine code, so that you don't wish for an assembler?

Physical description

In front of you is a box with an unusual keyboard and two knobs, one above the other, each with a 16-digit numeric LCD display next to it. Each knob has 16 positions around it labeled with the letters J through Y. A third knob with a third display is located below the keypad; it also turns indefinitely in either direction, with a larger number of detents. The top knob and display are labeled Γ; the bottom knob and display are labeled Δ. Turning a knob to one of its 16 positions selects the number displayed on its display; there are 16 different such numbers. Both knobs access the same set of numbers; when they are turned to the same position, they display the same number.

There is also a peripherals box connected to it with a cable which will be described later.

The keypad looks like this:

0↓ 1↓ 2↓ 3↓ ± B C
7 8 9 + ~ > ←
4 5 6 - ^ : ↓
1 2 3 × & ; ⇐
0 @ ! /% | { }

XXX missing: ››, ‹‹, some way to swap stacks

Calculation

The numeric keys enter a number into the upper display, Γ. If a number was there previously, it is replaced, unless the last thing you did was also a numeric or backspace key, in which case it is appended to the number there. The backspace key (⇐) removes the last digit from the number. If you turn the Γ knob away from that position, the displayed number will change, but when you turn it back to that position, the number you entered is still there. If the Δ knob is in the same position, the same number shows on both displays, and they update together.

This is because the machine contains 16 registers, named P, Q, R, S, T, U, V, W, X, Y, J, K, L, M, N, and O, and the knobs select which of the registers is displayed on each display at each moment.

The arithmetic keys +-× alter the number on the Δ display by respectively adding to it, subtracting from it, or multiplying by it the number on the Γ display. The Γ number remains unchanged, unless both knobs are in the same position, in which case it shows the same number as Δ.

The arithmetic key /% alters Δ by dividing it by Γ, leaving the quotient in Δ, but does not leave Γ unchanged; rather, it leaves the remainder in Γ. The quotient is rounded toward zero, and the remainder is consequently sometimes negative.

XXX maybe use 32.32 fixed-point? × differs only by a right shift but /% differs radically.

The ± key changes the sign of the number in Γ.

So far, this allows simple use of the machine as an integer (XXX?) calculator. You select some register with both Γ and Δ, type a starting number into it, and then select a different register with Γ and type more numbers into it and apply operations to update your Δ accumulator.

The keys ^~&| are similar to the arithmetic keys, but perform bitwise operations on the numbers: XOR, NOT, AND, and OR, respectively. ~ applies only to Γ, not to Δ, on the theory that much of the time that you invert the bits in a number, it's because you're about to use it to clear some bits in something else. You can pull out either knob to set the corresponding display to hexadecimal, or push it back in to go to decimal.

Memory

The @ and ! keys use Γ as an address into a larger "main memory" of 65536 numbers; @ reads main memory at the address Γ and copies it into Δ, while ! copies Δ to main memory at the address Γ. The display below the keypad displays one of the numbers in this memory, and the knob moves backwards and forwards in this memory. It has a narrow stem in the middle to allow spinning it quickly, and this implements something like mouse acceleration so that you can scroll rapidly through memory. Main memory is nonvolatile, retaining its contents even when the device is unplugged.

Programmability

The other keys are somewhat more difficult to explain, because they pertain to the device's programmability.

Defining subroutines

Whenever you are typing, the program counter displayed in the display below the keypad advances, recording your instructions in the memory so that you can replay them later; numeric entry is not recorded until it is complete, so backspace does not create a program instructions, nor does typing a number followed by : or a few other things. The : key marks a label, which can be used as the beginning of a subroutine or loop. This is implemented by storing the current program counter at address Γ and magically restoring Γ to its previous value. The ; key marks a return address; it has no effect on the Γ and Δ registers, so its only immediate effect is to store an instruction in memory.

This means that you can define a subroutine by doing a calculation, typing ;, then using the third knob below the keypad to

scroll back in memory to the beginning of the calculation, entering a subroutine number, and pressing :. Or you can enter the subroutine number and :, then do the calculation. This third knob's somewhat larger display shows a main memory address and its contents in two formats: as a decimal number and in a strange alphabetic hexadecimal format, the same one used to label the register knobs.

Only numbers in [0, 511] are valid labels due to the instruction encoding.

There is the potential to accidentally overwrite things by recording an interactive computation in this way, so you may want to use the third knob to jump backwards from time to time when you aren't recording things on purpose.

Interactive mode and execution mode

Once a subroutine is defined, you can invoke it with the ↓ key, which calls the subroutine whose label number is displayed in Γ by pushing the program counter on a stack and fetching from memory at the number displayed in Γ into the program counter. An extra bit in the program counter tracks whether the machine is in interactive mode or execution mode, so upon returning from the subroutine from an interactive call, the machine reverts to interactive use. The B , and C keys interact with this bit: B in execution mode sets the machine to interactive mode (pausing whatever program is running) and C sets it to execution mode, resuming the paused subroutine. B in interactive mode is used for single-stepping; in most cases, it simply interactively executes the instruction at the program counter without leaving interactive mode, but if that instruction is a ↓ or ;, its effect is slightly modified — the interactive bit remains set after the context switch, so you can single-step through the subroutine as well. (Or you can continue until its exit with C.)

This fact of having an interactive bit saved on the stack with the return address allows you a lot of debugging flexibility. You can interrupt the program at any point with B to see what it's doing, then C to continue, even if you're nested inside some other similar debugging session higher up the stack. You can update the contents of the registers by typing numbers followed by B to step to the next instruction.

XXX exactly when entered numbers get stored into the instruction stream needs some clarification. What if you turn the Γ knob to modify multiple registers before typing B?

Shortcut keys

The 0↓, 1↓, 2↓, and 3↓ keys are shortcut keys to call the subroutines whose addresses are stored at addresses 0, 1, 2, and 3. This allows you to implement simple applications just by vectoring those functions to subroutines of your choice and, perhaps, placing stickers on the keys.

Local variables

Registers P through W are saved and restored on the stack in memory along with the program counter by the ↓ call and ; return instructions, making them local variables in each subroutine. However, because they are copied to the stack during the call instruction, their value in the caller is available in the callee; so you can use them to pass parameters, and the callee's changes to those

parameters will be automatically discarded upon return. So if the callee wants to return values to the caller, it should put them into one of the other registers, such as X and Y.

Conditionals

The { key adds a conditional jump instruction to the program and stores the address of that jump instruction on a special small stack not stored in main memory; the } key then backpatches that conditional jump to point to the current program counter. The conditional jump jumps if Δ is zero. In order to preserve the programming-by-demonstration feel of the machine, all 16 data registers are also stored on that stack and restored when you type }.

The > key sets Δ to 0 if it was negative. This means that the sequence ->{…} executes the … only if Δ was greater than Γ originally.

Unconditional jumps and loops

The ← key fetches an address from the address Γ points to and adds a jump to that address to the current program. This is basically intended for loops: you type, say, 8: at the beginning of the loop, and then 8← at the end of the loop; but it can also be used for subroutine tail calls. I'm not sure what it should do in interactive mode.

Generally you will want to use { and } inside the loop in order to make it not infinite. In particular, {;} conditionally terminates the subroutine the loop is within.

Wide program counter

Because memory words are fairly large (64 bits?) several instructions can normally fit into one; the opcode field is 4 bits and the source and destination register fields are 4 bits each, for a slightly awkward total of 12 bits. This makes the program counter three bits wider than the usual 16-bit memory addresses, plus the interactive bit, which allows single-stepping at sub-instruction-word granularity, as well as returns, calls, and jumps into the middle of instruction words.

Interrupts

No.

Peripherals

In addition to the three numeric displays already mentioned, there are some other peripherals on the calculator, mostly in the peripherals box.

When the value of the last register, register O, changes, there is an audible click whose volume increases with the magnitude of the change. This is because register 15 is connected to a digital-to-analog converter which is connected to a speaker.

The peripherals box has eight smoothly turning dials on it. The current positions of these dials are mapped into main memory at positions 0x7f00 (WOPP) through 0x7f08 (WOPX).

The current time is always available in memory at address 0x7f09 (WOPY).

There is a 320×240 touchscreen LCD display on the peripheral box with 16-bit color and an integrated framebuffer. The current touches are available in addresses 0x7f10 (WOQP) through 0x7f14

(WOQT); each one contains X and Y coordinates and an active bit to indicate whether it's active or not. Its current vertical scan position is in 0x7f15 (WOQU). By placing 80 64-bit words in the addresses starting at 0x7f20 (WORP) and writing a scan line number to 0x7f70 (WOWP), you can update a scan line on the display.

There is a switch on the peripherals box attached to an LED that illuminates a sign saying "RECORDING" and a microphone. When the switch is turned on, the LED flashes, and the current microphone sound pressure level is available at 0x7f30 (WOSP).

When in execution mode rather than interactive mode, the current 34-bit state of the keyboard (except for the B key, which returns to interactive mode) is available at 0x7f40 (WOTP).

Machine code encoding

This is especially important on this machine because if you're writing the machine code without an assembler, you're probably also reading it without a disassembler, and indeed the third knob and display are designed for this.

Each instruction except for immediate constants is 12 bits, 3 nibbles. A memory word is 64 bits, 16 nibbles; this allows space for five instructions in 60 bits, plus an extra one-nibble tag field which is used for literals. Normally it is 1111 (f/O) but when it is 0000 (0/P) the word is a literal word with 56 bits of immediate data, here nibbles labeled nd through n0.

nibble f e d c b a 9 8 7 6 5 4 3 2 1 0
normal 1111 op0 A0 B0 op1 A1 B1 op2 A2 B2 op3 A3 B3 op4 A4 B4
literal 0000 reg nd nc nb na n9 n8 n7 n6 n5 n4 n3 n2 n1 n0

The fields labeled An and Bn vary in interpretation according to the instruction format; for two-operand instructions, they are generally the source and destination registers; for one-operand instructions, An specifies the instruction and Bn specifies the operand register; and for zero-operand instructions, all three fields specify the instruction.

Two-operand instructions

Optimizing for 12 bits and hexadecimal legibility, the two-operand opcode format is as follows:

opcode Γ Δ
xxxx xxxx xxxx

The Γ source and Δ dest fields each identify one of the 16 registers (with the low 4 bits of the register's letter), and the opcode fields identify one of the two-operand operations:

conventional hex 0 1 2 3 4 5 6 7 8 9 a b c d e f
operation + - × /% { ^ & | ! @ one-operand (see below) (control flow) (see below)
ASCII alphahex P Q R S T U V W X Y J K L M N O

This means, for example, that the three-nibble sequence PTJ means "J = T + J".

For most of these operations, as explained earlier in the section about the user interface, the source operand Γ is left unchanged and

the destination operand Δ gets the result. There are two exceptions: /%, which like the /% key, leaves the remainder from the division in the source operand; and {.

{ tests the source operand Γ to see if it's 0, and if so, executes a forward jump. The destination operand field Δ is not treated as a register reference at all. Rather, its value from 0 to 15 indicates a number of instruction *words* to skip over, starting from the instruction word following the word containing the { instruction. That means that when the Δ field is 0, the conditional jump goes to the beginning of the next instruction word, skipping up to four instructions; when the Δ field is 1, the conditional jump goes to the beginning of the instruction word after that, skipping 5, 6, 7, 8, or 9 operations (or possibly only 1 to 5 if the next instruction word was a literal word); and so on up to 15, which skips 15 entire instruction words, plus the remainder of the word containing {. If the register indicated by Γ was nonzero, however, instruction continues as if nothing had happened.

This means that the } key may need to insert nop operations to fill out the current instruction word, so that it has a word-aligned address to backpatch the { instruction with. This is an annoying amount of complexity to attach to a key, but the alternative would be to only enable { to skip over up to 16 *instructions*, rather than up to 84, which would be a painfully anemic conditional. (But hey, HP calculators' conditional only skipped a single instruction.) Still, the complexity involved here is small compared to the complexity of division or procedure calls.

One-operand instructions

There's a different format for one-operand operations:

opcode xop reg
1011 (b/K) xxxx xxxx

The one-operand operations are encoded with the following xop values:

conventional hex 0 1 2 3 4 5 6 7 8 9 a b c d e f
operation 0 1 2 ~ ± > ;/nop -2 -1
ASCII alphahex P Q R S T U V W X Y J K L M N O

The operations named with numbers set the register to that constant.

Zero-operand instructions

; and nop need no operands, so we distinguish between them by the register field: 0 for nop, 1 for ;.

Instructions taking labels

Looking at what's left of the keyboard, : just defines a label; it doesn't add anything to the instruction stream. B break, C continue, and ⇐ backspace similarly are purely interactive operations. However, both ← and ↓ require a label number, which is not actually taken from Γ (that's just the interactive user interface!) but encoded in the instruction. These are encoded with a 9-bit label number field:

opcode label (meaning)
110 x xxxx xxxx ←, goto
111 x xxxx xxxx ↓, call

Literals (immediate data)

As mentioned above, literals larger than ±2 are encoded using an entire instruction word by putting tag 0000 in the first nibble of an instruction word, followed by a destination register and 56 bits of literal data. The literal data is sign-extended to put it into the specified register. In particular, this means that a memory word containing a positive number less than 2^{56} can be interpreted as an instruction to load that number into register 0 (P), which is a technique that you can use to improve the legibility of your program.

Examples

So an instruction word written in alphahex as O PTJ PUT KTU KWP consists of two addition instructions (J += T; T += U), a negation instruction (U = -U), and a nop.

XXX reshuffle opcodes for ASCIIhex legibility. P for + is good, maybe M or L for -, T or M for ×, O or Q or R for /%, N for ~, X for ^, S for !, L for @, which leaves the less important & and | to get less desirable letters. But we need groups of two letters for ← and ↓, chosen from PQ, RS, TU, VW, XY, JK, LM, or NO. JK suggests "jump" and TU suggests "to" (maybe shuffle × off to M), but nothing in particular suggests "call" or "invoke".

XXX maybe use BCD?

XXX maybe merge &, ^, and ~ into a NAND operation, given their relative rarity? Maybe add a MOV operation?

Thanks

This document draws on discussions with Sean Palmer and John Cowan.

Topics

Image approximation

Kragen Javier Sitaker, 2019-05-14 (10 minutes)

I've mentioned the problem of image approximation in a few different notes — Ideas to ship in 2014 (p. 1409), Simplifying computing systems by having fewer kinds of graphics (p. 1110), and Ideas to pursue (p. 1084). Basically the idea is a specific application of mathematical optimization algorithms: making something look, visually, like some reference image. But it turns out to be an interestingly fertile family of applications.

There's now a Twitter bot called something like "Minimal Pictures" which illustrates the idea; it applies some single graphical primitive (Bézier curves, translucent rectangles, or translucent triangles, for example) a few hundred times to get an image that looks similar to a reference image, but not so similar that you can't see how it's done. And in 2016 I wrote a hill-climbing optimizer to approximate a given image with opaque irregular pentagons, with stunning, but very slow, results.

So here is an attempt to outline the space a bit.

Design spaces

A design space is a set of variables the optimization algorithm can tweak to satisfy its objective of making the right image. This can be pretty much anything. Interesting design spaces include polygons (translucent or opaque); polylines; Bézier curves; points; DCT coefficients in a JPEG; images constrained in some way, such as sparsity, to be convolved with the known, estimated, or optimized OTF of some optical system; widths and/or slight displacements of a given set of lines, as in engraving or font hinting; toolpaths, or more generally robot movement plans; arrangements of a given set of pieces (overlappable or not, scalable or not, but generally rotatable); heightfields, 3-dimensional meshes, or other three-dimensional models such as collections of spheres; parameters controlling a model like a face puppet; or colors of pixels chosen from a small set, as on an inkjet printer.

Similarity metrics

The simplest similarity metric to optimize is probably just the sum of squared pixel differences, but this leaves a lot to be desired. A very simple alternative is blurring the images a bit first, but there are many other possibilities. Transforming the images into Fourier or Gabor space and then weighting the different spatial frequencies according to perceptual salience might be an improvement. A perhaps more interesting possibility is using the first few layers of a neural network trained for image classification or in a GAN; this would presumably put extra weight on things that are important to "real images" in some sense, and might do a reasonable job of simulating the first few layers of human visual perception, too.

In some cases, perhaps something like Canny edge detection would be a useful step in the similarity function, if the visual similarity you're going for isn't purely literal.

Other objective function components

The objective function you optimize might not be purely similarity; it might also include some kind of "cost model" for the design space. For example, if you're planning a toolpath, you might want to prefer faster toolpaths; if you're trying to approximate an image by deriving a sparse set of stars or other bright points that have somehow gotten convolved with a known or unknown OTF, you might want to prefer sparser sets; you might prefer heightfields that are mostly smooth rather than containing thousands of sharp spikes; for artistic purposes, you might want to minimize the number of graphical primitives used, just to keep the approximation from being too perfect; and so on.

Simulation

In many of these cases, you won't know how well you've actually approximated the image until you set some kind of physical machinery in motion. A robot wielding a pen, a hot wire, a FDM nozzle, or a milling cutter can produce somewhat unpredictable results. Also, it's slow. A potentially worse problem is that its results aren't differentiable — you can evaluate your objective function over them, but you may not be able to determine the gradient of the objective function with respect to your design variables.

So in many cases you'll want to do some kind of simulation, even if in the end what you're optimizing is a physical process. A numerical simulation, even of things like robots cutting things, can support automatic differentiation with respect to your design space, which enables the use of non-gradient-free optimization algorithms.

Optimization algorithms

There's been an enormous amount of work on variations of gradient descent in the last few years, largely with an eye to accelerating learning of parameters for deep neural networks, and there are also a substantial number of older variants. This includes Nesterov accelerated gradient, AdaBoost, AdaGrad, Adam, and so on.

Gradient-based algorithms have an enormous advantage in high-dimensional design spaces; they tend to slow down linearly with dimensionality, while gradient-free optimization algorithms commonly slow down exponentially with dimensionality.

However, other metaheuristics other than gradient descent are applicable; for example, Nelder–Mead optimization, genetic algorithms, and simulated annealing.

In some cases, the most appropriate optimization algorithm may actually be something like a particle filter.

Purposes

This family of techniques can be applied to many problems.

The minimal-picture bot mentioned above has output that looks cool.

Non-photorealistic rendering is a useful approach for avoiding the plasticky hard look of a lot of 3-D rendering. Image approximation is a general approach to NPR — you ask for an approximation of a conventional near-photorealistic rendering, using a design space that

only allows pencil drawing, for example, or pastel chalk painting, and perhaps using a similarity measure that privileges whatever you think is most important to preserve, such as edge orientations.

In the opposite direction, a possible way for someone to specify a 3-D model that can later be used for photorealistic rendering, virtual puppeteering, or 3-D printing is to sketch it in a non-photorealistic way, maybe from a few different points of view, then optimize in an appropriate design space of 3-D models, using photorealistic rendering to generate the image to be compared to the sketch, and some kind of similarity metric that emphasizes the features that are likely to survive.

Autotracing, vectorizing raster images, can be framed as an image approximation problem: you want to approximate the raster image as closely as possible with a polyine with a very limited number of control points. Carrying this further, you could optimize a representation of a printed book that consisted of one or more fonts and a vector of (pageno, x, y, codepoint) tuples, subject to appropriate penalties for wiggly lines of text and fonts with too many glyphs, in order to simultaneously OCR the text and digitize the font used. This is presumably not too far from what current bilevel image compression does.

Star tracking for spacecraft orientation control can be viewed as an effort to approximate a series of images by estimating the camera's OTF, the attitude of the spacecraft, and the contents of the galaxy (although we already have a pretty good prior on that). Sun glare and Earth glare need to either be taken into account or somehow ignored by the similarity metric.

Grid fitting and hinting of fonts can be viewed as an image approximation problem. The difficulty with just doing nearest-neighbor things is that you can end up with uneven spacing, uneven widths, etc., which are more perceptually salient than what you got right. Optimizing the rasterized glyphs according to a perceptual model, for example in Gabor space, should solve this problem.

Dithering is the problem of approximating an image using only a few colors, and existing dithering algorithms face unappealing tradeoffs between edge sharpness and color precision. By optimizing the similarity between the dithered image (which, note, is not a continuous domain) and the original image as measured by a perceptual model, you should be able to get substantially better results, privileging edges where those are more perceptually salient, and color precision where that is, for example where there are no edges. Dithering algorithms could even take into account LCD subpixels if it's just a matter of tweaking the similarity function.

Lossy compression is, generally, the problem of approximating an image using some kind of sparse representation, and is of immense importance right now. Again, optimizing the similarity between the original and decompressed images according to a perceptual model might give improved results.

Resampling of images to different sizes could also be viewed as a process of image approximation, using a larger image to approximate a smaller one, or vice versa. This requires a similarity metric that penalizes you if sharp edges get blurry when the image is enlarged.

Structure-from-motion (SfM) and structure-from-shading can be

formulated as image approximation problems; you want to approximate an image or series of images with a colored point cloud and a camera trajectory, or possibly some other kind of 3-D model like a heightfield. In some cases lighting may be changing. By using a sufficiently fancy renderer, specular reflections and refraction can provide information about 3-D structure.

Topics

Statically bounding runtime

Kragen Javier Sitaker, 2016-07-19 (4 minutes)

I want to implement preemptive multithreading inexpensively at the level of a language compiler with a hard upper bound on response times, without depending on interrupts. This means that the compiled program has to yield control to a scheduler every so often, and we need to statically guarantee that it never goes for a long time without yielding; but, while the scheduler check is relatively inexpensive, it isn't free — maybe 100 or 1000 CPU cycles. (For example, if the scheduler calls select() to see if there are I/O events that need to be handled, that's about 3000 cycles.) So we'd like to use static analysis to avoid inserting it in too many places if we can.

Let's suppose our program is built from the following grammar:

```
function ::= name expr
expr ::= instruction | call | seq | forloop | whileloop | conditional
call ::= "call" name
seq ::= expr ";" expr
forloop ::= "for" var const expr
whileloop :;= "while" expr expr
conditional :;= expr "if" expr "else" expr
```

This version has a statically computable control-flow graph and therefore doesn't accommodate function pointers; you could treat them as CLOS-style generic functions and treat them as conditionals on a type test.

I'm glossing over parameter passing and local variable allocation for the time being; you can consider them to be implemented by some unspecified instructions.

It seems clear how to compute a conservative approximation of run-time-before-yield. whileloops can run forever, as can recursive functions; either of these need to have a yield inserted into them. Then, the run-time-before-yield of a call expression is the run-time-before-yield of the function it calls, plus a tiny amount; each instruction presumably has a known worst-case run-time-before-yield; a seq expression's worst-case run-time-before-yield is the sum of those of its children; a forloop's is N times that of its body expression, where N is the constant; and a conditional is the maximum of its two branches plus its condition.

It may be necessary to separately compute a worst-case run-time *before* yielding and a worst-case run-time *after* yielding for each syntax tree node.

Note that the above remains safe even in the presence of early-exit instructions from either loops or entire functions: those might shorten the run-time, but they can't lengthen it.

Inserting yields optimally is almost certainly an NP-hard problem.

Ideally if you have a recursive loop of functions, you would only insert a yield into the prologue of one of the functions rather than all of them.

You can generally do a better approximation. Most while loops are not intended to run forever; by supplying a "loop variant", an

integral nonnegative quantity that strictly decreases every iteration, you can guarantee that the loop terminates and even provide an upper bound on its iteration count. But that level of detail probably isn't necessary for inserting yields, and it probably requires extra work from the programmer.

This kind of analysis also works, with some tweaks, for bounding stack depths and total heap allocation size, either to optimize the number of checks down toward some limit or to provide strong static bounds on resource usage, statically guaranteeing resource adequacy. (Heap allocation can be twice as fast if it doesn't have to check the allocation pointer against the end of the arena, and implementations of threading and call/cc that use segmented stacks also normally have to bounds-check the stack poitner on every function call.)

Topics

The book written in itself

Kragen Javier Sitaker, 2016-06-12 (updated 2016-06-14) (18 minutes)

Darius Bacon has suggested writing a software system that is a "book written in itself": a book-length hypermedia literate program which is fully self-sustaining, all the way down to hardware, and accessible to any interested reader.

"Self-sustaining" means that the book contains full compilable source code for itself, for a compiler with which it can be compiled, and for a computer on which to run it.

Comparison to other related projects

In a way, this is a project analogous to Knuth's magnum opus, The Art of Computer Programming, in that it has to cover a very broad spectrum of computer applications — minimally: hardware design, compilers, filesystems, operating systems (also known as kernels), garbage collection, dynamic dispatch, windowing systems, font design, and typographical layout; but possibly also including text editors, video and audio codecs, data compression, database query evaluation, HTML parsing, constraint systems, type inference, theorem proving, SAT solving, cryptography, full-text indexing, games, numerical methods, SPICE-like electrical simulation, CNC fabrication, and OCR (to be able to scan its own text).

Other similar projects are the "NAND to Tetris" book, formally known as *The Elements of Computing Systems*; and Charles Petzold's *Code*; the Thwaites Toaster Project; the Dave Gingery "Build a Machine Shop from Scratch" series; the Primitive Technology series on YouTube; the 1978 port of Smalltalk-76 to the portable 8086-based NoteTaker computer, known as Smalltalk-78, which weighed in at 200 kilobytes; Project Oberon; Plan 9; the VPRI STEPS program to rebuild a full personal computing environment in twenty thousand lines of code; and the biological cell.

Differences from TAOCP

The project has two enormous differences from TAOCP and some smaller differences.

First, it's much less ambitious in scope — TAOCP seeks to provide an overview of all known solutions to a problem, telling the story of their historical development and mathematically analyzing the limitations and advantages of each solution; the Book-In-Itself need ony provide one solution. For example, TAOCP's volume 3 is "Sorting and Searching", which covers nothing more than sorting and various kinds of table lookup (in trees and hash tables) for 800 pages. The Book-In-Itself might include a single kind of hash table, or it might need three or four different kinds of hash tables for bootstrapping or efficiency reasons. It probably needs sorting; a single implementation of Heapsort is likely adequate. In any case, it's unlikely to spend more than three pages on sorting and searching.

So this reduces the project scope by a factor of 300.

Second, it's much more ambitious in scope — TAOCP's scope is limited to software, and even more so, to generally applicable algorithms. You aren't going to find a filesystem implementation, a

gate-level CPU design, or a text layout engine in TAOCP; although its author found it necessary to design his own font design system, his own fonts, and his own text layout system for TAOCP, those are described in other books.

This probably increases the project scope by a factor of 10. This multiplies out to about 30× reduction. Since TAOCP has taken 50 years so far and may be done in another ten or so, this suggests that the Book-In-Itself should take about two years of work.

Smaller differences are that the Book-In-Itself is almost certain to be written by lower-quality authors; it's likely to be a team effort, rather than an individual effort; and, simply in order to be feasible, it's likely to suffer from Dirac-like levels of terseness.

Differences from "NAND to Tetris"

The NAND to Tetris book (*The Elements of Computing Systems: Building a Modern Computer from First Principles*, 320 pages, 2005, Nisan and Schocken) describes a computer system, starting from NAND gates and D flip-flops, moving up through sequential logic, the design of a Harvard CPU called "Hack", assembly language, bytecode interpretation, a simple recursive-descent compiler for a "high-level" language called "Jack", an "operating system", and some video games.

This is thrilling, and additionally the authors have separately published a series of exercises and computer software to guide readers through the implementation of the entire system themselves, rather than simply providing them with working code.

Unfortunately, some aspects of the system are sort of fake, and some others are overcomplicated:

• the "operating system" is just a set of device drivers and doesn't provide multitasking or a filesystem;
• the "high-level" language Jack doesn't provide garbage collection, function pointers, or even dynamic dispatch, and its compiler emits bytecode for a stack machine implemented in software rather than machine code;
• the hardware definition language seems to have been designed by software people so soft they don't know what dynamic RAM is, don't know the difference between a chip and a printed circuit board, and have never seen a RISC instruction set;
• the hardware doesn't have virtual memory, interrupts, any I/O devices other than a screen and keyboard, or any way to modify its program memory;
• the tokenizer and parser output XML;
• although they have infix expression parsing, it doesn't handle operator precedence;
• there is this entire unnecessary bytecode virtual machine;
• the absence of a filesystem prevents you from running the compiler under the "operating system";
• the whole system is so slow that students are never expected to run the whole thing at once, always using a provided efficient implementation (written in Java) of the layer below the one they're working at. Their Pong game is 400 lines of code and compiles to twenty thousand lines of assembly.
• although this is subjective, all in all, the book has a somewhat leaden

and uninspired feel; even the authors describe Jack as "clunky". After each chapter, one is left with the sad feeling that the functionality of the system at that point is surprisingly weak, given its complexity, and the forlorn hope that it will get better in the next chapter.

By contrast, the Book-In-Itself will not rely on separately provided software; it will use more tasteful design choices to get surprisingly powerful functionality instead of surprisingly weak functionality; and it will be efficient enough to run its entire software stack unmodified on a CPU emulated in software on a modern CPU.

One gets the idea, for example, that Jack's syntax is "clunky" in part by the authors not wanting to introduce backtracking, perhaps because they don't know about it, although they published their book three years after Bryan Ford's thesis came out.

Differences from Petzold's *Code*

Where NAND to Tetris is somewhat too theoretical, idealized, and plodding, Petzold's *Code* (1999) goes into nitty-gritty detail, but doesn't teach you enough to build a working system. It's considerably more hardware-focused than NAND to Tetris, which is sort of unfortunate, since Petzold's hardware knowledge is a bit weak.

On page 304, for example, you have the pinout of the 2102 one-kibibit MOS SRAM chip, state of the art in the 1970s. At times Petzold seems to be trapped in the 1970s, too; he claims that TTL is faster than MOS, which stopped being true about 15 years before he published this book.

But the book includes a lot of detail that's hard to find coverage of elsewhere outside of much more specialized books, including things like DIP switches, ripple-carry adder circuits made of relays, Samuel F.B. Morse's hobby of making daguerreotypes, the PARC visit inspiring Macintosh, bit-shift operators in C, the atomic number of lithium, the PostScript format, Java bytecode, how the resistance of tungsten varies with temperature, CP/M filesystem and memory layout, the dynamic range of the CD-DA PCM format, tri-state MOS outputs, GNU and Linux, circuit diagrams of master-slave edge-triggered D flip-flops, the Sieve of Eratosthenes, redundancy in the design of UPC codes (and their detailed decoding algorithm), the vector graphics display screens at SAGE, the Memex, the ALGOL committees, binary-coded decimal, IBM extended ASCII code pages, telegraph relays, the RTF format, Siskel and Ebert movie ratings, LZW compression in the GIF format, the number of electrons in a coulomb, Sutherland's Sketchpad, the process by which Louis Braille went blind, the number of fingers usually possessed by cartoon characters, the comedic etymology of "WYSIWYG", the number of pins on the 8087, the IEEE-754 floating point format, DX-encoding on 35-mm film canisters, Grace Hopper's career trajectory, the frequencies used by 300-baud FSK modems, the pricing of 10-gauge hookup wire at Radio Shack, George Boole's ancestry, ANSI cursor-addressing escape sequences, the 8080 instruction set and assembly syntax, Longfellow's poem about Paul Revere, Morse code, Radio Shack relay part numbers, NTSC sync pulse voltage levels, logographic "contractions" in Braille, the rise of the IDE starting from Turbo Pascal, Altair BASIC, the hardware floating-point encodings of the IBM 704, examples of Ohm's law, the failed GUI

VisiOn, MIDI sound cards and the structure of MIDI commands, transcendental function approximation using Taylor expansions, and so on.

In short, it goes into enough detail that you can actually learn things from it — it's not at the Wired-article level of pseudo-informative flimflam — but it's far from presenting a complete design for a working computer, much less a healthy software ecosystem.

The Book-In-Itself will have to omit most of the flashy, entertaining historical detail, but will include a complete system.

Differences from the Thwaites Toaster Project

Thomas Thwaites mined mica and metal ore from mines and plastic from garbage dumps; he made a wooden mold to shape the plastic; he smelted the metal; he refused to travel by air or use CNC mills. After nine months of work and US$1800 of expenses, he had an nearly functional toaster. Unfortunately, aside from looking terrible, it burned out after a few seconds of operation, because his metallurgy wasn't up to snuff when it came to making oxidation-resistant alloys.

Thwaites complicated his life significantly by deciding to use steel and plastic for his toaster. His choice of Constantan for the heating elements was probably unavoidable, but I suspect that making most of the toaster from copper would have worked significantly better. (To be fair, his mentor Professor Cilliers informed him of this at the beginning of the project. Steel and plastic were a hair-shirt for Thwaites.) He made his life harder still by insisting on using the obsolete bloomery process to smelt the iron, then lowering its carbon content by remelting it in his microwave.

The Book-In-Itself starts at a layer above where the Toaster Project ended: it presupposes the capacity to make complex physical artifacts, including megahertz-class logic circuits with tens of thousands, if not millions, of components. However, given that capacity, the Book-In-Itself is intended as a complete design that can execute *itself*, rather than just a story of a personal quest, perhaps a story that could inspire another person to undertake their own quest.

Moreover, a possible extension of the software core of the Book-In- Itself is an automated manufacturing module, capable of fabricating the hardware artifacts automatically, given suitable materials. This may or may not be included, depending on time and experiences. It would likely enable the Book-In-Itself to fabricate not only more instances of itself, but also to fabricate toasters — perhaps a *reductio ad absurdum* of Lincoln's axe-sharpening dictum.

Differences from the Dave Gingery series

Dave Gingery, and later his family, have published a series of books on how to build a metalworking shop from scrap materials, including techniques like hand scraping flat surfaces, sand casting of metals in a charcoal foundry, and lathe turning.

The books are unreadable without an existing background in the techniques they discuss; if you don't know the names of the parts of a lathe, you aren't going to get very far in understanding Gingery's explanation of how to build one.

Unlike most of the other similar projects, the results in the Gingery

books have been successfully replicated by several people, some of whom have documented their experiences on YouTube.

As with the Thwaites project, the Gingery series stops at a level of abstraction below where the Book-In-Itself begins, but Gingery succeeded in producing competent industrial machinery — not quite from scratch, as Thwaites aspired to, but from scrap materials. (Gingery had the advantage of being an experienced machinist rather than a 24-year-old second-year postgraduate design student.) But the relationship is rather similar: the Book-In-Itself cannot be built without the ability to construct complex artifacts, going even beyond what the Gingery books can produce, but perhaps it could include software modules for producing machine-shop equipment, given access to the right materials.

Differences from the Primitive Technology series

An anonymous or pseudonymous man in the rain forest in Australia is undertaking an even more ambitious project than Thwaites' Toaster: he is constructing a series of artifacts made entirely with materials and tools available in nature. The process is meticulously documented in a carefully edited YouTube video series and a blog. So far, he has created several stone axes, a stone adze, a ceramic-firing kiln, some ceramic pots, a ceramic tiled roof, two houses (one with a hypocaust), a woodshed, some wicker baskets, bow and push drills, a bow and arrow, charcoal, and a fenced sweet potato patch. He aspires to smelt iron. So far he has not mentioned any plans for electrical appliances.

This project consists of the levels of abstraction underpinning the Gingery project, and points out one of the reasons the Thwaites project was so difficult: Gingery used existing industrial materials and tools, while Thwaites attempted to abjure them. Within Gingery's constraints, a toaster would be entirely feasible — perhaps a project of a day or two, maybe a week, certainly not nine months. But once you're mixing your own Constantan, you're playing at a much higher level of difficulty.

However, the Primitive Technology series has a purity that parallels the purity of the Book-In-Itself: its author apparently begins with no tools, not so much as a steel pocketknife, and thus begins his project by banging rocks together to get split-cobble cutting edges. Similarly, the Book-In-Itself depends — in theory — on no hardware or software outside of the book itself; once you have the capability to make machines of almost any kind with tens or hundreds of thousands of parts, you need no further information-processing facilities to bring up a complete computing environment.

There's a crucial distinction here: the Book-In-Itself will be "self-sustaining", in the sense that it can easily replicate itself, and modified versions of itself, once it's running; but only optionally will it be "self-bootstrapping" in the way the Primitive Technology series is — that it's easy to get it running starting from a minimal basis. A self-sustaining system is written in itself; a self- bootstrapping system is written in something that already exists.

There are two basic approaches to adding bootstrapping to a self-sustaining system: you can write two versions of the bootstrapping core, one implemented in the system's own language and another written in some already-implemented language; or you

can write the bootstrapping core in the intersection between the language implemented by the system and a language implemented by something else, which may be a subset, a superset, or an extended subset of what the system implements. In either case, by positing different base layers to bootstrap from, you can get very different results

In the more fundamental layers explored by Gingery, Thwaites, and the Primitive Technology series, a self-sustaining system might consist of a lathe; a small blast furnace, like the kind used in China in late antiquity (around the Qin era); a charcoal-generating kiln employing metal cans; a woodland, harvested using steel axes; a smithy, with its hammer, tongs, anvil, and hearth; a concrete grindstone, mounted on bearings cut on the lathe; a ceramic-firing kiln for making refractory bricks; a cement kiln for making new grindstones; and mines for fire clay, quartz sand, iron ore, and limestone. But bootstrapping that system from stone, mud, and trees is likely to involve additional tools and materials, such as stone axes and perhaps facilities for smelting tin and copper to make bronze tools.

Differences from Smalltalk-78

Differences from Project Oberon

Differences from Plan 9

Differences from the VPRI STEPS program

The STEPS target of 20,000 lines of code is about 300 pages.

Differences from the biological cell

Topics

- Independence (p. 3520) (63 notes)
- Self-replication (p. 3703) (24 notes)
- BubbleOS (p. 3352) (17 notes)
- Hypertext (p. 3512) (13 notes)
- Bootstrapping (p. 3348) (12 notes)
- Reproducibility (p. 3682) (3 notes)

Cold plasma oxidation

Kragen Javier Sitaker, 2019-05-01 (updated 2019-08-21) (7 minutes)

You should be able to oxidize things that are difficult to oxidize using a rapidly cooled air plasma, and this should be feasible even without refractory electrodes and at atmospheric pressure.

The basic principle

If you convert a gas into a hot electric plasma between two electrodes, then cool the plasma rapidly to the gas's original temperature, the cooled plasma will still be different from the original gas in several ways: its molecule energies will still be far from the Maxwell–Boltzmann distribution, as it will still have a significant number of very hot molecules and free electrons (this is called an "anisothermal plasma"); many of the molecules will still be ionized; many of the non-ionized molecules will have excited electrons; and, if it's not a noble gas, many of the molecules will have been ripped apart, and some will have reformed in new conformations. In particular, if the source contained O_2, the resulting gas will also contain O and O_3, which are very strong oxidizers, and if it contained O_2 and N_2, it will also contain a variety of nitrogen oxides, some of which are also very strong oxidizers.

Wikipedia's nonthermal plasma article tells me that the oxygen atoms have a lifetime of about 14 μs.

Applications

Directing a stream of air, converted to this cool plasma, against a material that should have an excellent chance of oxidizing it further, if it's not already fully oxidized with oxygen or something stronger, without necessarily setting it on fire or even heating it much, and volatilizing some of the oxide, which could be useful for a variety of purposes:

- removing organic contaminants from oxide surfaces, such as calcite, clay, concrete, glass, sapphire, or viridian, including the glass, sapphire, and viridian passivating layers formed naturally on silicon, aluminum, chromium, and stainless steel surfaces in air;
- metal passivation, including selective thickening of passivation layers, including on iron;
- selective functionalization of organic surfaces;
- selective production of oxidation products on organic surfaces — in some cases, for example, this can be used to make a dark mark, making the device a printer;
- low-temperature cutting of materials whose oxides are volatile, including any organic material (other than teflon), graphite, diamond, zinc, and possibly even silicon carbide, molybdenum, iron, and steel (certainly at higher temperatures where the oxide will melt and can be blown off);
- acquiring trace amounts of volatilized oxides and nitrides for spectroscopic analysis;
- gasifying an oxidizable blowing agent from a matrix without heating it excessively, producing a foam, as in the traditional means of

producing firebrick by burning organic material out of the pores of a matrix;
• sterilizing drinking water or solid objects, especially solid objects that would be damaged by high temperatures, although maybe you should see if an ordinary ozone generator without plasma jets would be adequate (I see that this is currently under investigation for food processing);
• selective oxidative polymerization of, for example, linseed oil, tung oil, or stand oil, for example for 3-D printing, for 2-D printing with the polymer as a binder, or for depositing resists to guide later etching — and polymerizing almost any energetically favorable monomer should be possible at the right temperature, perhaps including polymerizing aqueous silicic acid into silica gel, as plasma polymerization is already used to form films of polystyrene, polyethylene, and poly(methyl methacrylate), among others, though normally from low-pressure gas-phase monomers;
• curing linseed-oil or stand-oil finishes very quickly;
• very rapid selective polymerization of cyanoacrylate by selectively adding hydroxyl radicals, although maybe ordinary water vapor is a better way to do this;
• oxidizing hazardous organic chemicals such as mustard gas, PCBs, benz(a)pyrene, and possibly-unknown mixtures into tamer and simpler oxides that can be more easily disposed of;
• cutting composites made of oxide grains cemented together with a carbon or organic binder.

Possible applications of non-air plasma ingredients include surface-nitriding metals and selectively "salt-glazing" silicate ceramics with a part-sodium plasma, or superpassivating surfaces that need to withstand exposure to strong oxidants by using a part-fluorine plasma. If the sodium source is chlorine-free (for example, NaOH or $NaNO_3$), the chlorine and HCl emissions that plague salt-glazing will not occur.

Directing the plasma

If the stream is sufficiently cool, its rate of erosion of already-oxidized materials such as teflon, soda-lime glass, iron oxides, quartz, viridian, sapphire, zircon, or zirconia should be relatively low, so it might be feasible to use a nozzle made from these materials to direct it onto the workpiece without eroding the nozzle too rapidly.

Cooling the plasma

Rapidly cooling of the plasma that has just passed through the arc can be effected by misting water into the plasma. The water will flash to steam, some of which will itself ionize and dissociate, contributing further reactive oxygen species to the mix.

Alternatively, you might be able to use corona discharge to ionize enough of the air to be useful, without ever heating it to arcing temperatures — maybe like a garden-variety ozone generator with more concentrated output and maybe gold-plated or carbon points for longer life.

Liquid electrodes

Optionally, the electrodes themselves can be covered with a

constantly replenished liquid electrolyte, such as water including a substantial mixture of NaOH or KOH, or molten $NaNO_3$ or KNO_3. This avoids the need for electrodes of refractory materials such a graphite, silicon carbide, tungsten, or hafnium. Some of the electrolyte will find its way into the arc, which may be preferable to the evaporation from solid refractory electrodes under some circumstances.

(Why don't potters just use chlorine-free sources for salt-glazing in a kiln? NaOH and $NaNO_3$ convert to NaO at high temperatures, which doesn't boil until 1950°, much higher than NaCl's 1413°, so a pottery kiln at 1100°–1200° has a hard time volatilizing the sodium from NaOH or $NaNO_3$ without an admixture of the lower-boiling NaCl, while the arc should have no trouble. However, they *do* use $NaCO_3$ and $NaHCO_3$ for "soda firing", sprayed into the kiln during firing, so I don't know.)

Hazardous emissions

However, if made from air, the cooled plasma inevitably contains sufficient ROSes and nitrogen oxides to be hazardous; the resulting gas needs to be thoroughly reduced unless you're doing this at a very small scale somewhere adequately ventilated. The standard approach to this problem is a platinum catalytic converter, but maybe bubbling the gas through consumable linseed oil or passing it over a consumable bed of sulfur, phosphorus, powdered lanthanoids, or really any easily oxidized chemical would be adequate.

Topics

What would a basic income guarantee for Argentina cost?

Kragen Javier Sitaker, 2014-04-24 (7 minutes)

What would a basic income guarantee for Argentina cost?

Last year I calculated that a person here could get adequate nutrition for some AR$3 per day, which is probably more like AR$5 now. Housing in the capital costs more; I'm paying AR$1500 per month for my one-bedroom apartment, but an apartment this size could host four people if they got along really well; living in it alone is a "luxury", which I put in quotes because I fucking hate living alone.

I did spend a week and a half late last year in an apartment here in the capital that was AR$300 per person, but the roof leaked badly enough that mushrooms were growing from the ceiling, there were crack dealers downstairs, and getting along with the other roommates and other people who had claims on the place proved impossible — not just for me, but for them as well.

On the other hand, just outside of the capital city, housing costs are a little lower. So the AR$400 or AR$500 per person suggested by the price of my apartment might actually be a little high. Let's say AR$300 per person, or AR$10 per day per person.

Beyond living space and food, what else do you need to survive? Utilities here are inexpensive; here's what I'm paying per month:

	AR$/month
building management services ("expensas")	200
electricity	17
ABL	27
telephone and internet	162
cellphone, approximately	50
gas	22
water	10
total	488

ABL is a set of city taxes known as "ABL" ("Impuesto Inmobiliario, Alumbrado Barrido, Limpieza, Mantenimiento, y Conservación de Sumideros"). The electricity and water bills come every two months, so the amount on the bill is twice what I've shown above; if someone pays $20 on their water bill, they're paying $10 per month.

The cellphone above is individual, but the other items would be shared with other people if I weren't living alone; they might be a little more expensive with more people (more gas for hot showers, say), but not much. Let's say a bit over 20% with two roommates, plus two more cellphones, rounding off at AR$700 per month, AR$233 per person per month.

In a city, transportation is a necessity rather than a luxury; typical is AR$1.70 for a bus ride twice a day per person, AR$102 per month. You may be able to get by with a bicycle, and a bicycle may actually work better at some times of day, but it's not an option for everyone

(like me, when my knee started to hurt a couple of years ago, although it's better now) and the bicycle is not without its own expenses; and transportation is more expensive for some people, particularly once you stray outside the capital; so I'm going to use AR$102 as my estimate of the basic cost of transportation.

That is, the basic expenses come to roughly this:

	AR$ per person per month
food	150
housing	300
utilities	233
transportation	102
total	785

There are people who live on less than AR$800 per month, but it's difficult and risky; there are always unplanned expenses. For example, I was very grateful to have AR$30 for antibiotics earlier this week when I came down with an unplanned kidney infection. A few weeks ago, I wouldn't have.

On the other hand, it seems like an AR$800/month/person basic income guarantee would be enough to nearly eliminate material insecurity. Right now that's about US$95, or US$150 at the official exchange rate. Per year, it would be about US$1800 per person, using the official rate. Argentina's per-capita GDP is about US$10600 nominal, so this is about 17% of GDP. Current population is about 41 million people, and the official government budget (not counting the difference between the official and unofficial exchange rates, which amounts to a heavy export tariff on legal exports) is about US$113 billion per year, US$2800 per person per year, or US$233 per person per month.

Providing a universal basic income, then, would cost about 64% of the current government budget, 17% of the total GDP. But it would subsume certain existing government programs --- for example, there's already a limited basic-income program called the *Asignación Universal por Hijo*, which only applies to minor children, only to 30% of them, and provides much less than the AR$800 I'm arguing is necessary.

I think the US$40B budget of ANSeS, the National Social Security Administration, is a good estimate of the size of the government programs that would be replaced; ANSeS's budget includes the AUH, the pensions of basically everybody (since the 2008 nationalization of the private pension funds), maternity leave, and so on. There might be some programs within the ambit of ANSeS that would not be replaced by a basic income, but there are also housing and healthcare subsidy programs which could possibly be replaced with a basic income which are not part of ANSeS.

So if we replaced an ANSeS-sized chunk of the budget with a basic-income program that paid US$1800 per person per year, and whose administrative costs were an additional 10%, we'd add only US$42 billion per year to the national budget — US$1010 per capita, a 37% increase from its current level:

	US$/year/person
basic income	1800

total expense of basic income guarantee program	1980
cost of ANSeS	-970
net tax increase from replacing ANSeS with BIG	1010
current total national government revenues	2700

On the other hand, this might be unrealistic; in particular, I think most pensioners need quite a bit more than AR$800 per month, something like twice that, and they're not as flexible as younger people — working for extra money, or moving in with roommates, is very difficult for many of them, and they have medical expenses younger people don't. So you might not be able to replace pensions entirely with a basic-income guarantee. You might need to allocate an extra AR$1000 per month to each of the 5 million retirees, an additional AR$5B or US$1B per month, US$12B/year; and there might be a need for other programs along these lines.

But it does seem like the magnitude of the expenditure would be manageable within the context of a modern society like Argentina, if nontrivial. Politically, however, it seems impossible at the moment, since even the modest AUH attracts heavy criticism, along the usual welfare-queen lines.

Topics

Friction-cutting plastic

Kragen Javier Sitaker, 2019-02-25 (8 minutes)

I just tried friction-cutting some PVC pipe with cotton string. A thicker braided string worked very reliably, but required a lot of work to melt the large amount of plastic, also leaving a rougher cut with bigger burrs. A thin twisted cotton string worked with a lot less effort, leaving a cleaner cut, but also broke several times.

I was thinking that you could probably do a kind of low-energy friction bandsaw/jigsaw for thermoplastics, as a kind of low-budget substitute for laser cutting and waterjet cutting, at least those that soften enough at a low enough temperature; cellulose string works up to about 200°, while steel wire should work well at higher temperatures. Nylon reputedly also works for PVC, since its softening temperature is so much higher than PVC's; cellulose reportedly works for nylon itself, while other sources claim cotton won't cut PVC. I think UHMWPE cord will not work at all, though I haven't tried it.

Ideally you'd have much, much less mass attached to the string than my hands and arms, so that when it encounters extra resistance it just stops instead of breaking the string, and then maybe backs off a bit to extricate the string from the melt before trying again.

In theory the energy for making such a cut is strictly proportional to the volume of the cut, so as the cut becomes thinner, the energy to make it decreases proportional to the width. However, the force available decreases proportional to the cross-sectional area of the cord, which (if it stays round rather than becoming a strip like an actual bandsaw or even jigsaw) is proportional to the *square* of the width; so applying the same amount of power per unit material requires increasing the speed in inverse proportion to the thread thickness.

I was thinking that for straight cuts, slots, and non-through cuts, you might be able to use a paper or card disc in a sort of angle-grinder configuration — clamped somewhat close to the edge between two thicker discs to provide it with a sufficiently stiff forward force, to keep the paper from buckling too much. In this case, at a high enough speed (for a small enough disc) the centrifugal force of the paper itself would provide most of the pressure against the material.

Plumbers say PVC is rated for up to 154° F continuous service, which works out to be almost 68° in modern units and softens at 250° F (121°), but doesn't melt until 360° F (182°); Wikipedia claims the Tg is 82° and the melting point can be as high as 260°. Probably somewhere in the middle is when it gets soft enough to be easily moved by the string.

The variability in PVC's heat-stability is partly because it needs heat stabilizers to work at all; otherwise it starts to release HCl at only 70°, according to Wikipedia.

According to the plumbers' discussion thread, its specific heat is 0.25 "Cal/°C/gm", which I think means a quarter of that of water, working out to 1.05 J/K/g in SI units, pretty close to the 0.9 kJ/kg/K in Wikipedia. I think we're talking about a ΔT of around 150 K, so around 150 J/g; due to PVC's density of about 1.4 g/cc, this is roughly 210 J/mℓ. The cotton string I was using might have been 0.8

mm wide, the pipe was about 1.5 mm thick and about 20 mm across, and it took me less than a minute to cut through it at about 1 m/s of string motion at close to its breaking tension, which is about 30 N — about 30 W and about 30 seconds, so about 900 J. The annulus of $\pi((10\text{ mm})^2 - (8.5\text{ mm})^2) \approx 90\text{ mm}^2$ had a volume of about 70 mm³ and thus weighed about 70 mg or 0.07 g. So the heat needed to soften it adequately should be about 15 J, about 60× smaller than what was being applied. (More experienced people report shorter times to cut even thicker pipe.)

Presumably the extra heat was lost to inconsistent pressure, conduction into the plastic with its ≈0.2 W/m/K conductivity (in part because the contact area was large — cutting worked much faster once I had gotten through the first wall), heating of the air, and heating of the string. So probably using much higher power would have been more efficient; perhaps instead of 1.7% efficient we could reach 5% or 10%.

To be concrete, maybe 10 m/s would have been a better speed for the 800-µm string, cutting through the pipe in more like 3 seconds, or maybe 1 second if the hoped-for efficiency advantages materialize; if we were using an 80-µm cellulose string, we might prefer to run it at 100 m/s and only about 2–3 N; this would theoretically melt the same amount of material per second, but in an 80-µm-wide path, so it would use 0.3 seconds or 0.1 seconds to cut through the same pipe. This works out to about "500–1500 inches per minute" in archaic units, which is a quite fast cutting speed for panel-cutting machines like laser cutters, CNC plasma torches, CNC oxy-acetylene torches, and waterjet cutters.

How much efficiency improvement could we expect from cutting faster? A W is a J/s, so we have 0.2 J/s/m/K of conductivity. Presumably if we put the heat in in one-tenth as many seconds, we'll have the heat spread out into the same set of nested cylinders it would have had if we had been cutting at the same speed in a material with one-tenth the thermal conductivity, 0.02 W/m/K. I don't know how to estimate how much this improves your efficiency.

Is it plausible to run an 80-µm cellulose thread at 100 m/s (220 mph in archaic units) and 2–3 N (7–11 ounces)? You could maybe use a 10-meter long loop of it at 10 Hz; it would be 50 mm³ and thus about 50 mg. 3 N could accelerate or decelerate it at 60000 m/s/s, so it could start or stop in 1.7 milliseconds at the edge of breaking. That sounds fast, but it's 85 mm of travel at that speed, which is how much stretch would need to happen.

So that might be too much, but a slightly less ambitious goal is probably feasible, though.

Doing this in Styrofoam or other foams might be even more interesting, since there's so much less mass, its thermal conductivity is so much lower, and the quality of a melted cut through Styrofoam is so much better than that of a cold cut, due to sealing off the cells.

You might think polyethylene would be much easier to cut this way, given its lower melting point, but in practice it's more difficult; I think this is because it's softer and has a higher specific heat (2.3 J/g/K, 2.5 times higher) and higher thermal conductivity (0.3–0.5 W/m/K rather than 0.2), but also because it lubricates much better, so much higher normal forces are needed to get the same frictional force. Presumably stepping up the power as outlined above

would be sufficient to solve the problem.

Topics

Methods of pumping ice-vest coolant silently

Kragen Javier Sitaker, 2019-09-28 (12 minutes)

I was thinking about Ice pants (p. 298) and related ice vest stuff on the bus the other day. One of the potential problems with the design is that it needs a pump; pumps are usually noisy and unreliable, in part because they contain a lot of moving parts. A pump that avoided these problems would be desirable for an everyday-wear ice vest, which after all only needs a small total flow rate. (To remove 200 watts of heat by warming water from 0° to 20°, you need about 2.4 milliliters per second: 200 W / (20 kcal/ℓ).)

Magnetohydrodynamic motors

Magnetohydrodynamic motors are pumps that solve these problems by passing a current through the liquid to be pumped at right angles to an applied magnetic field. The magnetic field produced by the current creates a pressure gradient in the fluid, which propels it silently through the passage. Jacque Fresco famously envisioned silent ships propelled by this means, although this might require unreasonably large fuel cells if you aren't going to generate the electricity with a noisy heat engine; perhaps some navy has tried this approach for silent nuclear submarine propulsion. In, I think, 2018, the YouTube channel Cody'sLab demonstrated the use of such a pump for pumping mercury between reaction vessels for the chloralkali process.

The electrolysis problem

However, electrolysis poses a potential difficulty for magnetohydrodynamic motors in water and other ionic conductors: the electrodes immersed in the liquid will tend to produce bubbles of oxygen and hydrogen. This problem can be reduced to some extent by periodically reversing the direction of the current (and of the applied magnetic field, so as to not reverse the flow direction) but not, I think, eliminated, in a classical MHD motor. In most circumstances, this is not a problem for MHD motors, though measures must be taken to prevent anodic dissolution, such as using carbon electrodes.

Perhaps the lore of electroplating has useful information here, because bubble buildup is a problem for electroplating — plating doesn't happen in regions of the cathode covered by bubbles. But anode bubbles are no problem for electroplating.

The buildup of explosive gases in the tubing of an ice vest could not only impede liquid flow but potentially even pose a hazard of rupture and fire.

Brine coolant

The coolant in the ice vest needs to have a melting point below that of pure water so that it can pass unimpeded through the ice pack without risk of freezing, and it needs to prevent bacterial and fungal growth inside the tubing, since those can also produce gases and

potentially corrode the tubing. One possible solution to this problem is to use water sufficiently salty that almost no life forms can survive in it, and this has the advantage of greatly increasing the water's conductivity. This probably requires keeping the water from contact with glass or metal to prevent heavy corrosion, though most common plastics would be fine.

Eddy-current-driven pumping

A possible electrolysis-free way to do a magnetohydrodynamic motor to pump an electrolyte is to use eddy (electrical) currents rather than linear (electrical) currents through the electrolyte, thus eliminating the need for electrode contact — in essence, a squirt coilgun rather than a squirt railgun. This is the same approach used in squirrel-cage AC motors, coilguns, and some magnetic-levitation systems. I think eddy currents should provoke no ionic concentration gradients at all.

One way to do this would be to wrap many coils around an electrolyte-filled tube and energize the coils in the same current direction in sequence, moving the magnetic field along the length of the tube. This will induce eddy currents inside the electrolyte around the axis of the tube opposing the direction of the coil current, thus producing an opposing magnetic field, growing with a time constant related to the inductance of the single "turn" around the axis and the resistance of the electrolyte; when the coil magnetic field moves to the next coil, the thus-magnetized electrolyte will be attracted to it, with its magnetic field potentially diminishing with the same time constant. Energizing the coil behind it with the opposing direction of current will also help.

The movement speed of the applied magnetic field needs to bear a certain relation to the movement speed of the liquid and the RL time constant of the induced eddy current. If the field moves too fast (e.g., 100 000 m/s), the eddy currents induced in the liquid will not have time to build up to a level where they can produce an appreciable magnetic force; indeed, the magnetic field will be confined to the skin of the electrolyte by the skin effect. (Am I misunderstanding this? Perhaps large eddy currents occur almost instantly and the magnetic field penetrates more deeply as they begin to die away?) On the other hand, if the field moves too slowly (e.g., 1 μm/s), the eddy currents will have decayed to very low levels before the applied magnetic field moves to the next coil. Somewhere in between is a sweet spot.

It may be more efficient to use a small tubes to get a more concentrated magnetic field or large tubes to get less viscous losses, to use concentric tubes separating liquid layers to force the eddy currents in the outer parts of the liquid to enclose a large area of magnetic flux, to use an annular tube with just an air space in the middle for the same reason, and perhaps even to replace that air space with something like ferrite — although perhaps that would result in increasing the force on the ferrite rather than the liquid.

A centrifugal magnetic stirrer

As an alternative to magnetohydrodynamic motors, you could drive a solid impeller.

Chemistry labs nowadays commonly use magnetic stirrers, often

built into hotplates. These apply a magnetic field rotating at a few Hz to the reaction vessel, typically an Erlenmeyer flask; a magnet moving freely at the bottom of the vessel is free to rotate to align itself with the rotating magnetic field. (The magnet must be encapsulated in something; I suspect teflon.) This usually produces a substantial amount of noise, but much less than a conventional motor driving a gearbox which turns an impeller on a bearing-mounted shaft that passes through a seal. And, although it has a moving part, that moving part has tolerances measured in centimeters, so wear is not much of a problem.

You could apply the same approach to the ice vest, applying a rotating magnetic field to rotate a solid impeller entirely contained inside the liquid chamber; the low speeds involved (< 60 rpm) suggest the use of a permanent-magnet-based impeller rather than something like a squirrel cage. To prevent the clattering noises common with chemistry-lab magnetic stirrers, the impeller could be circular, like a centrifugal blower, rather than oblong; a Tesla-turbine design might work. By giving it the same overall density as the liquid, balancing the extra density of the permanent magnets by including air bubbles, you could avoid stresses from impacts and changes in the direction of gravity.

The magnets themselves might be sufficient to prevent its axis of rotation from deviating too far from the axis of the pumping chamber without requiring a shaft, and to prevent it from translating axially until it hit a wall of the chamber; thus it could normally operate without any solid-to-solid contact and thus without any wear. Whether the magnetic fields were radial (as in a squirrel-cage motor or ordinary BLDC motor) or axial (as in a pancake motor), active position control may be necessary to prevent instability that would cause it to drift closer to some of the electromagnets until it hit a wall. Perhaps fluid-bearing effects could be adequate to prevent this.

In some sense the ideal would be to have the highest MMF density and thus flux density at the center of the pumping chamber rather than near its walls, so that the magnet or magnets in the impeller would tend to drift toward the center rather than toward the walls. Given that the coils applying the magnetic field are necessarily in the walls, I'm not sure if this is feasible, but it might be a good start to wrap the coils around the center of the chamber (e.g., in the xz plane and the yz plane, if the z-axis is the rotation axis and the origin is the center of the chamber — not parallel to those planes, actually in them) rather than around cores outside its walls.

The use of a solid impeller would obviate the necessity for using an electrolyte, allowing the use of less corrosive liquid coolants (see Coolants (p. 3235)) such as propylene glycol or a water–(propylene glycol) mixture. This in turn could perhaps enable the use of a bare metal impeller without excessive corrosion, though I would think that plastics such as PET would still be a better choice.

If the pumping chamber were spherical rather than cylindrical, the impeller could rotate freely in it when the pump was being rotated in space, thus avoiding collisions with the walls, even if the impeller itself were still cylindrical (though it could be spherical as well, like those spherical compasses people mounted on their car dashboards in the 1980s). The applied magnetic fields would eventually bring it back into alignment for proper pumping action.

Pump energy use

Another reason to want to use a tiny, low-power pump is to reduce the amount of heat added to the water from viscous friction. If you're using a 10-watt pump that's 50% efficient, it's adding 5 watts of energy to the water, and so the water will heat up at 5 watts. While this is small compared to the total cooling load of a one-person ice vest, it is not insignificant --- someone relaxing at 70 watts might find that the reduction in "battery life" from 5 hours 17 minutes to 4 hours 56 minutes (on 4 kg of ice) was a significant loss.

An even bigger issue, though, is the weight of the battery used to power the pump. 10 watts over 4 hours is 144 kJ, which is about 250 g of lithium-ion batteries. Reducing the pump power usage down below the one-watt level, if it's possible, would lighten the battery weight of the suit dramatically.

Pulsing

As described in Intermittent fluid flow for heat transport (p. 521), a pulsing flow is more effective than constant flow for this sort of thing, because it distributes the coolth more evenly. One way to achieve this is to use a pump with several times the needed flow rate, but only run it a small fraction of the time. This results in higher viscous losses (for a given tube diameter) but may be worth it.

Topics

- Electronics (p. 3430) (138 notes)
- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Mechanical things (p. 3569) (45 notes)
- Cooling (p. 3393) (15 notes)
- Water (p. 3773) (13 notes)
- Safety (p. 3693) (9 notes)
- Electrolysis (p. 3429) (7 notes)
- Ice vests (p. 3513) (3 notes)
- Electrohydrodynamic motors

String tuple encoding

Kragen Javier Sitaker, 2017-04-28 (2 minutes)

A number of algorithms and data structures depend on the lexicographical ordering of bytestrings; for example, tries, LevelDB, `LC_ALL=C /bin/sort`, the American Flag Sort, and suffix array construction algorithms. These are often asymptotically higher in performance than alternatives based entirely on item-to-item comparisons, and often have better constant factors as well. So it can be useful to find bytestring encodings of different abstract data types that preserves those data types' natural orderings.

There is existing work on this. Dean Landolt's `bytewise` is a library for encoding arbitary JS data structures as byte strings for just such purposes. UTF-8 is an algorithm for transforming a sequence of Unicode codepoints into a sequence of bytes or vice versa, and it preserves lexicographical ordering in precisely the way I'm talking about here.

This note, however, is about a specific subproblem: the problem of encoding a tuple of bytestrings as a bytestring while preserving lexicographical order. That is, if the alphabet of bytes is `Σ`, we want an injective mapping `Σ** → Σ*` that is a homomorphism when we consider the elements of `Σ**` and `Σ*` as elements of a totally ordered set whose order is defined lexicographically.

Bytestuffing

The approach taken by `bytewise` for arrays is to encode them as an array type byte 0xa0, then each item followed by a NUL byte 0x00, then a final terminating NUL byte 0x00. This is clearly correct if the encoded items in the array cannot contain NUL bytes, but of course they can if they themselves are arrays (or, as it happens, numbers, buffers, or some other types). So `bytewise` bytestuffs the item encodings as follows: an embedded 0x00 as 0x01 0x01 and an embedded 0x01 as 0x01 0x02, and symmetrically, but for other reasons, 0xff and 0xfe are bytestuffed to 0xfe 0xfe and 0xfe 0xfd. This correctly preserves the lexicographical ordering.

The example currently given in the README is that `new Buffer('ff00fe01', 'hex')` encodes as (hex) 60ff00fe01, `'foo'` encodes as (ASCII) 'pfoo', `['foo']` encodes as (C) `"\xa0pfoo\0\0"`, and `[ new Buffer('ff00fe01', 'hex') ]` encodes as (hex) a060fefe0101fefd01020000.

While this is correct, it has the disadvantage that, for a single level of bytestuffing, the worst-case encoded size is double the decoded size, and, as it happens, the encoding of a value needing bytestuffing will

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- Sorting (p. 3720) (8 notes)
- Bytestrings (p. 3357) (3 notes)

Lab power supply

Kragen Javier Sitaker, 2017-02-21 (updated 2018-06-18) (17 minutes)

I just made a 0–11.5V adjustable power supply from an ATX power supply in the Ekospace hacklab. Soldered the green wire (what is this called?) to ground so the power supply turns on automatically and cannibalized a SATA power connector to hook up the +12V yellow wire and ground to an emitter follower built out of a TO-220 D13007K NPN power transistor someone salvaged from a motherboard, three 2.2kΩ resistors (half-watt I think), and a 10kΩ potentiometer.

This works for spinning up a tiny motor I found in the parts bin, but it's not very efficient, and it's likely to burn up the transistor at some point, because whatever voltage doesn't appear on the output is dissipated across the transistor. The TO-220 could theoretically dissipate 80W if it were like screwed to a heatsink or something, but it's not. It could maybe dissipate like 20W, which at 12V would be just 1.7 amps, although it can probably handle its rated 8 amps at its max voltage, if not the full 18 available from the power supply in question.

It has the additional disadvantages that there's no display for the voltage or current (you can set the voltage by hooking up a voltmeter to the output while turning the knob) and the output voltage will fluctuate with the input voltage from the ATX power supply, which can vary.

So, what would a better solution look like? You could build a buck converter based on the AVR ATMega328 used in the Arduino. It has an internal 1.1V voltage reference. You could use one of its six PWM outputs (with 8 bits of precision you get 31.25kHz; I think you can get 62.5kHz if you accept 7 bits of precision) to control the buck converter. A couple of external resistors could form a voltage divider to scale down the buck converter's output voltage to the 1.1V range, which feeds into the ATMega328's 10-bit ADC. A transistor, inductor, diode, and capacitor would complete the buck converter. An additional low-value sense resistor could measure current on a second ADC channel (there's one ADC but 6 or 8 pins that it can multiplex between). Then the AVR could produce output displaying the voltage and current either on six seven-segment displays (13 digital GPIO pins in the usual multiplexing arrangement) or through a speaker on a second PWM channel. A third ADC channel could be used to read a potentiometer to control the output voltage.

How big do these external components need to be, and how much precision do we need?

Supposing arbitrarily that I were to use a similar ATX power supply capable of 18A on its 12V output, which works out to 216W (a bit over a quarter horsepower), it would be nice to be able to carry that 216W most of the way down the range, say down to 2V — which would mean 108 fucking amperes. This is a somewhat unreasonable amount of amperage, as common power transistors are typically in like the 1 to 8 amp range. A popular power transistor with somewhat more oomph is the Siliconix SiS410DN, which costs 94¢ from Digi-Key and handles 35 amps; then there's the Nexperia

PSMN4R0-40YS from 2010, which costs 88¢ and handles 100 amps. But I'm probably not going to salvage those from discarded Argentine electronics.

I can buy similar power MOSFETs here in town at G.M. Electrónica or SyC Electrónica, though, even if they are somewhat inferior and more expensive. They only carry International Rectifier parts (or STMicroelectronics versions of them) in that range, though; they both have, for example, the IRF540. SyC's price for the ST version is 60¢.

So let's say I use one of those. It turns on at 4V, so I don't need a gate drive level shifter. The STMicroelectronics version SyC sells for 60¢ handles 22 amps, which is more than the ATX power supply can deliver anyway (though maybe a capacitor on the input of the buck converter could help with that). We only really need (or can display on a 3-digit LED) about 100mV precision, so 128 duty cycles is probably enough, so we can probably use the 62.5kHz speed. How much energy do we need to store in the inductor at 62.5kHz?

That's 16 microseconds per cycle. I'm a little unclear on exactly how the math of buck converters works out but I am pretty sure that it will not involve storing more than 16 microseconds' worth of power in the inductor, which would be three or four millijoules, and I'm pretty sure it's okay for the inductor current to fluctuate by 10% or so, maybe a lot more. So if $\frac{1}{2}LI^2 = 4$ mJ and $I = 18$ A, then $L = 2{\cdot}4\text{ mJ} / (18\text{ A})^2 = 25\mu\text{H}$, which I would have thought a fairly small inductor but is apparently around the 75th percentile of modern inductors. Combining that with the high current requirement leaves very few options in Digi-Key's catalog.

One such is the Würth 7443643300, which goes for US$8.50 — 33µH, 30A, 2.4mΩ, self-resonant at 7MHz, ferrite, 28.5mm × 19.5mm × 18.5 mm, ferrite, saturating at only 11.5A (which seems like it could be a problem in this application!). It seems to be five turns of flat 3.8mm × 0.8mm copper tape!

So it's feasible but maybe more study of buck converter math would help me more. PDM might also help by reducing the cycle time to submicrosecond levels; ST's IRF540 has a turn-on delay plus rise time of 105ns, which means we can't go deep submicrosecond without losing efficiency, though Infineon's part may be a bit faster.

Horowitz & Hill draw the buck converter with a Schottky diode, which makes a certain amount of sense — 18 amps at an 0.7-volt voltage drop would be about 13W dissipated in the diode, and indeed most popular large-current diodes on Digi-Key are Schottky, like the VB30100S-E3/8W, which is 100V, 30A, in a TO-263AB surface-mount package with a cathode-body connection; this has the ordinary 300mV Schottky silicon voltage drop at ordinary currents, but at 10A it's already up over 500mV, and at 18A it's almost 700mV. At its rated maximum 30A it's 910mV.

Horowitz & Hill also end up using a 150µH inductor for their first example (5V, 500mA, 50kHz). Is that less of a pain in the ass?

A 220µH 700mA ferrite inductor costs 32¢ at Digi-Key; a 150µH 1A ferrite inductor is 56¢. It has axial leads and is 6.4 mm in diameter and 14 mm long. A 150µH 2.2A ferrite inductor (saturating at 1.8A) costs US$1.03 and is 12mm × 12mm, and at this point we're starting to get into lower frequencies, higher costs, and shielded construction. At 4A and 150µH we're getting into US$2.37 25mm iron toroids with

thick copper wire around them, and the price trend is clear, although at this point still only linear with current — this one holds 1.2 millijoules, 500µJ/$, while the 2.2A one was only 350µJ/$. Above 8A, prices start to climb proportional to energy and we're getting into big wirewound powdered iron cores and then laminated silicon steel.

Someone tried making an Arduino-driven buck converter like what I'm suggesting and discovered that since the IRF540 is an N-channel FET it's a pain to switch the high side with it. Also apparently buck converters need to react quickly to inductor saturation to prevent explosions. Some Croats did it successfully. They did use a humongous wire-wound toroid.

Man, I've really gotten hung up on the inductor, haven't I? It's just that I'm worried that a giant piece of shit like that could really ruin what would otherwise be a tiny, cheap, ferocious power supply.

The freewheeling diode itself might function as an adequate current sensing "resistor"; as mentioned above, a Schottky diode varies over the 300–900mV range on its way up to 30 amps. This is pretty dependent on the temperature, but if you can somehow correct for that, you should be able to measure the current within about 3% over a 10mA to 10,000mA range, no problem. I think the inductor-driven current through the freewheeling diode may generate a negative voltage with respect to ground, though, which is inconvenient for measurement.

Alternatively, you could have like a 50mΩ sense resistor on the ground side; at 18A this would be 900mV, and each millivolt change (about one count on the ADC) would be 20 mA. This introduces a little instability into the voltage regulation, although for many loads you could compensate for that adequately in software.

Scanning six 7-segment displays should be fairly trivial if you have 13 available GPIOs after the buck regulator PWM output and two ADCs.

To run the AVR itself off the 12V supply, if you're not using an entire Arduino, you could just use a 7805 regulator. An ATMega328 supposedly uses 12mA at 8MHz active, so probably 30mA at 20MHz (though another part of the datasheet says 12mA at 20MHz). If it's using ¼W or ½W, it isn't going to burn up the 7805 to be wasting another ⅜W or ⅝W. Or maybe you could use a resistor and a 5V 1W zener (SyC has them in stock for 11¢, although I have no idea how to salvage them).

Cheaper AVRs might work. The ATTiny5 costs 35¢, runs at 12MHz, has a four-channel 8-bit ADC, and has four I/O pins. The cheapest AVR with 16 or more I/O pins is the 76¢ ATTiny40; it has 4KiB of flash, 256 bytes of RAM, runs up to 12MHz, and has 12 channels for its 10-bit DAC.

I think probably the very next step is to hook up an AVR display to the existing linear power supply to merely passively measure its voltage. This involves minimally an Arduino, three current-limiting resistors (220Ω or 470Ω, say), three seven-segment displays, 10 GPIO pins to run them, and a voltage divider to get the voltage input down below 1.1V.

As a sub-step before that, I should see if I can run this calculator LCD off my Arduino. For that I won't even need resistors. But maybe I can buy some transistors too.

So, I salvaged (is that the word?) a four-digit seven-segment green

LED clock display from a Philips clock radio. It's an LTC-637D1G, which turns out to be a Lite-On product from 2000; I already reverse-engineered its pinout before googling up the datasheet, which pretty much confirms what I'd already figured out — it's a supremely shitty pinout, with the dubious grace of having only two cathodes instead of the expected four, so I can get by with two resistors instead of four. It had the benefit of lighting up visibly when probed with the multimeter in the hacklab on the diode-test setting. LED voltage measurements on the variable-voltage power supply show that the LEDs start to become visible at about 4V on a 220Ω resistor, at which point they themselves are dropping about 2 volts, leaving 2V for the resistor, so the current is about 9mA. Turning the power supply up to 11.5V illuminates the LED more brightly but doesn't burn it out; at this point it has about 3V across it, which exceeds the 2.6V max from the datasheet, though I guess that was at 20mA, and this is more like 38mA, which also exceeds the 25mA continuous current max in the datasheet.

The datasheet says it can handle 100mA peak currents, but that's not safe if they're running straight off an AVR pin. The AVR as a whole is only rated for 200mA on its Vcc and ground pins, and only up to 40mA per channel. And if I use the 220Ω resistors I now have soldered to the common cathodes, the max I'll get at 3V (5V minus the LED's 2V or more drop) dropped across the resistor is 14mA. On the plus side that means I can run two diodes off one anode pin at the same time, so I can scan across the active anode pins, toggling the cathode pins between tri-stated and low according to whether that "pixel" is supposed to be on or off.

Since this is a 24-hour clock display, the first digit is missing the segment that isn't displayed in either the digit 1 or 2. So it's probably better to use just the last three digits. These have only 11 anode lines between them; if all 11 have active data on them, I can manage a 9.1% duty cycle. This works out to an average bright current of 1.3 mA, which I think will be barely visible at all. This could be improved slightly to like 1.8mA average, but getting it up to near the 10mA average suggested in the datasheet would require 11 high-side drivers for the LED anodes, like bipolar transistors or something, since that would require the full 100mA peak current specified — twice that if both cathodes are active. So for now I think I'll give it a pass.

So the AVR power supply voltmeter needs 11 anode pins, 2 cathode pins, and one analog input pin, which means it can fit into much smaller AVRs than the ATMega328. The ATTiny40 with its 18 GPIO lines and 4K of Flash, for example, should be fine. It costs 76¢ and is 3.1mm × 3.1mm in a VQFN. (The 328's smallest package is 5.1mm × 5.1mm.)

Refreshing at 1kHz (again, as suggested in the display datasheet) would require iterating at 11kHz. At the AVR's internal RC oscillator speed of 8MHz, this gives us 727 clock cycles per display update — far more than necessary to respond to the timer interrupt. At an Arduino's 16MHz speed, we have twice as many.

I cobbled together a voltage divider out of some carbon-composition resistors found lying around; it turns out they are 140kΩ and 4.2kΩ, so the voltage scaling factor will be 4.2/144.2 = .0291, so 12 volts would be measured as 350 millivolts. However, the AVR's 1.1V internal bandgap voltage reference is specified to be

between 1.0 and 1.2 V, so that might digitize as anywhere from 298 counts to 358, and the count will change at least every 40 millivolts, which is about the right accuracy for an 0.1-volt-resolution digital voltmeter.

A thing I largely neglected above: a bench power supply needs current limiting. A purely linear approach to this is to use an LM317 (1.5A max) with ADJ connected to the load and a sense resistor between it and OUT — the LM317 will let the full voltage through until 1.25V is dropped over the sense resistor. 2Ω limits you to 625 mA, for example.

You could reasonably run a sense resistor like this in series with the usual voltage divider, limiting a weighted sum of current and voltage rather than either one alone. Or maybe you could use a couple of diodes to limit them separately with a single LM317: either a large enough output voltage or a large enough output current would pull down the ADJ pin through diodes from the ADJ pin to the end of the sense resistor and the middle of the voltage divider. This also gives you an extra 0.7V or so, so the ADJ pin won't get to 1.25V below output until the point that's pulling it down is 1.95V below.

The LM317 has internal thermal overload protection, a plus, but it needs 2.5 mA of output current to stay in regulation. One big disadvantage it has for this purpose is that it's not LDO — it has a 2.5 V dropout, so it can't deliver more than 9.5 V off a 12V ATX supply. If you keep the output resistor to ground under 500Ω then it'll always have enough bias current.

Topics

Jello printing

Kragen Javier Sitaker, 2016-12-14 (8 minutes)

I think jello normally has about the refractive index of water, but it's possible to change it from about 1.3 to about 1.7 by adding sugar, or lower it as low as 0.7 by adding alcohol; both will diffuse through the jello once it is set. To avoid chromatic diffraction patterns, the difference in optical path length should be over 10λ, or about 7μm for red light. This in turn suggests that a variation in refractive index from 1.3 to 1.4 would require layers of some 70μm or more in thickness. But of course that is easy to achieve; what is difficult to achieve is layers of *less* than about 100μm.

The proposal, in short, is to deposit jello with a 3-D printer in thin layers with continuously variable sugar or alcohol content, in order to approximate a desired optical transformation function. Other food gels such as agar, carrageenan, or konjac are also likely to work well, and have the advantage of avoiding ethical issues with cruelty to animals. As the jello is deposited, it will cool and gel, slowing but not eliminating the diffusion of the dopants.

Discontinuities in refractive index that scatter light in conventional optical systems, producing stray light that limits optical system performance. Dopant diffusion eliminates these discontinuities, replacing them with gradients, permitting order-of-magnitude improvements in optical system complexity.

Active closed-loop control of refractive index of deposited material permits the use of very small differences in refractive index to improve control of optical phase delay. Given a layer thickness of 200μm, a minimally acceptable phase delay error of 100nm would seem to require .05% error in the refractive index of the deposited material, for example, controlling whether the refractive index is 1.4000 or 1.4007. This turns out to be feasible with closed-loop control, as follows.

According to the International Scale of Refractive Indices, water's refractive index at 40% sugar is 1.3997; at 41% it is 1.4016, a difference of 0.0019 units of IOR over 1%. Ordinary lab refractometers provide a readout to 5 digits of precision. So this level of control amounts to controlling the sugar content of water to a precision of about 0.3%, or 0.3°Bx, which I think is feasible. Indeed, it should be feasible to control it to within the precision of the refractometer, 0.0001 units. This corresponds to a phase error of some 20nm, which is acceptable for even the most demanding optics. Some refractometers have 5× smaller errors than this, which requires wavelength calibration of their light source and temperature compensation of the sample.

Such closed-loop process control using process refractometers has long been common practice in industries that process sugar solutions.

It isn't necessary to control the IOR of the finished product to this level of precision in an absolute sense, because what matters for refraction is the *gradient* of IOR; it's just necessary to avoid random errors between passes and layers. This is fortunate, because the refractive index of the sugar-water system varies with temperature by about 0.0001 per kelvin.

It should be possible to achieve fine control of sugar concentration

by mixing two or more homogeneous solutions in varying proportions; for example, a mixture of 36%-sugar jello A and 42%-sugar jello B that is 50-50 would be 39% sugar, while if it's 55-45, it would be 38.7% sugar. In this way a 5% error in the mixing process amounts to only an 0.3% error in the final concentration.

Some other related solute systems have nonmonotonic refractive-index curves; the ethanol-water system, for example, reaches its maximum refractive index of about 1.3658 at about 80% ethanol, thereafter declining to 1.361. In the neighborhood of this local maximum, fairly large changes in the concentration of ethanol are needed to obtain small changes in refractive index; for example, IOR is 1.3654 at 72% ethanol, but 1.3657 at 76% ethanol. This improves the control over IOR that can be achieved despite errors in mixing.

Ethanol has the major disadvantage that it is a very small molecule, so diffusion and consequent degradation of the instrument is likely to be very fast. Instead, larger molecules than sucrose would be advantageous in order to delay diffusion further.

A perfectly flat "Fresnel" lens of jello with a thickness of about 200μm should be achievable with this approach. A simple waveplate pattern would provide an aspheric focusing lens, for example, although at the cost of dispersion and some stray light from the boundaries between the rings. A thicker lens, probably thicker than normal glass lenses, could reduce the number of rings (perhaps to 1) and thus eliminate the stray light. It should be feasible to fabricate an entire compound microscope or telescope in a single solid block in this way.

To prevent decay of the product (before it is destroyed naturally by diffusion), some kind of antibacterial and antifungal agent should be included. Ethanol at levels above about 15% is one possibility; sodium chloride is another; acids such as acetic acid or citric acid are yet another. Finally, a sufficiently high level of sugar would also prevent decay, but that level might be high enough to prevent gel formation.

It's probably desirable to encapsulate the final result in some kind of airtight container (sandwiched between glass plates, filling a transparent polyvinylidene-chloride balloon, etc.) to prevent evaporation from converting the result to a xerogel, which would likely impair its optical properties. Alternatively, if the degradation is acceptable, xerogel conversion would protect against decay, dramatically slow solute diffusion after drying, and reduce layer thickness considerably, thus improving phase delay control.

Diffusion could also be controlled by chemically bonding the index-altering solutes to the solid matrix of the gel, perhaps by making the gel itself from a mixture of different substances with different refractive indices, or perhaps just using glucomannan or a similar dense gel former at different concentrations. (What's the name of that hydrophilic polymer they use for contact lenses and ceramic gelcasting?)

Correcting chromatic aberration probably requires multiple solutes that induce different dispersions, thus varying IOR independently at different wavelengths.

Avoiding air bubbles is of the highest importance. The simplest way to achieve this would be to do the entire printing process under vacuum, but unfortunately that is not an option with a water-based

gel at room temperature, because water boils rather violently at room temperature in vacuum. Alternatively perhaps the product could be degassed under vacuum before the gel forms, the way we do with polyester resin casting, if there aren't too many bubbles. It would be better to use a liquid (like, uh, polyester for resin casting) that has a relatively low vapor pressure and therefore can be printed under vacuum.

These gels are capable of substantial elastic deformation, often with strains exceeding 10%, which provides another axis of variation; they can be designed to provide an optical transformation which depends in a designed way on elastic deformation.

To research:

- phase-delay spatial light modulators
- konjac gelification
- diffusion
- total internal reflection
- that nontoxic hydrophilic polymer they use for contact lenses and ceramic gelcasting

Topics

Additive smoothing for Markov models

Kragen Javier Sitaker, 2007 to 2009 (updated 2019-05-19)
(11 minutes)

So when you estimate the probability of an event from a sample, you have to "smooth", which is to say that you have to increase your estimate of the probability a little bit so that it is never zero, since concluding that the probability of any event is zero based only on a sample is clearly wrong. The most common way is to add 1 to both the sample size and the number of observed events, and I think that this is in fact an unbiased estimator.

But if you have more information, maybe you can do better. In particular, I was thinking about estimating the statistical likelihood of a particular word being the next word in a sentence. This is useful, for example, for travesty generators like Mark V. Shaney.

Zeroth-Order Markov Models

You can use a "zeroth-order [Markov model][]", and just figure that the probability that the next word is, say, "figure" is the probability of "figure" being any particular word --- regardless of context. That is, it's the number of occurrences of "figure" in your training corpus (plus one), divided by the total number of words in your training corpus (plus one).

That gives you some information, for sure. But if you use that model to generate text, it doesn't look much like real text.

First-Order Markov Models

If you use a first-order Markov model, you use the previous word, say, "just". And you look for the number of occurrences of "just figure" in your corpus (plus one) and divide that by the number of occurrences of "just" (followed by any word) (plus one) in the training corpus. And you use that for your estimate.

Text generated from a first-order Markov model tends to look like relatively real text. If you don't smooth the probabilities, then you only ever get word pairs that occurred in the real text, so the text minimally conforms to a [regular-language][] approximation of the [grammar][] that generated your corpus, and if the model is [ergodic][], the frequencies of the words will additionally be about right. (See also the Jargon File entry on Dissociated Press.)

Second-Order Markov Models

You can use a second-order Markov model and look at the two words, say "and just", and then divide the number of occurrences of "and just figure" (plus one) by the number of occurrences of "and just *" (plus one) in the corpus. This generates even more realistic text, because although it's still using a regular-language approximation, the [DFA][] (a Markov model is a DFA augmented with transition probabilities) can now have N^2 states instead of N states, where N is the number of distinct words in the corpus. So it can be a better approximation.

However, now you run into a problem. The number of states in a second-order Markov model can be greater than the number of words in your text. For example, before I wrote this sentence, this note contained 475 words, of which 162 were unique. That means that a second-order Markov model built from its vocabulary would contain 26244 states, only 474 of which, at most, could contain any sample information!

So if you smooth in the way I suggested above, by adding one to the counts of both "and just figure" and "and just *", you probably wind up with a gross overestimate of the probability of "and just 26244" and a gross underestimate of the probability of "and just the", and so in fact the output can look less realistic than the output from a "zeroth-order Markov model". The usual way to deal with this in Dissociated Press is to not smooth at all; most of the transitions have zero probability. Unfortunately this often results in repeating long passages from the training corpus, especially once you go to third- and higher-order Markov models.

Essentially, such higher-order models are almost inevitably [overfitted][] --- they are so flexible that they end up learning the detailed structure of their training set, and if you try to compensate by smoothing, they don't learn anything.

How to Fix the Problem

However, it seems to me that you could do the smoothing in a more effective way, by using first-order and "zeroth-order" Markov models from the same training corpus to fill the gap.

Effectively, when you're spewing out randomly generated text, the smoothing amounts to adding one extra occurrence of, say, "and just". The standard way of smoothing amounts to assuming that all words that haven't been seen after "and just" are equally likely.

If instead you use your first-order Markov model to decide on that distribution --- in this case, the smoothed probability distribution of words following "just" --- you'll do much better.

And, of course, you can smooth the first-order Markov model using the observed word frequencies (the "zeroth-order model"), instead of assuming that all words in the vocabulary are equally likely.

When you're generating random text, there's no obvious way to smooth the zeroth-order Markov model; you can't straightforwardly generate an arbitrary word you've never seen before. But in some other applications of Markov models, such as data compression, OCR, speech recognition, and entropy estimation, you only have to deal with things that are actually found in the input text.

Note that in the case that the word pair in the current state does not occur in the training set, this reduces to a first-order Markov model. This suggests that it is the Right Thing.

Data Structures

If you want to implement this algorithm, you can use a [suffix array][] or [suffix tree][] on the training set. This allows you to efficiently answer questions like "how many times does 'model, and just' occur in the input, and what is the distribution of what occurs next?" without storing an excessive amount of data. Suffix trees take up an awful lot of space, but Udi Manber and XXX Myers

discovered a reasonably efficient algorithm for constructing suffix arrays in 1989–1991 (or was it Manber and Wu in the 1990s?), and it's used as the basis for the [Glimpse][] search engine. The suffix array merely needs space for one pointer per index point in the original text.

If you want to know what words occurred somewhere preceding a given word or phrase you really want a "prefix tree", which is a suffix tree built on a reversed version of the input corpus. ...

Other Applications

It occurred to me that if you wanted to know what words to boldface in an English text, you might have some success highlighting the lowest-probability words. This could also be useful in source code highlighting: figuring out which tokens are most informative and which are just noise.

References

Dissociated Press correctly describes Markov-chain random text generation, but incorrectly claims that Emacs's "dissociated press" is an example thereof. According to the Emacs manual:

> Dissociated Press produces results fairly like those of a Markov chain based on a frequency table constructed from the sample text. It is, however, an independent, ignoriginal invention. Dissociated Press techniquitously copies several consecutive characters from the sample between random choices, whereas a Markov chain would choose randomly for each word or character. This makes for more plausible sounding results, and runs faster.
>
> It is a mustatement that too much use of Dissociated Press can be a developediment to your real work, sometimes to the point of outragedy. And keep dissociwords out of your documentation, if you want it to be well userenced and properbose. Have fun. Your buggestions are welcome.

Penn Jillette wrote a lovely explanation of Markov-chain travesty generation in his article on Mark V. Shaney, "I Spent an Interesting Evening Recently with a Grain of Salt", published in PC Computing volume 4, number 7, July, 1991, p.282.

The idea of an index point is explained in Gonnet, Baeza-Yates, and Snider's 1991 paper, "Lexicographical Indices for Text: Inverted Indices vs. PAT trees". They had been working on putting the Oxford English Dictionary on CD-ROM, and so they'd done a bunch of work on full-text indexing. Gonnet was at ETH at the time, and Baeza-Yates was already at the Universidad de Chile. The paper is mostly devoted to advocacy of suffix arrays (called PAT arrays in the article) rather than inverted indices. They explain:

> An index point is the beginning of a word or a piece of text which is indexable. Usually such points are preceded by space...

I am a little bit puzzled about this paper; it says the PAT tree data structure "was originally described by Gonnet in the paper 'Unstructured Data Bases' [Gon83]". But the PAT tree is "a Patricia tree constructed over all the possible sistrings [suffixes] of a text," citing Donald R. Morrison's 1968 CACM paper on PATRICIA, which I believe describes constructing PATRICIA trees over all the possible suffixes of a text. But I don't have the paper handy to check.

Tim Bray, who also worked on the OED project, recently wrote a series of articles about the technical aspects of full-text indexing. He also worked for some years at a startup called Open Text whose main product used suffix arrays for its searching. His analysis of why suffix

arrays are not widely used today is very interesting.

PATRICIA has been explained in a lot of places, but unfortunately I don't know where to find the original paper online.

Overfitting is a problem with statistical models generally. Sometimes with neural networks and other machine-learning systems, it's called "overtraining".

Hey, maybe I should look at http://www.stat.cmu.edu/~cshalizi/754/.

Perry Lorier tells me that this is how the PPM* algorithm models; IIRC that's what's used in the champion PAQ family of compressors. I should look into it.

Topics

Rendering iterated function systems (IFSes) with interval arithmetic

Kragen Javier Sitaker, 2014-09-02 (6 minutes)

Not sure if this is a new IFS-rendering idea or not, but I was thinking about a recursive algorithm I'd written to render implicit functions using interval arithmetic, and it occurred to me that you can perhaps use it to do efficient escape-time IFS rendering.

Interval arithmetic computes with intervals of real numbers rather than pointwise real numbers; (-1, -½) / (-3, -2) evaluates to (⅙, ½), for example, because any number in the interval (-1, -½) divided by any number in the interval (-3, -2) will give a number in the interval (⅙, ½); and, if there's no further relationship between the numerator and the denominator, it can give *any* number in that interval.

This is a kind of very simple abstract interpretation; it gives you a conservative approximation of what values a function can range over, given a conservative approximation of its inputs. If you're, say, just looking for zeroes of the function, you can disregard whole swaths of its domain once you've computed that the function's range over that part of its domain is strictly positive, strictly negative, or (in the case of multi-interval arithmetic, which is useful for handling division by zero) a union of the two; this can give you asymptotic speedups.

My 2-D implicit rendering algorithm with interval arithmetic, inspired by my very limited understanding of Flórez Díaz's 2008 raytracing thesis, works as follows:

- evaluates the given function over X and Y intervals covering the entire canvas to be drawn;
- if the output interval it computed excludes zero, it is done;
- otherwise, if it is now evaluating deep-subpixel intervals, it simply draws white;
- otherwise, it divides the rectangle into thirds, and recurses on the thirds. (Halves or fourths or any other number works too, but is less efficient.)

So it occurred to me that you can compute a similar kind of approximation of an IFS's attractor by escape-time analysis. You can run an IFS either forward (toward the attractor) or backward (away from it). If you run it forward, regardless of which transform you choose, you will stay on the attractor if you're already there, and approximate the attractor more closely if you're not; while if you run it backward, even if you're in the attractor, most choices of transform will generally push you off of it, while there exists at least one choice (generally exactly one) that will keep you on the attractor. This means you have to *search* for the correct choice.

So I propose that you divide the canvas into a k-d tree, as in my implicit-function rendering algorithm, but one that remains stored, in which each node (bounding box, let's say, or just "box") is in one of three states:

• it's known to not overlap the attractor;
• it's known to be entirely inside the attractor, which I believe is only applicable if the attractor has Hausdorff dimension the same as the space we're working in;
• it's suspected to overlap the attractor.

We iteratively look for nodes in state 3 that are still big enough to be interesting for our purposes, and we transform them with all of the transforms at our disposal, both forward and reverse. If *any* forward transform leaves its image entirely inside a node in state 2, then this node also changes to state 2; if *all* reverse transforms leave its image entirely inside a node in state 1, then this node also changes to state 1.

This also suggests that we should link nodes in state 3 to the other state-3 nodes that we have discovered to be able to transform to and from them, so that we can propagate state changes properly when we change the state of a state-3 node.

Once this link structure has been discovered, though, we choose a state-3 node to divide into pieces, and transform it again. Presumably once the pieces are small enough, we will choose to stop subdividing, but smaller pieces might transform entirely within a state-1 or state-2 node.

But how do we get any state-1 or state-2 nodes to begin with? If we can compute a conservative bounding box for the attractor to start with, then boxes outside that bounding box will be in state 1, but state 2 is a little trickier. To find the fixed point of any of the transforms, we can solve some simultaneous linear equations, or just exponentiate the transform, since it's contractive. But that still just gives us a point (a unique point, by the Banach fixed-point theorem) rather than an interval that could actually contain another interval inside of it.

And indeed many IFSs will contain no such intervals that are entirely inside their attractors. Others, however, do; consider the 1-D IFS $f_1(x) = x/2$, $f_2(x) = (x+1)/2$, whose attractor solidly covers the space between 0 and 1; or the 2-D IFS

```
f1(x, y) = (x/2,     y/2)
f2(x, y) = ((x+1)/2, y/2)
f3(x, y) = (x/2,     (y+1)/2)
f4(x, y) = ((x+1)/2, (y+1)/2)
```

which I believe similarly solidly covers the square you would expect it to.

State 1 and state 3 are clearly enough to render a fractal, but if your particular IFS has state-2 regions in it, then it will be exponentially more efficient to be able to recognize them, since you'll be able to focus on the boundary of the set instead of deeply recursing on the whole thing. I just don't know how yet.

Anyway, so I think this algorithm, even without state 2, should scale with an exponent very nearly the Hausdorff dimension of the IFS you're looking at, multiplied by the number of IFS transforms.

Topics

High temperature semiconductors

Kragen Javier Sitaker, 2019-12-01 (2 minutes)

The distinction between insulators and semiconductors is temperature-dependent; an insulator becomes a semiconductor when the thermal energy of its electrons is enough to permit a significant number of its electrons to transition from its valence band to its conduction band, continuously replacing those that fall back down. Even if the bandgap doesn't decrease with temperature, as they generally do, this means that any insulating material will eventually become a semiconductor at a high enough temperature.

There is a lot of work on "wide-bandgap high-temperature semiconductors" like gallium nitride and silicon carbide with applications up to 300°. But I think we can think bigger: what about semiconductors at 900° or 1200°? The Stefan-Boltzmann law gives us power dissipation proportional to T^4; if at 100° we can dissipate 1 W/cm², then at 700° we should be able to dissipate 16 W/cm², and at 1200°, some 243 W/cm². This should permit substantially higher computational speeds, even despite other temperature-driven phenomena that will slow down computation.

What materials barely begin to conduct electricity at such temperatures? Quartz, of course, but perhaps also other refractory materials like lime, beryllia, thoria, urania, and magnesia. Other conduction phenomena also come into play at these temperatures, like zirconia's conduction by mobile oxygen ions.

Building devices that operate successfully at such temperatures will be challenging, among other things because they would probably be destroyed if ever allowed to cool to room temperature, and because the lifespans of structures even in fairly stable materials is human-scale at high temperatures. As discussed in Gardening machines (p. 2365), though, automated fabrication should reduce the importance of such durability considerations considerably.

Another difficulty is that you probably still benefit *something* to use as an insulator in your circuits, although room-temperature gallium arsenide integrated circuits seem to mostly do okay just using reverse-biased diode junctions.

Topics

- Electronics (p. 3430) (138 notes)
- Physics (p. 3632) (119 notes)
- Physical computation (p. 3631) (26 notes)

Composing code gobbets with implicit dependencies

Kragen Javier Sitaker, 2018-04-27 (updated 2019-05-21) (3 minutes)

Suppose we have this scrap of code:

```
f = buxus.open(buxus_filename)
viewport = f.bbox()
canvas = buxcanvas.create(viewport)
```

There are lots of ways to read this gobbet of code, but one of them is as a rule which, given values for `buxus`, `buxus_filename`, and `buxcanvas`, can produce values for `f`, `viewport`, and `canvas`. Or possibly fail to.

Or consider these two lines of code separately:

```
interval_size_log = math.log(stop) - math.log(start)
```

That gives us a rule to compute `interval_size_log` given values for `stop`, `start`, and `math`.

```
n_divisions = int(math.ceil(interval_size_log / math.log(1 + spacing)))
```

That gives us a rule to compute `n_divisions` given values for `int`, `math`, `interval_size_log`, and `spacing`. This rule chains nicely with the previous one, which provides `interval_size_log`.

If you were going to apply this approach in a general way to large programs, you'd need some way to namespace these names, of course. And you need some kind of subroutine call mechanism.

In Python 3.3 and later, you can supply a custom mapping to `exec` that logs these accesses as they happen. So you can really write these just as little gobbets of Python code.

Here are some ideas for how such a soup of code gobbets could be useful:

• Conditional computation. Given the knowledge that `interval_size_log` depends on `stop`, `start`, and `math` (or, more pleasantly, `log`) you can efficiently compute all the `interval_size_log` values for a range of `stop` values, a range of `start` values, corresponding sets of `stop` and `start` values (depending, for example, on some index `i`), or independently varying sets of `stop` and `start` values.

This becomes more powerful if you add quantifiers and aggregation, although it is not clear to me how this should work.

• Incremental recomputation, although of course this requires you to make your changes to variables rather than down inside of data structures somewhere.
• Transactions. You can run an arbitrary piece of code that runs in an environment where the variables it reads and writes are monitored, and only commit its writes if none of the variables it read have been changed by a previously committed transaction.
• Inference systems.
• Hot code reloading.

• Lazy computation — although you do have to try to run each gobbet at least once to see what it might produce.
• Pattern matching. You can provide different possible ways to compute the same variable, given different possible inputs.

For many of these applications, you could have a subroutine call mechanism that works by putting some parameters into a new namespace and then trying to pull things out of it. For example:

```
isl = ns(start=1, stop=20).interval_size_log
```

Something like that might be the way to handle quantifiers and aggregation, too. Instead of saying, "What if start=1?" you're saying "What if start is any value in range(10)?" But then of course if you are going to get a scalar value out of it at the end you need to specify how you are going to aggregate the pointwise values.

(Related: A principled rethinking of array languages like APL (p. 1995), Relational modeling and APL (p. 1217), IRC bots with object-oriented equational rewrite rules (p. 838), OMeta contains Wadler's "Views" (p. 842).)

Topics

• Programming (p. 3658) (286 notes)
• Incremental computation (p. 3517) (24 notes)
• Arrays (p. 3326) (17 notes)
• Transactions (p. 3755) (14 notes)

Win32 startup

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

So how does a process get started in Win32?

I compiled a program `fract.exe` with a WinMain with MinGW. Callers:

```
_WinMain@16:
    _main:
        # (which seems to call _GetModuleHandleA and _GetCommandLineA etc.)
        ___mingw_CRTStartup:
            # (which seems to call _ExitProcess@4)
            _mainCRTStartup:
                # 0x401210, offset 0x0210
                no callers
            _WinMainCRTStartup:
                # 0x401230
                no callers
```

This made me suspect that those bottom two are the actual entry points, and that they have magic names looked for by the COFF loader or something. But those strings don't appear in the Wine source code, so that must not be it. But neither their addresses nor their offsets appear in the output of `objdump --full-contents` or `xxd` on the stripped binary.

The top few levels of the stack in a crash look like this:

```
  3 0x0040155a WinMain+0x9d(hInstance=0x400000, hPrevInstance=0x0, lpCmdLine=0x11o
o57f1, nCmdShow=0xa) [/home/kragen/devel/w32dry/fract.c:87] in fract (0x0069fe38)
  4 0x00401867 in fract (+0x1867) (0x0069feb8)

  5 0x004011d9 __mingw_CRTStartup+0xc9 [/home/ron/devel/debian/mingw32-runtime/mio
ongw32-runtime-3.9/build_dir/src/mingw-runtime-3.9/crt1.c:226] in fract (0x0069feeo
o8)
  6 0x00401223 in fract (+0x1223) (0x0069ff08)
  7 0x7b86eeab in kernel32 (+0x4eeab) (0x0069ffe8)
  8 0xb7e0b7a7 wine_switch_to_stack+0x17 in libwine.so.1 (0x00000000)
```

The address +0x1223 is right after the call from _mainCRTStartup to ___mingw_CRTStartup.

Looking in the WINE source, in kernel32/process.c I find "static void start_process", which has this code:

```
    LPTHREAD_START_ROUTINE entry;

    LdrInitializeThunk( 0, 0, 0, 0 );

    nt = RtlImageNtHeader( peb->ImageBaseAddress );
    entry = (LPTHREAD_START_ROUTINE)((char *)peb->ImageBaseAddress +
                                     nt->OptionalHeader.AddressOfEntryPoint);
```

AddressOfEntryPoint is in struct

_IMAGE_OPTIONAL_HEADER... but I can't find where it gets initialized for executables loaded from the filesystem.

HOWEVER! `winedump dump fract.exe` lists, among other things:

```
Optional Header (32bit)
  Magic                          0x10B         267
  linker version                 2.56
  size of code                   0x1000        4096
  size of initialized data       0x1a00        6656
  size of uninitialized data     0x200         512
  entrypoint RVA                 0x1210        4624
  base of code                   0x1000        4096
  base of data                   0x2000        8192
  image base                     0x400000      4194304
```

Which corresponds to most of this data from `xxd`:

```
0000090: 0000 0000 e000 0f03 0b01 0238 0010 0000  ...........8....
00000a0: 001a 0000 0002 0000 1012 0000 0010 0000  ................
```

Topics

Comparable counters

Kragen Javier Sitaker, 2018-08-16 (1 minute)

For generating PWM waveforms, it would be nice to have a state machine design that could expand to larger registers without increasing its depth. Synchronous ripple carry has O(N) depth, while lookahead carry has O(log N) depth, but designs with O(1) depth include ring counters, LFSRs, and Johnson counters. However, those three cases have disadvantages: ring counters require N bits to represent N states, while Johnson counters are almost as bad, requiring N/2. LFSRs are nearly optimal in flip-flop efficiency and can always work with depth of one or slightly more, but comparing LFSR states for beforeness is very difficult. In some cases it may be adequate to compare the LFSR state for equality.

Carry-save counters have depth of 1, but it is also not clear to me that they can be compared for greaterness with any reasonable degree of circuit depth. Also, they use twice as many bits as a regular binary counter, and while this isn't as bad as a Johnson counter, it's significantly bad.

The context of all of this is that it would be nice to have counters that we can clock very quickly.

Topics

- Performance (p. 3621) (149 notes)
- Electronics (p. 3430) (138 notes)

The ultimate capacity of human memory if spaced-practice memorization works as advertised

Kragen Javier Sitaker, 2017-01-04 (updated 2017-01-08) (14 minutes)

Here I am going to consider learning with ulterior motives — the learning we do in order to become more human and more capable, the ostensible motive of an educational system, rather than the learning we do purely for pleasure. None of what follows applies to recreational learning, because it all considers time applied to studying as a cost, not a benefit.

The spaced-practice psychology literature find that generally the ideal spacing interval is 10% to 20% of the time you want to remember something — so, say, 15 years if you want to remember it for your entire life. You would think that this would imply you need exponentially increasing study intervals, since it would seem that your chances of remembering that $\int \cot x\,dx = \ln|\sin x| + C$ if you hear it once every 15 years are very poor indeed. So far, though, no convincing experimental evidence confirms this apparent common-sense conclusion; the studies that have been done did not find evidence that such "expanding-interval" spaced practice is superior.

However, it seems clear that the schooling system as it stands is at the opposite extreme. K-12 is 13 years of 36 weeks of 30 hours of torture, about 14000 hours of institutional child abuse in total. Abusive treatment aside, that's a massive cost: about 7 years of normal working hours or 2.4 years of waking hours. You would like such a massive cost to be well-spent, providing an equivalently massive benefit, for example of education. But, by concentrating them in 13 years, we guarantee that any learning point with a 15-year practice interval is presented at most only once; the possible degree of recall after a single presentation is very low even when testing is the next day, much less decades later.

So, in order to achieve any kind of reasonable level of recall at all, schools waste the students' time with massed practice, guaranteeing that nearly all of what is taught will be forgotten by adulthood. Worse, the waste of practice-massing in schools is fractal, happening at many levels:

- K-12 schooling is massed in 13 years rather than being spread throughout a person's entire life;
- taking courses on different topics each year guarantees the loss of most learning even before K-12 schooling ends;
- summer vacation guarantees the loss of nearly all new learning from the last months of the school year, and although this could be avoided by devoting those months exclusively to review of earlier material, this is not done;
- moving from unit to unit within each course, with little attention given to material from previous units, guarantees the loss of most learning from the course even before the end of the course;

• pre-scheduled large examinations incentivize students to "cram", massing their practice in the days immediately before the examination, in order to cheat the examination into indicating a level of mastery of the material that they do not in fact possess, or, more precisely, will lose within a week or two;
• and, although the evidence for this is less clear in psychological studies, hour-long classes are very likely to result in poorer retention from one day to the next than if they were split into two half-hour chunks at different times of day. Teachers assign homework to compensate somewhat for the loss of retention, but this worsens rather than improves the waste of time.

Psychological studies have shown 40% increases in learning throughput (that is, 70% as much study time to reach the same level of achievement) from adding a single level of distribution to practice sessions, which is something more than two standard deviations. If we could increase learning throughput by 40% six times in succession, the total increase would be 650%. A person whose life had not been wasted by any of these six levels of massing could — very speculatively! — perhaps learn seven times as much per hour of study, reaching apparently superhuman levels of intellectual achievement, at no increase in total study time.

However, this learning would be spread across a lifetime rather than concentrated in the 5–18 range. If we spread it across the 70-year-long 5–75 age range, it would be only 200 hours per year (33 minutes per day). If we were to accept the optimistic 650% improvement speculation above, this would be equivalent to some 1500 hours of regular K-12 schooling per year, only moderately better than the 1080 hours in the standard system. You would only start to see a really significant difference after age 18, when the students using a program rationally designed to optimize their learning, rather than subjugate them and provide fake "educational achievement" on tests, continued learning, and their knowledge and competence continued to grow, while the victims of industrial-age schooling instead began the inexorable intellectual decline that is such a universally-remarked- upon phenomenon today.

(These numbers change a bit if we include "higher education", which, for an undergraduate degree, is typically about 5 years of about 40 hours a week, 28 weeks a year, between lectures, labs, and homework, about 5600 more hours; most of the same criticisms can be applied to it, some *a fortiori*. This would bring us to almost 20000 lifetime hours of schooling, 280 hours a year if spread across 70 years.)

This even allocation of 200 hours per year represents a tradeoff, though, and probably not a good one. If you learn a skill at 10 years old and die at 75, you can use that skill to your advantage for 65 years. But if you learn it at 65 years old, you can only use it to your advantage for 10 years. This is the logic that underlies the traditional soul-destroying system of studying first, then working, and in itself it isn't flawed; it just doesn't take into account the psychological discoveries made since the 18th century. It's yet another form of the exploration-exploitation tradeoff, a general feature of bandit problems.

A better tradeoff would weight the expense of learning toward the beginning of life, while leaving the human more time to exploit their

skills toward the end of their life. The optimal curve depends on many factors, including your model of how skills improve life utility, how studying fatigues you, how risk-tolerant you are about dying early or late, the probability of dramatic human life extension, and so on, but a linear reduction from 400 hours per year at 5 years old down to 0 hours per year at 75 seems likely to capture the majority of the benefits from the optimal curve.

On this study schedule, you start by studying 66 minutes per day on your 5th birthday and taper down by about 56 seconds per year (154 milliseconds per day) until reaching 0 on your 75th birthday. On your 18th birthday, when you would normally graduate from high school, you are down to studying almost 54 minutes per day, and you've spent 4740 hours of study, about a third of what the victims of high school have. But, because they were in a system optimized to waste their time, you've learned more than twice as much.

How much is that? If we take the 1.8 minutes per "card" in Gwern's review of spaced repetition systems as a good approximation, it would be about 158 000 "cards", somewhat more than the users of these systems report having memorized, somewhat more than the number of words in the Oxford English Dictionary. Of course, most knowledge cannot be separated into isolated "facts" or "cards" in this way, but I suspect that improves the situation for our optimized learners — when you work on a real-world problem, you are inevitably practicing many procedural skills at once, while recalling a flash card only reminds you of a single fact.

This suggests that the speculation in my 2010 post, "could we learn a new foreign language every week?", might actually be plausible, as unlikely as it sounds. The 70-hour budget mentioned there works out to about 2 hours per year or about 2300 "cards", which is not far from the size of existing Anki decks designed to reach basic competency in a foreign language; that post estimates that you really need more like 8500 cards' worth, which would work out to 260 hours rather than 70.

Devoting an hour a day to a system like Anki, it's reasonable to add about 30 "cards" per day with an hour a day devoted to practicing them, so you can't compress those 8500 cards into less than about 280 days, and that only works if that's the only subject in which you're adding new "cards". While this takes into account that you're continuing to practice other things during that time, it seems likely that learning will be more efficient if you aren't adding material entirely from a single subject, spacing out the addition of similar things. I've found, for example, that memorizing the atomic weights in the periodic table, I frequently confuse elements that I add in the same day (especially if they're otherwise similar), and in my hanzi recall practice, I'm currently confusing the characters 是 and 在, even though they look nothing alike, just because they're the same kind of thing and I learned them at about the same time.

However, if you add those 8500 cards over the course of, say, four years, you could easily speak three foreign languages at a basic level by the time you're 18, at a total cost of about 1000 hours out of the 4700 total prescribed above.

(It seems likely, though not certain, that acquiring languages has a critical period in childhood and later becomes much more difficult or even impossible to do in the same way, so it might be better for small

children to spend more of their time on foreign-language learning and less on, say, math and history.)

At 8500 cards per language and 158 000 cards in all, you could expect to speak 19 languages with basic fluency in an average lifetime if you were to focus entirely on foreign-language learning, all in the same amount of time a normal person loses to K-12 education. This may not be the best use of your time, but it gives a very plausible example of the kind of superhuman intellectual achievements we could expect with an optimized learning system, even if all we optimize is the practice schedule. Note that this number is not computed from the speculative exponential computation of the waste of traditional K-12 education above, but from observed time per card in existing spaced-repetition systems and a speculative guess that you can achieve basic fluency (roughly equivalent to that of a six-year-old native speaker, who has some 6000 vocables) with 8500 cards per language.

As some kind of indication that such large improvements are possible, I'm currently studying the Hebrew aleph-bet using Anki since a week ago. Currently I'm at 80% correct on the review cards and have spent a total of 21 minutes on its 22 cards over those 7 days. I'd probably be doing better if I hadn't added half the cards all at once on Saturday, resulting in confusion between some letters like taf and tet; it will probably take me a total of more than 40 minutes to finish memorizing the whole thing. But how long do Israeli children or Hebrew students normally spend learning the alphabet? I think it's typically several hours.

Above I've talked about how you could spend the same 14 000 hours of K-12 education or 20 000 hours of an undergraduate degree in a much more efficient manner, learning several times as much, at the same cost in time and probably without even learning more slowly. However, in practice, the Jevons paradox will probably kick in. Just as improving the efficiency of steam-engines, allowing them to do more work on the same amount of coal, resulted in them being applied to new applications and increasing the total consumption of coal, it seems likely that if time spent learning things is dramatically more effective, people will consider it worthwhile to learn more things.

So, a typical optimized learner probably will not spend only an hour a day studying in this new, more efficient manner; they might be unable to resist spending two or three hours a day, even if the learning is not quite as efficient, learning perhaps 50% or 100% more. At this rate, an average learner might complete the equivalent of an undergraduate degree's 20 000 hours in 6000 hours, sometime around age 12. (XXX actually do the calculation!)

Topics

• Spaced practice (p. 3721) (2 notes)

Simple dependencies in software

Kragen Javier Sitaker, 2014-06-05 (9 minutes)

Suppose we have a function to compute some value:

```
def f(a, b, c):
    if get(a, 3) > 5:
        return get(b)-get("x")
    else:
        return get(c)
```

Now, we run this function in an environment where we will record its return value where we will record its return value for the results of pending or future calls to `get()`. (So far I haven't specified how `f` gets associated with some value you can pass to `get`.) We also record its calls to `get` as resulting from this invocation of `f` and may even throw exceptions from some of the `get` calls, for example if their values are not known yet, exceptions which will be caught by the context that invoked `f`. Now we know which set of `get` invocations the current return value of a particular call to `f` depends upon: clearly `f(3, "/", "@")` invokes `get(3, 3)`, and if that returns 4, it will cause it toinvoke `get("@")`. Now we can infer that the return value of `f(3, "/", "@")` depends only on `get(3, 3)` returning 4 and on `get("@")` returning whatever it returned at the time; so it's valid to cache `f(3, "/", "@")` until one of those two values changes. In particular, we do not need to worry about `get("/")` or `get("x")`, since they weren't invoked. But if `get(3, 3)` changes its return value, then the cached value for `f(3, "/", "@")` is no longer valid, and when a new value for it is computed, it might have a different set of dependencies — it might depend on `get("@")`, for example.

This allows us to write code in a fairly imperative style and automatically transform it into a reactive dependency-driven computation.

Meteor's implementation

Meteor's `Deps` facility is probably the most popular implementation of this in practice today, although I first encountered a variant of it in the paper that popularized software transactional memory, Composable Memory Transactions, in 2005; and The Trellis is an older Python package that does the same thing. `Deps` has a few surprising design choices, which are interesting in that they represent the experience of Meteor's developers and users using it extensively in practice over the last two years, rather than merely in theory:

• Its implementation supports nesting;
• If an exception is thrown the first time a computation is invoked, it prevents the computation from ever being rerun in the future; there's also an explicit `computation.stop()` API to prevent this.
• Computations are rerun eagerly when their dependencies change, but only once the system is idle or someone calls `Deps.flush`, which you can't do from inside a Computation; this enables you to use them to e.g. update the DOM in the page rather than just producing a return

value, and the delay-until-idle makes them serializable so that one Computation doesn't see partial updates produced by another Computation not having finished;
• they additionally have an `onInvalidate` callback which is invoked immediately when their result is invalidated;
• Computations normally communicate with one another by writing to a shared global store (e.g. the database, which is replicated onto the MongoDB server) rather than by nested invocation, although nested invocation is also supported;
• dependency links have separate identity from the variables they track:

> The reason Dependencies do not store data themselves is that it can be useful to associate multiple Dependencies with the same piece of data. … A Dependency could represent whether the weather is sunny or not, or whether the temperature is above freezing. `Session.equals` is implemented this way for efficiency. When you call `Session.equals("weather", "sunny")`, the current computation is made to depend on an internal Dependency that does not change if the weather goes from, say, `"rainy"` to `"cloudy"`.

• Dependency links remove their dependent computations after invalidating them, trusting that they will re-add the dependency again if necessary.

I've been told by Meteor users that Meteor doesn't have transactions, but the Computation and Dependency objects described here are essentially the same ones that implement transactions in a software transactional memory system: a Computation is a transaction, and its Dependencies are the variables it reads. (It just doesn't support rollback, which would require at least logging the variables it writes in order to undo the effect.)

Continuously changing variables

Time changes continuously, so in the simplest possible system, any transaction that depends on the current time will be constantly invalidated, and thus constantly recomputed if you're using an eager re-evaluation strategy. But, as the `Session.equals` example above from Meteor shows, in some cases you can make the transaction's dependency considerably more specific. You could say `if time.between(nyc.hour(10,00), nyc.hour(10,05))":`, for example, so that your transaction would only be rerun upon crossing the 10:00 or 10:05 boundary.

This is difficult to generalize, but there are ad-hoc approaches that may work well in practice. For example, you could sample a variable periodically, and assume that it doesn't change at other times. Meteor in fact does this on the server to detect MongoDB changes that aren't intermediated by Meteor itself.

Dependencies with exceptions as a substitute for futures

Futures are a mechanism for writing code that computes with data that aren't known yet — actually, several similar mechanisms. The first was proposed by Baker and Hewitt in AIM-454 in 1977. They used "future" to refer to more or less what we know in Haskell as a "lazy value" — Baker and Hewitt were proposing that structuring the program's run-time state as a pure dependency graph, and then eagerly computing it in parallel while incrementally

garbage-collecting the parts of the dataflow graph that had been cut off by a conditional, would be a good way to write parallel programs to efficiently compute on massive multiprocessors. They turned out to be wrong about that, but nobody would actually build a massive multiprocessor until 1983, so we have to give them credit for trying. (Besides, it's not too far off from `make -j 16`, which is an effective way to program moderately massive multiprocessors, and Stu Feldman wrote the first, non-parallel, version of `make` only the year before.)

Basically their mechanism was that your computation would block when it attempted to apply a primitive operation such as arithmetic to a future, unblocking once the value was available.

Composable Memory Transactions proposes that transactions can "block" by, essentially, throwing an exception (called `retry` in the paper) which aborts the transaction, marking it to be retried when one of its dependencies changes. As long as the code inside the transaction really has no communications with the outside world that aren't intermediated by the transactional memory (a problem cited by Joe Duffy in 2010 as one of the killers of STM.NET), then the effect is very similar to the transaction magically not being able to execute at all until one of the dependencies that would guide it toward the fatal exception is modified. It's like the system knows that the transaction would fail, so it doesn't attempt it, and it magically knows what inputs the transaction takes. (Of course this won't work in systems like Meteor that don't support rollback.)

Consider this Python code to calculate how much a share of AT&T stock costs in pounds sterling:

```
def t_in_gbp():
    quotes_csv = "http://download.finance.yahoo.com/d/quotes.csv"
    data_url = lambda sym: quotes_csv + "?f=sl1d1t1c1ohgv&e=.csv&s=" + sym
    t_data   = www.get(data_url("T"),        max_age=minutes(5))
    gbp_data = www.get(data_url("GBPUSD=X"), max_age=minutes(5))
    price = lambda text: float(csv.reader(StringIO(text)).next()[1])
    return price(t_data) / price(gbp_data)
```

You could implement this using blocking HTTP client calls, storing the result in a cache, and then, to get concurrency, spawn off a separate thread for each such computation, sharing the result between threads. As an alternative to threads, though, you could run `t_in_gbp` in a transaction and have `www.get` try to fetch the data from the cache, initiating the fetch and aborting the transaction if the cached data is stale.

Topics

Sandwich theory

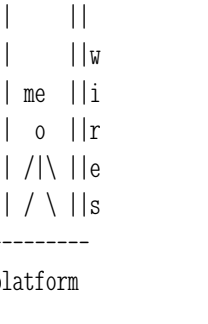
Kragen Javier Sitaker, 2019-08-05 (updated 2019-08-29) (31 minutes)

Sandwich theory is the theory of sandwich structures, which are a kind of anisotropic composite material consisting of two stiff "facesheets" bonded to a lightweight "core", like a two-dimensional version of a one-dimensional I-beam; corrugated cardboard and drywall are the most common examples. I started reading about this because of an interest in cardboard furniture (see Cardboard furniture (p. 742)) and recycled materials.

The basic theory

Basic strength-of-materials background

When built with materials that aren't full of cracks, structures stressed in tension are generally quite strong, and in a way that barely varies with size. Piano wire (aka music wire — a kind of steel used mostly for springs nowadays) has a yield stress of about 2.5 GPa, so a platform that can support my 110 kg can be supported by 0.43 mm² of piano wire, a wire 0.75 mm thick. I'd have to sit down on it very slowly to not break the wire on impact, and if you just connect the wire directly to the platform, it will be kind of tippy; but hanging by three or four such wires would be safe:



Scaling

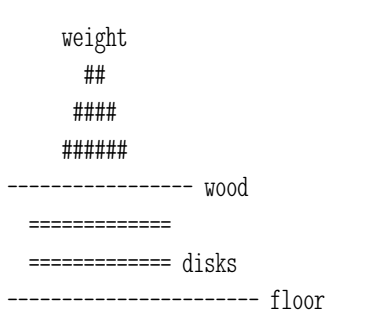
Moreover, this holds whether the wires are connected to the ceiling 2.6 meters above my floor or to the top of a 100-meter-tall building I'm swinging at the base of. 100 meters of four 0.43 mm² piano wires would only weigh 1.4 kg, which is not a significant addition to my weight; you have to get into kilometers of height before the weight of the wire is a significant fraction of what it can support.

As Descartes observed, if we scale up the whole situation instead of just the cable length, things get a little worse: a King-Kong-sized person ten times my height, width, and breadth would weigh a thousand times what I do, about 110 tonnes, while scaling up the cables from 0.75 mm to 7.5 mm in diameter only strengthens them to be able to support 11 tonnes. You'd need to increase them to 24 mm in diameter to support the Kragen Kong. Still, though, that's not such an extreme departure from uniform scaling.

Compressive strength

There's a compressive strength that works the same way; if I put a 10-cm-diameter disk of cardboard under a wooden board and start

piling weight onto the board, there's a certain amount of weight above which the cardboard will get crushed; let's say 300 kg, which is in the ballpark but not a measured figure. If you use several different disks of the same cardboard, they'll all crush at about the same weight, and moreover that weight doesn't change even if you stack two or three or ten of the disks up; the same 300 kg will suffice to crush the stack of ten disks as to crush a single disk. If you use a bigger disk, the weight needed to crush it will go up proportionally.



Buckling

But if you use smaller and smaller disks (and proportionally smaller weights, including the wood), or taller and taller stacks, something weird happens. Well, not weird if you have any experience with material objects, but different. Once the stack gets taller than a certain limit — say, around ten times its width, for cardboard — taller and taller stacks can support less and less weight. So I can stand on one 10-cm cardboard disk, and I can stand on a stack of ten of them, and I can probably stand on the column of 650 of them that it would take to reach 2.6 meters, but I definitely cannot stand on a column of 100 meters of them. In fact, I probably can't even build one. Adding more cardboard to the stack in this way *reduces* its compressive strength rather than leaving it unchanged.

I'm not talking about the top of the stack falling over sideways, which also gets harder to avoid when the stack is taller; the phenomenon I'm talking about happens even if you go into an elevator shaft and rest an elevator-shaft-sized wooden platform on top of the stack of smaller-than-shaft-size disks. I'm talking about *buckling* (*pandeo* in Spanish), where the *middle* of the stack bends outwards in some random direction and it stops being a linear stack. Buckling is governed not by the compressive strength of the column but by the *rigidity* of the column.

Euler worked out the basic math for this. For compressive forces below Euler's critical value, any small bend in the column, for example due to sound waves, gets straightened back out by the column's elasticity. But once the column is bearing a load over Euler's critical limit, such a small bend will get amplified by the compressive stress more and more. Euler's critical load is $\pi^2 EI/(KL)^2$, where E is the modulus of elasticity of the beam, I is the least planar area moment of inertia of the column's cross section ($\pi r^4/2$ for a solid circle), L is the length of the column, and K (the "column effective length factor") varies between ½ and 2; it's ½ in this case because the cardboard disks resting flat on the ground and the elevator-shaft platform are not free to rotate.

Sadly I don't have the faintest idea what the modulus of elasticity of corrugated cardboard in lateral compression is, but the things to

notice about this expression are:

• It's inverse-quadratic in L, so a column ten times as long can bear only one-hundredth the weight without buckling.
• It's directly proportional to the modulus of elasticity, so a beam that's made from a material that's one-third as stiff can only bear one-third the weight without buckling.
• It doesn't involve the *strength* of the cardboard *at all*; it only depends on its *stiffness*.
• Compressive and tensile strengths are proportional to the cross-sectional area, but Euler's critical buckling load is proportional to the area moment of inertia, *not* the cross-sectional area.

The consequences for designing large structures

Point #1 here means that, normally, large structures fail because of buckling before than they fail because of tensile or compressive failure, unless they're made of materials that are full of cracks. This means that if you can improve buckling behavior somehow, you can dramatically extend the size of structures you can construct.

These last three points are crucial to sandwich theory! By stiffening columns with high-stiffness facesheets, and increasing the moment of inertia by moving those facesheets further apart by inflating the core (even at the cost of some strength), we can make sandwich-structured composites that resist buckling enormously more effectively than their component materials.

There's also flexural stress. Sometimes large structures fail because of flexural failure — trees dropping branches or blowing over in a storm are examples. Moreover, flexural stress shares with buckling the property that you really need a great deal more material to resist the load than you would expect — a steel beam that can support my weight in flexion is generally going to need a great deal more than 0.43 mm² of cross section. I am going to mostly ignore flexural stress, except as it contributes to buckling, because ⓐ the same sandwich-panel construction that helps with buckling also helps with flexural stress, and ⓑ flexural loads for a given beam construction only fall inverse-proportional to the characteristic dimension of the structure, while buckling loads are inverse-proportional to its square, so for large enough structures buckling still dominates.

Cracks

The cheapest building materials — earth, brick, other fired-clay ceramics, concrete, mortar, glass, plaster of Paris, and often even stone — are all full of cracks. As long as they're in compression, this doesn't matter, but once they're in tension, the cracks form stress risers that weaken them enormously. This makes it difficult and dangerous to build large structures from them unless they're heavily reinforced with materials with good tensile strength. Adobe is made from earth reinforced with straw, plaster of Paris is commonly reinforced with horsehair, and concrete is commonly reinforced with steel rebar. Even so, larger buildings are invariably steel-framed rather than relying on the tensile strength of even reinforced masonry.

Basic sandwich-theory background

The moment of inertia around the x-axis, I_x, is $\iint y^2 dx dy$ where the area integrated over is that of the cross-section. For panels, that's the only one we care about, because the panel is much wider than it is thick, so if we take our x-axis parallel to its surface, the y-axis

moment of inertia is orders of magnitude too big to worry about.

This says that if we have some cross-sectional area of material, we can increase its moment of inertia proportionally, to an arbitrarily high level, by moving it further from the x-axis:

```
small moment  large moment
               *****


  *****
--*****--     --*****--
  *****


               *****
```

The trouble with this is that once we insert empty spaces in the middle, it stops being a solid object. We need to put enough stuff in the middle to keep the parts moving together; the stuff in the middle is pretty much only stressed in *shear*. Also, when the column is flexing, the facesheet on the inside of the bend is in compression merely from the flexing, and that compression can cause *it* to buckle.

If we only have one material to work with, we can improve the situation by moving almost all the material as far as possible from the x-axis in both directions, leaving the rest of the material as fine "webs" in between to resist the shear, and we have an I-beam, or rectangular tubing, or a channel, or a sort of sandwich panel:

```
  *****     *****     *****     **********************
    *       *   *     *         *   *   *   *   *   *
    *       *   *     *         *   *   *   *   *   *
----*---- --*---*-- --*------ --*---*---*---*---*---*--
    *       *   *     *         *   *   *   *   *   *
    *       *   *     *         *   *   *   *   *   *
  *****     *****     *****     **********************
```

(This kind of sandwich panel, a single plastic with square channels running through it, is in common use for translucent roofing for bus stops around here; it's called "twinwall plastic" when made from polycarbonate, and similar sandwich panels made from other plastics are called "corrugated plastic", "corriboard", or "coroplast".)

More generally, you can use a different kind of material in the middle, maybe one with not much strength of any kind, just enough to resist the shear stress. Incidentally, that shear stress is *also* inversely proportional to the distance between the facesheets, so if your panel is thick, you can get by with a very wimpy core material. You do still need enough tensile and compressive strength provided by the cross-sectional area of the facesheets *and* the core to withstand the tensile and compressive stresses on the panel from its edges, but in a large structure, that is a much easier problem than preventing buckling.

XXX understand the theory

An underappreciated aspect of sandwich panels is that they are not only stronger (in flexure and buckling) but also softer than the same materials would be if solid. This can increase their impact strength.

Common current sandwich-panel examples

Single-wall corrugated cardboard, as I said, is the most common kind of sandwich panel in daily life. It has a smooth "linerboard" paper on the "outside", according to Angela Ben-Eliezer, a corrugated paper layer of "flutes", and a second linerboard paper on the "inside" which is not smooth, all glued together, typically with sodium silicate. The linerboard layers play the role of facesheets, while the flutes play the role of the core — a highly anisotropic core, in this case.

The inner fluted layer is mostly air — typical corrugated cardboard is made from paper of about 90–130 g/m², which makes it about 150 μm thick, but has a 4-mm-thick fluted layer, so the fluted layer is about 96% air and 4% paper, and the cardboard as a whole is about 88% air and 12% paper. In fact, only part of the flutes — about half — is in the fluted layer itself. The other half is glued flat to one or the other linerboard, making it part of the facesheet.

Double-wall and triple-wall corrugated cardboard is a less common material, used for higher-strength applications. Triple-wall cardboard has three layers of flutes and four layers of linerboard, and the interesting thing here is that the inner layer of flutes is typically substantially thicker than the outer layers. That is, it's a sandwich panel whose facesheets are themselves sandwich panels! It's a recursive sandwich!

Drywall is a sandwich of gypsum (plaster of Paris) between two sheets of paper. Unlike the other examples, this isn't to help it bear buckling loads; it's to give it tensile strength (and thus flexural strength) so you can carry it around and drive screws through it, instead of shattering the way plaster on lath would do if you treated it that way.

Corrugated sheet steel is similar to corrugated cardboard, but just the flutes, without the linerboard; it's just a wavy sheet of steel. This is not the most efficient use of material, since the "core" consists of roughly as much steel as the "facesheets", but it's very cheap to make. Slightly more elaborate corrugated sheet metal, like that used for the walls of shipping containers or aluminum siding, uses sharp bends to increase the amount of material in the "facesheets" while still being inexpensive to fabricate by bending a single sheet.

Foamcore is a sandwich made of two sheets of paper with styrofoam in between, commonly used for architectural models and picture framing.

The Hexayurt is made from sandwich panels sold in the US and some other countries for house insulation; these are foamed polyisocyanurate with aluminum-flashing facesheets. Similar lightweight rigid sandwich panels made of various materials are common insulating materials; when the facesheets are some kind of structural board material like OSB or drywall, it's called a "structural insulated panel".

Current more exotic examples

Fiberglass fabric glued onto the surface of styrofoam is a common material for small airplanes, especially model airplanes.

RV enthusiasts have taken to fabricating furniture by cutting styrofoam to shape, fitting them together, and coating the surface with some kind of stiffener, such as fiberglass window screening material stuck to the foam with latex paint.

I'm seeing sandwich panels on some of the bus lines here in Buenos Aires; they seem to have polyethylene cores and melamine facesheets.

Cement board has been a common building material for decades; it's a sandwich with a cellulose-reinforced portland cement core and glass-fiber mesh facesheets. It's kind of like drywall, but a lot stiffer; it's useful as an underlayment to keep tile floors laid over wood from breaking when the wood flexes.

Common FDM 3-D printing slicers will, by default, fill the interior of the model (whatever its shape) with a honeycomb, thus making it a sort of three-dimensional quasi-sandwich-panel. This is mostly to cut down printing time, but it also helps to compensate for PLA's abysmal impact strength.

In aerospace, sandwiches with sheet-metal facesheets and metal honeycomb cores are common.

The Grenfell Tower fire in 2017, which killed 72 people, about a quarter of those present, was caused in large part by the use of Arconic Reynobond PE sandwich panels, with aluminum facesheets around a polyethylene core (not, I think, foam), which were covering an 150-mm-thick layer of polyisocyanurate foam under a ventilation space.

Candidates for upcycling garbage into sandwich panels

Very common materials: styrofoam, cardboard, boPET, TetraPak, beer cans, plywood, sheet metal

Styrofoam and cardboard are the two most obvious candidate core materials, but many others are possible. boPET, as in discarded chip bags and one of the layers in TetraPak drink boxes, is fairly stiff, and so boPET and TetraPak are abundant candidates for facesheet materials; cardboard is also a good candidate.

Thin aluminum sheet, as from drink cans, is considerably stronger and stiffer in tension than either of those; it might make a good facesheet. More broadly, sheet metal is often discarded, and it makes a fantastic facesheet material, as do plywood and OSB.

Drink cans have a particularly interesting property: they are mostly coated on the inside with a thin layer of very chemically inert plastic, usually a food-safe epoxy. It might be possible to use this coating as a ready-made glue (see below about the difficulties of glues) to stick the rolled-flat shreds of shredded cans together, like wood chipboard. Some combination of heat, pressure, and previous functionalization with cold plasma (see Cold plasma oxidation (p. 2406)) might do the trick. This should not require temperatures anywhere close to those necessary to melt the aluminum.

Expanded sheet metal — that stuff they use for security grilles and transparent nonskid steel stairs — is likely an *excellent* facesheet material, but it is only very stiff along one of its axes.

Fabric-backed vinyl is discarded in bulk when billboards are updated with new advertising, and it has its merits, but it is sufficiently useful that it already tends to get recycled.

Cloth and circuit-board facesheets

Woven cloth (from discarded clothes, mattresses, etc.) is another candidate facesheet ingredient — unbroken chains of fibers running in long straight lines along the surface will contribute more stiffness than the same fibers in a more disordered mass like that of paper. Knit cloth would contribute little stiffness by itself, but "nonwoven cloth" ought to be intermediate, like paper. Glass-fiber cloth or cloth woven from Kevlar, Nomex, or Spectra would be ideal, but are hard to find, except in the form of glass-fiber window screens. Some cloths are probably not suitable: nylon and silk are elastic enough that they probably wouldn't add enough stiffness.

(Aluminum or steel window screens would work too.)

Of course, these materials by themselves need some kind of glue to turn them into facesheets. (See below about glues.)

Discarded circuit boards are extremely stiff: they are made of glass-fiber cloth impregnated with a plastic such as phenolic resin. They are relatively abundant in the waste stream, and would be ideal.

Other core materials derivable from garbage

Since the core is ideally very thick, it can be made from a quite weak material, including plastic foams, especially microcellular foams. So foaming discarded thermoplastics, for example with nitrogen, is a potentially very interesting way to upcycle them.

Most organic matter can be reduced to some kind of charcoal by heating at high enough temperature without oxygen. Charcoal is very rigid — more so the purer it is — and lightweight, because it's basically a foam. It might make a very reasonable core material. It's usually full of cracks, so unless you crush it into a powder, its tensile strength is very low. One possible approach to that problem is to form the organic matter into a fine *open-cell* foam *before* carbonizing it, for example by baking it into bread, and then heating and cooling it very slowly. That might yield a carbon foam with reasonable tensile strength but I wouldn't bet on it.

Carbon black — charcoal formed as smoke rather than in solid form — is routinely used to add tensile strength to rubber, for example in car tires and shoe soles. It, too, can be produced from almost any organic matter.

Pretensioning facesheets

Facesheets are only strong when in tension; when in compression they rely on the core material to support them against buckling. If there's a way to pretension them somehow, this could increase the strength of the whole structure. One way would be to simply pull on the facesheets during the lamination process, so they are stretched while the bond is being formed. Alternatively, some plastics might enter into tension when heated, the way PET drink bottles and shrinkwrap do. Another alternative: in cloth aircraft construction, after construction, the cloth surface is impregnated with a substance called "dope", which shrinks as it cures, setting up tension in the cloth. Is there a "dope" that could be used to pretension sandwich panel facesheets in this way?

Alternatively, if the core material is a closed-cell foam, you might be able to pretension the facesheets this the other way around, by setting up compression in the foam. A couple of ways to do this: you could assemble the sandwich panel in a chamber under high pressure,

thus partly collapsing the cells of the foam; or you could arrange for air to diffuse through the foam into the cells after assembly, for example by baking the foam in a vacuum *before* assembly, long enough for a substantial fraction of the air to diffuse out.

Typically the blowing agents diffuse out of the foam after it is produced, and this is often considered problematic when it happens faster than air diffuses in — it means that the foam contracts shortly after fabrication, then later expands again after absorbing air — but it produces a critical period during which sandwich-panel fabrication would pretension the facesheets. So, perhaps the same phenomenon could be provoked intentionally, either by foaming the plastic with the usual blowing agents or by provoking a very volatile gas like hydrogen, helium, or methane to diffuse into the foam first. (A chamber full of methane is a lot easier to produce than a vacuum chamber, if perhaps more dangerous.)

Stiff facesheet materials (some weak)

Because the primary purpose of the facesheet is increasing *rigidity*, not strength, in order to resist buckling, it might make sense to use materials with high rigidity and low strength; glass, plaster, and cement come to mind.

Generally, filled plastic systems can get a lot of rigidity from the filler, so using a high-rigidity filler like rocks, sand, clay, cullet, grog, eggshells, carbon, or machining swarf might make sense. (The usefulness of such fillers isn't strictly limited to "plastics" as such; they are of course a key part of concrete, mortar, and fired-clay ceramics, in which cases they mostly contribute strength rather than rigidity.) Such fillers work better with large aspect ratios, so that the individual filler particles can more easily be much closer together than their large dimension, so that when the material as a whole is stressed in tension or compression, the binder between them is stressed in shear over a much larger cross-sectional area than that of each interacting filler particle. This way, most of the tensile deformation of the bulk material comes from stretching filler particles rather than stretching or shear-deforming the less-rigid binder.

A hexagonal, herringbone-brick, or jigsaw-puzzle pattern of facesheet tiles, with a little space between them, could provide rigidity without causing the facesheet to break when its elastic limit is exceeded.

These approaches aren't viable if ⓐ the core material is so weak that you're depending on the facesheet to provide tensile *strength* as well; or ⓑ the sandwich panel is sufficiently stressed in flexion, as opposed to buckling, to depend on the facesheet's strength as well as its rigidity.

Other facesheets

Masonite, medium-density fiberboard, wood, paper, PMMA, CDs, wet wipes, fabric-softener sheets, and many other things are also plausible facesheet materials.

Particle board is made from sawdust pressed into panels under pressure with a glue, typically melamine resin or phenolic resin. This approach, possibly with different glues (see below about glues), might be feasible for fabricating facesheet panels from a variety of waste-stream fiber sources: sawdust, wood shavings, coffee grounds,

yerba mate, grass clippings, hair clippings, machining swarf, dry leaves, chipped prunings, shredded paper, shredded cardboard, dehydrated sliced vegetables, fiberglass from demolition (at the risk of asbestos), shredded cloth, bagasse, straw, pencil shavings, peanut shells, sunflower-seed shells, wicker, or chicken feathers, for example. As mentioned above, adding some high-rigidity fillers like sand, clay, cullet, grog, or eggshells may improve the rigidity of the resulting material, just as it would a filled plastic system.

(Needless to say, such composite materials could be used for things other than sandwich panels, too, or even panels.)

Glues

How can you do the lamination, so the facesheets can't slide along the surface of the core?

You might be able to use just standard modern adhesives: epoxy, melamine, and phenolic resins. But these are entirely impractical to obtain from the waste stream, and there are cheaper alternatives.

Some materials can be bonded just with heat, and in some other cases you can use small amounts of thermoplastics to bond together layers of non-thermoplastic materials, like the EVA in a hot-glue gun. And, as mentioned above, latex paint works well enough under some circumstances, but it's difficult and somewhat dangerous to find discarded. Porous facesheet materials like cloth and window screen allow air access to the binder, which is essential for the curing of many binders.

In some cases the strength requirement for the adhesive may be low enough that even paraffin wax (or polyethylene or PET) could be used as a hot-melt adhesive, perhaps with a filler such as clay to stiffen it and add strength. Wet clay also makes an excellent weak adhesive which hardens and contracts on drying.

The five traditional adhesives are slaked lime, wheat paste, mucilage, hide glue, and birch tar, the first four of which can be made from garbage at varying levels of personal risk, or from very inexpensive non-garbage materials. They might work. Wheat paste and mucilage don't stick very well to nonpolar materials like common plastics, and slaked lime is caustic. Birch tar is probably not practical in the modern context. Most of these need some degree of care to keep them from rotting, blue vitriol being a traditional pH-neutral antifungal.

Blood is also sticky and dries hard, due to its albumin content, and there's a lot of animal blood in the waste stream, but for whatever reason it isn't widely used as an adhesive, perhaps due to excessive clotting.

"Pressure-sensitive" or "constant-tack" adhesives like those used in duct tape, packing tape, and postit notes are not really suitable for structural use, because, being viscous liquids, they will flow continuously if under strain. They can resist compression, but in shear or, especially, tension, they will gradually release their bond. In structural applications this is potentially disastrous: weeks, months, or years after initial construction of a structure, its sandwich panels could delaminate and fail, apparently spontaneously.

Asphalt or pitch, as used in roofing membranes, is on the edge here. It's a viscous liquid, but at room temperature its flow rates are low enough that it might be acceptable, if the panels won't be exposed to

heat.

Thin-enough porous facesheets could be simply painted with boiled linseed oil, but oxygen needs to be able to diffuse to the facesheet-core interface to be able to polymerize the oil there. More generally, it might be possible to polymerize such "drying oils" with oxidizing agents like sodium percarbonate or ozone. (Linoleum flooring is polymerized linseed oil with a filler painted onto a coarse cloth backing.)

Some kinds of core or facesheet materials could be welded with a small amount of solvent, such as acetone, MEK, or ethyl acetate. Doing this on styrofoam is tricky because the solvent tends to eat the cells it touches; perhaps a little vapor, rather than drops of liquid, would do the trick.

Such solvents can also be used to make glues from other waste plastics, such as PVC pipe or polystyrene foam, peanuts, or food containers.

A final, more exotic, glue is sodium silicate, which can be made in bulk very inexpensively, simply by boiling quartz in lye; this has the benefit of hardening on timescales measured in seconds when gassed with CO_2. As I mentioned above, this is commonly used to make corrugated cardboard; it also waterproofs concrete. (Potassium silicate is nearly equivalent.)

Glueless lamination

Since the objective of gluing the lamination interface is resisting shear, not tension, the facesheets could also be connected to the core by non-adhesive means, such as nails, tacks, staples, or sewing — a thread or wire passed repeatedly through holes in the sandwich panel to keep the facesheets from sliding relative to the core. These approaches are more practical with fairly rigid facesheets, since in between the mechanical attachment points, the facesheet on a side in compression must resist buckling on its own without any real support from the core.

Topics

- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Independence (p. 3520) (63 notes)
- Mechanical things (p. 3569) (45 notes)
- Garbage (p. 3468) (10 notes)
- Textiles (p. 3745) (4 notes)
- Cardboard (p. 3366) (3 notes)

Minimal transaction system

Kragen Javier Sitaker, 2017-09-21 (5 minutes)

A minimal transaction processing system

(See also A minimal dependency processing system (p. 911).)

The data store is a key-value store that supports set(k, v), get(k) -> v, and getNextAfter(k) -> (k, v) operations; keys and values are arbitrary bytestrings, and keys are ordered lexicographically.

You start with a transaction running as an isolated virtual machine. It has the above-mentioned data-store operations available, plus sbrk(n), commit() (which exits), fail(), and fork(), which commits the current transaction and creates two new coequal transactions with copies of the same memory, though one is running a different subroutine.

It starts by reading through a section of the key-value store (perhaps `"tx/" < k && k.startswith("tx/")`) where it expects to find source code for transactions to run. For each key of source code found, it fork()s a child transaction that reads that source code, compiles it into memory, and starts running it. When it finds no more source code, it invokes fail().

When a transaction invokes fail(), its writes do not become visible to other transactions, but the set of get() and getNextAfter() calls it made is remembered. If the data store changes in a way that would change what those calls saw, the transaction is retried. The system has the option to transparently checkpoint the transaction at any point and rerun it from there, rather than from the beginning — this is why fork() commits, because the child transactions are externally visible side effects.

These initially compiled transactions are mostly "servers". They look for messages in some section of the key-value store and, like the initial transaction, fail() if nothing is there. But if something is there, they spawn a child transaction to process it (and then fail if it's still there, to avoid repeatedly processing the same thing repeatedly.)

A transaction that has not yet committed will also be pre-emptively fail()ed by the system if it has previously read data that has now changed — when it attempts to commit(), if nothing else.

Initially I thought that after fork() the parent transaction should continue as before, but then I realized that it might fail, so either fork() needs to commit, or the child transaction needs to be nested, which would prevent writing servers. Then I resigned myself to the idea that, since the server must commit on every iteration, it wouldn't be automatically restarted if its source code changed.

Now, though, I realize that it is possible to have my cake and eat it too: ancestor transactions' read sets can be included in your read set, so if your source code changes, you can be instantly rolled back and recompiled, even though your committed ancestor transactions won't be rolled back. I think. I'm not sure how to make that work with message queues yet.

This form of IPC implicitly and modularly supports a wide variety of different communication patterns and both batch and interactive computing. If you want to wait for a message at either location a or

location b, you do this:

```
a = get("a");
b = get("b");
if (a == null && b == null) fail();
```

If you instead want to wait until both messages are present before proceeding, you do this:

```
a = get("a");
b = get("b");
if (a == null || b == null) fail();
```

If you want to get a message if one is present, but continue anyway if one is not, then you can handle nullness by a method other than failure:

```
c = get("key");
if (c != null) { /* handle keypress */ }
```

Because it doesn't support nested transactions, you can't directly turn blocking subroutines into nonblocking ones as with the orElse operator in the Composable Memory Transactions paper; if you want to be able to make a failure blocking or nonblocking, you should return that failure up the stack to the point where you actually want to make that decision. For example, the first example above might read as follows, assuming a context where a successful return value cannot be null:

```
a = get("a");
b = get("b");
if (a == null && b == null) return null;
```

You could spawn off a blocking subroutine into a child transaction that stores its result back to you somewhere that you're looking for it in a nonblocking fashion, but because fork() commits, this interferes with modularity.

This system is very simple, with only seven system calls (three of which take no parameters), full ACID transactions, and no explicit deadlock, although of course you can arrange communication patterns to provide deadlock. It should be possible to implement it extremely efficiently.

The definition of the store as a simple byte-blob key-value store is not essential to the system; it could have nearly any form whatsoever — relational, hierarchical, with private namespaces, graph-structured, message queues, tuple spaces, seekable files, a distributed transactional store, typed values, whatever. The byte-blob key-value store is just a minimal approach.

Topics

• Programming (p. 3658) (286 notes)
• Systems architecture (p. 3691) (48 notes)

Rsync message base

Kragen Javier Sitaker, 2019-11-08 (updated 2019-11-30) (29 minutes)

I was discussing eventual content synchronization for disconnected operation through eventually-consistent data stores for computer-mediated communication systems with an entity known as The Doctor, and they suggested the use of the rsync algorithm and its variants, which I thought was really interesting.

As it turns out, the rsync delta-transfer algorithm offers an interestingly scalable approach to synchronizing eventually-consistent data stores like these, one that has not been exploited previously to my knowledge.

Broadcast and flooding

The basic problem to solve here is how to distribute each of a large number of messages originating from many different sources to each of a large number of subscribers. For example, you might want every reader of alt.tasteless to be able to see all of the recent postings from anyone to alt.tasteless, or everyone who joins #hottub to be able to see all the messages someone sends to #hottub after they join, or every host on your Ethernet to be able to see every ARP request any host broadcasts on that Ethernet. Moreover, rather than achieving this through a centralized bulletin board or a shared broadcast medium like thinnet, you want to achieve it via some potentially complex and changing topology of interconnected network nodes (originally, "gateways"; nowadays sometimes "routers").

(For the time being I will ignore the questions of multiple channels, unicast messages as well as broadcast, whether the subscribers and the publishers are the same set, whether there are some security requirements on their membership, and whether there's some kind of need for load-balancing or "sharding" for performance.)

The obvious thing to try is for each node to send each message it receives to all of the other nodes to which it is connected. So if you have the topology A -- B -- C -- D; B -- E, then if node A sends a message, it will arrive at node B, which will forward it to nodes C and E; node C will then forward it to node D; and we're done. This simple flooding approach has one main problem: first, any cycle in the connectivity graph will produce an infinite number of copies of every message.

I know of four approaches to solving this problem: TTLs and return-paths, the spanning-tree protocol, the Usenet message-id approach, and the per-publisher log approach used in Kafka. And I think the rsync approach might offer some interesting benefits.

TTLs and return-paths

The most basic approach to limiting broadcast message duplication by routing loops is a "time-to-live" field on each message sent across the network. This field is decremented at each router, and messages with TTL of 0 or less are discarded. So a message originally sent with a TTL of 5 is guaranteed to reach a distance of no more than 5 routers from the sender, and a message originally sent with a TTL of 10 is guaranteed to reach a distance of no more than 10 routers from the

sender.

This is a crude approach, but it does at least guarantee that the number of duplicates of each message is *finite*. Consider the topology A -- B -- C -- A. If A sends a message with a TTL of 30, they will initially give a copy to both B and C. One copy will travel clockwise around the loop 10 times, while the other will travel counterclockwise around the loop 10 times; each node will receive the message (and every other message) 20 times.

But the number can be exponentially large. Consider the "diamond graph" A -- B -- C -- A; B -- D -- A. Every time a message reaches B or A, two copies of it will be made; for a copy arriving at B from D, for instance, copies will be sent to C and A, and A will then send copies to C and D. So a message with a TTL of 5 sent from A will go through the following population stages:

- B 1; C 1; D 1.
- B 2; C 1; D 1.
- A 4 (2 from B, 1 from C, 1 from D); C 2 (from B); D 2 (from B).
- A 4 (2 from C, 2 from D); B 2 (from A); C 3 (from A); D 3 (from A).
- B 10 (4 from A, 3 from C, 3 from D); C 4 (2 from A, 2 from B); D 4 (2 from A, 2 from B).

And at that point it stops because the TTL was 5. But it should be clear that even this simple graph with four nodes can produce exponential growth of the message population to some exponential function of the TTL with the simple flooding algorithm, making the TTL approach of limited effectiveness. This means that, at best, the network will perform very poorly, and very likely will fail badly under load.

A slight refinement of this approach was used on Usenet (though not as the primary duplicate-suppression mechanism); each copy of a message carries a "return path" that describes the path it took to get to where it is, which could typically be used as an email routing path to send a reply to the poster. The obsolete SSRR option in IP and the obsolete multihop mail routing system in SMTP work the same way (@foo,@bar:baz@quux, I think was the SMTP notation). This can serve to suppress duplicates due to routing loops because if a message received at node X has node X in its return path, then it is obviously going nowhere useful and should be dropped. This successfully suppresses exponential message growth in simple networks like the diamond graph above, but not in larger networks. Consider, for example, A -- B -- C -- D -- E -- F; A -- C -- E; B -- D -- F. A message injected at A goes through the following stages of evolution:

- B!A, C!A
- B!C!A, C!B!A, D!B!A, D!C!A, E!C!A
- B!D!C!A, C!D!B!A, D!B!C!A, D!C!B!A, D!E!C!A, E!C!B!A, E!D!B!A, E!D!C!A, F!D!B!A, F!D!C!A, F!E!C!A
- B!D!E!C!A, C!E!D!B!A, D!E!C!B!A, D!F!E!C!A, E!C!D!B!A, E!D!B!C!A, E!D!C!B!A, E!F!D!B!A, E!F!D!C!A, F!D!B!C!A, F!D!C!B!A, F!D!E!C!A, F!E!C!B!A, F!E!D!B!A, F!E!D!C!A
- B!D!F!E!C!A, C!E!F!D!B!A, D!F!E!C!B!A, E!F!D!B!C!A, E!F!D!C!B!A, F!D!E!C!B!A, F!E!C!D!B!A, F!E!D!B!C!A, F!E!D!C!B!A

Then it ends, because every return-path contains all six nodes, so there is nowhere else for any message to go. It should be clear that, although the amount of traffic generated on the network by this single message is finite, it is already large and grows exponentially with the size of the network because the number of simple paths by which a message can reach each node grows exponentially with its distance from the message origin. Node F got 13 copies of the message.

The spanning-tree protocol

In order to achieve very high message rates without the above kinds of explosive message duplication, IRC and networks of Ethernet switches use essentially the simple flooding approach described above. To avoid the disaster of infinite message multiplication, they maintain a strict spanning tree among the nodes: only enough links are active to achieve full connectivity, deactivating any links that would create cycles.

The topology of the network is assumed to change on a much slower timescale than the transit time of individual messages. Any message that is transiting the network when the topology changes may be lost or duplicated, so reliability and semantic deduplication must be done by network endpoints; if a message's transit is slow enough to include many topology changes, it may be duplicated many times.

Moreover, for the topology to avoid containing many cycles all the time, the interval between topology changes must be large compared to the end-to-end *latency* of the network, because activating two new links on opposite sides of the network may form a cycle which requires the deactivation of some link once detected. If the latency is large, this situation will go undetected for a long period of time. However, the same large latency will also limit the number of messages thus duplicated.

This approach is not very suitable for networks like FidoNet and UUCP, which in their heyday had end-to-end latencies typically on the order of a week, organized around daily modem telephone calls, and each of whose links might or might not function on any given occasion. So Usenet used a different approach for duplicate suppression.

The Usenet message-ID approach

The Usenet approach is to identify each message with a "Message-ID" unique to that message, but much shorter than the entire message. Then, before transmitting the entire message from one node to another, the communicating nodes verify that the message isn't already present on the receiving node; the Usenet protocol NNTP has commands called IHAVE and SENDME for this purpose. `IHAVE foo` indicated the availability of the message with the message-ID `foo`; `SENDME foo` requested its transmission, if possible. In the Bitcoin protocol, the `inv` message lists data blocks, block headers, or mempool transactions, like a bulk IHAVE; and the `getdata` message plays the role of `SENDME`.

A serious weakness of the Usenet implementation was that the message-ID is chosen by the sender, typically a string something like "trn.20190822.1830.10831@canonical.org", incorporating the

hostname, the date and time, the software being used, the PID of the running process, and so on, in an effort to avoid accidental duplication. Nevertheless, bugs did sometimes result in unintentional duplication, and people sometimes engaged in intentional duplication to attempt censorship.

Git uses a similar approach, but uses the SHA-1 of the "objects" as the message-ID rather than a sender-computed string. It is conjectured to be computationally infeasible to produce a different object with the same SHA-1, even intentionally, much less accidentally.

I think FidoNet used this approach to propagate messages in its "echos", which were distributed message bases similar to Usenet newsgroups, but somewhat more primitive. The Doctor tells me that the message-IDs FidoNet could use to avoid unlimited duplication of messages were tuples of the form (network, zone, region, board, sub, message), where the (zone, region, board) was a hierarchically assigned numerical address space uniquely identifying the particular bulletin board on which the message originated, and "network" presumably distinguished FidoNet proper from other networks that might use the same protocols.

(However, I'm not as familiar with FidoNet's protocols as I am with the internet protocols, and so I might have gotten that wrong.)

A weakness with the message-ID approach is that it still scales only linearly with the number of messages. If you have a billion messages comprising ten terabytes, each with a 20-byte message-ID, then an IHAVE command for each message-ID will still cost 28 gigabytes of network bandwidth in every conversation, even if only one or two new messages are to be transmitted. There are protocols involving more round trips, such as ping-pong breadth-first trie traversal and some approaches using compressed Golomb sets or Bloom filters, that can reduce this cost by a significant factor, but still only a linear factor.

Evidently some way of naming large, coherent groups of messages is needed if we are to get the desired superlinear speedup.

The per-publisher log approach

Kafka is a distributed modern publish-subscribe system used in high-bandwidth data-center environments. The way it works is that each new message is apended to a log and assigned an ordinal sequence number in that log; subscribers send requests not for individual messages but for ranges of ordinal numbers in a given log. Each subscriber remembers the ordinal number of the last log message it has seen on a given log, and when it loses a connection and reconnects, it requests the next, say, 100 messages after that point. This frees the server from maintaining any persistent per-subscriber state, allowing it to scale both to large numbers of messages per second and large numbers of subscribers.

This protocol permits a subscriber to efficiently mirror the log, if it wishes. It can then provide the same subscriber interface to other subscribers, as long as only the origin server is assigning new ordinal numbers to new messages. In fact, it can update its mirror from other subscribers in the same way; it doesn't need to talk to the origin server directly. (Kafka itself doesn't take advantage of this possibility, as far as I know.)

I think this is the way Secure Scuttlebutt works, as well. Each participant in the chat has their own append-only log of messages that it has published, and upon conversing with a peer, it asks for updates to the logs that it is a subscriber to.

Van Jacobson's "Content-Centric Networking" project (a generalization of which is known as "Named Data Networking", or NDN) uses this approach to handle streams of data.

In CCN, routing is done by naming pieces of data, not network nodes. Each router remembers some set of interests associated with its network links and some set of messages; it exchanges interests and messages with its peers. When it sees a message whose identifier matches an interest it has pending, it forwards the message to the router from which it got the interest, and if it receives an interest that matches a message that it has stored, it replies to the interest with a copy of the message. In either of these cases, it forgets the interest, since it has been satisfied.

On the other hand, if it receives a new interest that it cannot satisfy, it remembers it and uses some algorithm to choose which network links to forward the interest on to, perhaps related to which network links it has received similar messages on before, or some kind of hierarchical network addressing scheme and dynamically updated routing table. The interest being forwarded through the network leaves a path of backpointers which give a route back to the original requester, without that requester needing any kind of network address.

In this way, messages are only forwarded to routers that have requested them, eliminating many opportunities for denial-of-service attacks, and one copy of a message going into a router can evantually result in many copies flowing out of it, as if the router were a caching HTTP proxy, eliminating many other opportunities for denial of service.

The obvious question is how to handle things like streaming voice and video in a system like this: interests in data that doesn't yet exist. The answer is simply that you assign sequence numbers to the frames of streaming data ("ElRubius/videostream/d8s0g03402e/frame/3302") and the subscribers send interests for some window of sequence numbers that have not yet been produced, but which will be delivered to them when they have. As long as the window is large enough to compensate for the latency of propagation of new interests, this will result in immediate and efficient streaming.

This works out to be precisely the same per-publisher log protocol used by Kafka for the analogous problem. And in some sense the `getblocks` message in the Bitcoin protocol does the same thing --- but the "publisher" is the Satoshi consensus of the participating nodes, so blocks might sometimes be superseded.

Cryptographic authentication of messages in the log is useful in some cases to prevent one publisher from interfering with another. In the message-ID IHAVE/SENDME protocol this could be done simply by using cryptographic hashes as message-IDs, as Git does, but in the log-appending protocol, some different approach is needed; for example, each new message in the log could be signed with a private key associated with that log. Such an approach will only be successful, however, if the routing nodes in the system are checking

the signatures, which is a potential scalability bottleneck.

How does Git's protocol actually work? Originally it used rsync, but not the rsync delta-transfer algorithm mentioned below. Now it has "the 'smart' protocol" (better documented in the official documentation and also the v2 protocol. I don't really understand how this works; it has `have` and `want` messages but I'm not clear on how many of them need to be sent but it sounds like it uses the DAG structure to avoid sending all of them.

This per-publisher-log protocol is only more efficient than the per-message ID IHAVE/SENDME protocol as long as the set of publishers remains small, which, admittedly, covers many important cases. But if each message has a new publisher, it reduces to the per-message-ID algorithm.

Approaches based on rsync's delta-transfer algorithm

Rsync contains a delta-transfer algorithm designed to save bandwidth over Australia's undersea cables. Transmitting data to or from Australia was very expensive, so if you had slightly different copies of a file on opposite sides of the Pacific, it was important to find the parts that were different and transmit only those. If you have both versions of the file on one side of the connection, you can use the standard longest-common-subsequence ("LCS") dynamic-programming algorithm to find the minimal edit sequence. But how do you efficiently compute a small edit sequence between two files, each of which is only available to one of the parties in the protocol?

The simplest approach is of course to divide the file into blocks of some fixed size and use the message-ID approach, using the SHA-256 or whatever of each block. This would work well for files that are modified by overwriting some part of the middle of the file; only the modified parts will have a different SHA-256, and so only those modified parts (plus the rest of the blocks containing them) will be transmitted.

But if you insert a byte at the beginning of the file, shifting the rest of the data in the file by one byte, none of your hashes will match, and so the entire file will be transmitted even though the edit distance was one byte.

The bupsplit algorithm used by Avery Pennarun's bup backup program, and also the basis of Jumprope, attempts to overcome this problem by breaking the file into blocks of variable sizes in a way that will usually be consistent after insertions and deletions, similar to the "fuzzy hashing" used in forensics. (See Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) for some related notes.)

The rsync algorithm takes a different approach. One of the versions of the file is broken into fixed-size blocks in the usual way, typically using a block size of a few hundred bytes, and each block is hashed with two different algorithms: a weak linear rolling-checksum algorithm (in rsync, a modified version of Adler32) and a stronger hashing algorithm — originally rsync defaulted to MD4 for this, which was cryptographically very weak, and even nowadays uses MD5, which has also been broken. The resulting collection of

hashes (let's call it a "digest", since the rsync papers don't give it a name) is transmitted to the other participant, where the rolling checksum is computed over *every length-N substring* of the other version of the file; any matches found in the digest are checked with the strong checksum. This allows the relatively efficient and precise computation of the byte-ranges of either file that are present at any offset in the other, as long as the shared data is more than a block in length.

It is interesting to note, as Andrew Tridgell does in his dissertation, that in some cases the rsync algorithm finds smaller deltas than the LCS algorithm used by diff(1), because rsync can detect and take advantage of transpositions, while LCS cannot.

The rsync algorithm is used not only in rsync but also in zsync, rdiff, and some other software. zsync in particular allows a sender-server participant in the protocol to be nothing more than a dumb HTTP server capable of byte-range access; this is achieved by precomputing the digest and placing it in a "zsync file" on a web server that points to the real file. The zsync client, upon fetching the digest file, can run the rolling checksum over its local version, occasionally running the strong checksum, and compute the set of byte-ranges that it needs to fetch from the origin server to reconstruct the origin server's version of the file.

If you want to rsync a mebibyte of data using a block size of 4 kibibyte (Tridgell's dissertation discusses block-size tradeoffs in chapter 3, finding optimal block sizes in the range of 256 bytes to 8 kibibytes for a few datasets), the digest to be transmitted will be 5 kibibytes, 0.5% of the total.

Note, however, that this 0.5% doesn't decrease as the file size increases, unless you also increase the block size. If you were to digest 10 tebibytes using 4-KiB blocks and 20-byte digest entries, your digest would be 50 gibibytes.

As a simple intermediate step between the IHAVE/SENDME system and the log-appending system, you could imagine using some variant of the rsync protocol on a document containing the concatenation of all messages in some well-defined order. In effect, this assigns a message-ID to each entire block of messages, rather than each individual message.

A key difference from the usual use of rsync is that the receiver don't want to delete messages that the sender doesn't have from their own database; instead they want the union of all interesting messages.

For this to be efficient, you want the ordering chosen for the messages to make the likely updates somewhat local, in the sense that they leave large chunks of the file untouched. For example, you could order the messages in the file temporally, so that new messages are usually added near the end, or by a combination between temporal order and publisher ID, or a combination of temporal order and topic.

This approach also permits participants, in theory, to blacklist certain known blocks to save space — rather than storing a tebibyte of uninteresting data (last year's Wikipedia edits, say), they can just store its hashes and its sorting key range. However, if new data appears that belongs to that sorting key range, it would change the hashes, making the simple blacklist approach fragile.

A potentially more interesting approach is to store the ranges of sorting keys, or at least their longest common prefixes, in the digest along with the hashes, permitting participants to choose which subrange of the keyspace they bother to replicate.

Recursive rsync delta transfer

Suppose that instead of using a single block size, we use several different block sizes on the same file. For example, we compute digests for block sizes of 1 KiB, 1 MiB, 1 GiB, and 1 TiB. If our total dataset is 16 TiB, its 1-TiB-level digest might be 320 bytes (assuming, for now, no sorting keys — just treating the file as opaque); a peer who fetches that digest can efficiently discover whether it matches their local replica, or matches it except for a few bytes inserted at the beginning.

But suppose they find that the last TiB-sized block in the 1-TiB-level digest doesn't match any of the 17.59 trillion overlapping tebibyte-sized blocks in their own replica. Rather than sending a network request or a purchase order to have that tebibyte of data shipped to them, they can fetch the corresponding block of the 1-GiB-level index. The entire 1-GiB-level index has 16384 entries, but it's only interested in the last 1024 of them, totaling 20 KiB, to discover whether any of the gibibytes comprising that tebibyte are among the 17.59 trillion overlapping gibibytes in their existing dataset.

Perhaps all but one of those gibibytes is a known gibibyte; in this case it can recurse down to the mebibyte level, and then down to the kibibyte level.

In this way, if anywhere from 1 to 1024 bytes have been inserted or deleted in any single place in this 16-TiB dataset, our peer can discover them by transfering 320 + 20480 + 20480 + 20480 + 1024 = 62784 bytes. This is what rsync would report as a "speedup factor" of about 280 million, although it's still worse than the theoretical limit by a factor of between 61 and 62784. Note that this is amenable to zsync's digest-precomputation approach.

The overhead in the worst case is 20 parts in 1023, or 1.96%, the same as nonrecursive rsync. But there are important cases that should admit these higher efficiencies.

Storing the file in such a way that this can be done quickly, including a summary of the 70 trillion rolling hashes involved to avoid needing eight passes over the 16-tebibyte dataset, and the desire to keep a local "virtual copy" of the sending peer's dataset (to avoid re-transferring blocks the next time around whose only sin was that they lacked a message, rather than having new ones) seems like it might be a challenging problem both in terms of algorithms and in terms of systems design. However, I think it's in some sense straightforward; it doesn't require any novel inventions.

Recursive rsync delta transfer applied to message bases and similar CRDTs

Suppose we have a nearly-16-TiB data store and we append one message to it, a message of under 1024 bytes. This can be synchronized with the 62784 bytes mentioned above. Once we bump past the 16-TiB line things get even a bit better still: 340 + 20 + 20 + 20 + 1024 bytes, since all the recursion levels except the top one only

contain a single hash.

This is considerably better than the 50 gibibytes required for non-recursive rsync or the 28 gigabytes I suggested the IHAVE/SENDME approach would need for a similar-sized base of messages (although there I was postulating an average message size of 10 kilobytes). But it remains efficient if we have, say, a mebibyte of new messages to sync. If they're scattered in 1024 random places through the 16-tebibyte base, due to a poor choice of sorting keys, we need on the order of 63 mebibytes of bandwidth to sync them, a 63x multiplier, but several hundred times better than the other protocols. If, instead, they are gathered together more or less in one place, we need to transfer 320 + 20480 + 20480 + 20480 + 1048576 = 1110336 bytes, an overhead of about 6%.

Topics

Archival of hypertext with arbitrary interactive programs: a design outline

Kragen Javier Sitaker, 2018-11-09 (3 minutes)

The pages you look at are deterministically executed code drawn from a content-addressable store, using indirections through a namespace. (So far, this is very much like Git.)

Code execution, since it's deterministic, can have its outputs cached. Ultimately what you need to see is a pixel image, so that's the product of the final bit of code. But it's working from a variety of inputs, and intermediate steps in the process can be cached.

The interaction with a page is structured more or less as in Redux: a state is built up by applying a set of reducers to a sequence of user interface events, and what you see on the screen is a function of that state and stuff drawn from the content-addressable store. In this case, the user interface events are ultimately just keystrokes, mouse events, and touch events. But these are transformed into higher-level events such as scroll events, button clicks, and text changes, according to a sort of previous document state.

This approach allows a very small, stable computational core to be extended to arbitrary interactive documents.

Because the code's interaction with the files it draws upon (images, libraries, etc.) is indirected through a namespace, you can get new results from the same code by runnning it in a new namespace. This means that, for example, if you want to view an existing document with a new font, you can make a modified version of it with the new font in the relevant place in its namespace, where it used to find the old font.

The blobs in the content-addressable store are simply sequences of bytes. The code's access to items in its namespace is by way of memory-mapping, and it has the option to view those sequences of bytes in a variety of ways, including as arrays of integers. The code's output is, similarly, some set of blobs placed in a particular part of its namespace, found there when its transaction terminates successfully. This means that many jobs can be performed without undergoing the overhead of serializing and deserializing data structures.

Jobs cannot map new blobs into their namespace except by supplying their contents; thus they cannot access any data they are not initially granted access to, nor can they delegate such access to other jobs. This means that you can statically determine the entire transitive dependency set of not only a given job but also all jobs that could be launched from it via UI interaction. This makes it possible to securely archive the data necessary to run it, which means that you can run it without risk of a failure due to data not being available, and also means that you cannot

Topics

Dutch auction raffle

Kragen Javier Sitaker, 2018-06-05 (3 minutes)

A raffle, like a tournament or a dollar auction, is a system designed to sell things for more than they are worth, manipulating the buyers by setting their incremental incentives against them. But the raffle tickets must have some price set on them — if the number of tickets is fixed, then this price could be set too high (reducing the total price paid by selling too few tickets) or too low (reducing the total price paid by selling out at too low a price). Making the raffle open-ended does not eliminate this problem, although it does reduce it.

It occurred to me that a Dutch auction for the raffle tickets might be a useful variant — all of the tickets are sold at the lowest price that sells all of them. This provides an incentive to buyers to reveal their true value.

An open-ended alternative would sell a previously undetermined number of tickets at the price that generates the largest amount of revenue. For example, given bids for 10 tickets at $10, 20 tickets at $20, 20 tickets at $30, 10 tickets at $40, and 1 ticket at $60, you could sell 61 tickets at $10, 51 tickets at $20, 31 tickets at $30, 11 tickets at $40, or 1 ticket at $60, with revenues respectively of $610, $1020, $930, $440, and $60. So selling 51 tickets at $20 is the best option, leaving out those who would only have bid $10, giving all others equal winning chances of 1.96%.

I'm not sure what the incentive landscape looks like for buyers. Suppose the raffled good is worth $1000 to you, and there's a closed-end Dutch auction for 100 raffle tickets. If you're risk-neutral, you should be willing to buy all 100 at any price up to $10 each, but none at any higher price. If it's worth $1100 to another risk-neutral rational actor, they would bid $11 each for all 100; according to the rules of Dutch auctions, this results in them getting all 100 tickets at $10 and you getting nothing. So far so good, since you also paid nothing.

But, if you're truly risk-neutral, you should be just as happy to pay $500 for a 50% chance of acquiring it as to pay $1000 for a 100% chance, so all numbers of tickets are equally good to you, regardless of whether your true value is $1000 or $1100 or $2000 or what.

All of this seems okay so far, except that it doesn't have the intended perverse incentive of inducing buyers to pay more than they get in return, as I think raffles are observed to do in the real world.

We could attempt to explain this by supposing that some buyers are risk-positive, so that a 50% chance of acquiring a $1000 good is worth more than $500 to them. But there are a large variety of different ways that someone could be risk-positive.

Topics

- Economics (p. 3424) (33 notes)
- Strategy (p. 3734) (10 notes)
- Incentive design (p. 3516) (5 notes)

Dercuano grinding

Kragen Javier Sitaker, 2019-10-01 (12 minutes)

Right now all the code blocks in Dercuano are rather plain: black on gray, with one typewriter typeface in one size; I think some syntax highlighting would make it both more *pleasant* to read and more *inviting*. The contrast between the colorful source listing I see as I'm editing notes in Emacs and the dull gray listing I see in the rendered HTML is depressing.

The standard way to syntax-highlight code for the WWW is to run something like Pygments or Emacs `htmlfontify-buffer` on the server side to generate a passel of HTML spans that refer to some stylesheet, but in part since Dercuano doesn't have a server side, that isn't really an option, for the same reason prerendering of PNGs is not an option (see Dercuano drawings (p. 64)) — the resulting files are huge.

So, how can I syntax-highlight code in Dercuano? Presumably I need to write something in JS that can be instructed to attempt to tokenize code blocks with the lexical syntax of various languages, perhaps even parsing them, and apply spans and styles to them based on the results.

Languages: mostly Python, SQL, and very minimalist languages

First I thought it would be good to take a sampling of the syntax-highlightable languages I'm using in Dercuano.

Phase relations (p. 2200): none
Harvesting energy with a clamp-on transformer (p. 1952): none
Query evaluation with interval-annotated trees over sequences (p. 1423): SQL, Python, SQL fragments
Cheap textures (p. 736): none
Tagged dataflow (p. 405): none
A bag of candidate techniques for sparse filter design (p. 3250): Python
Flexures (p. 2211): none
Notes on Raph Levien's "Io" Programming Language (p. 1740): Io, Scheme, possible extensions of Io
Microfinance (p. 2875): none
Binate and KANREN (p. 3189): Binate
Midpoint method texture mapping (p. 1837): none
Forth with named stacks (p. 2101): possible extensions of Forth
Things in Dercuano that would be big if true (p. 3136): none
Spiral chinese windlass (p. 2915): ASCII art diagrams
Wang tile font (p. 1463): none
Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333): none

The main conclusion I can draw from this random sample of 16 notes is that there isn't much code in Dercuano (≈⅓), and what there is tends toward very minimalist programming languages (Scheme, Forth, Io, Binate) which in many cases don't even have existing implementations — and, also, Python and SQL. Languages I've recently used in other notes include SQL, Python, OCaml, Lisp

S-expressions, Lua, GNU MathProg, and Golang.

Syntactic categories and aesthetic treatment

Popular syntactic categories to highlight include, in more or less decreasing order of importance, keywords, function names, type names, string contents, numbers, comments, and punctuation.

Common practice is **to** emphasize **the** keywords, **but** that does **not** make any sense; **the** keywords are **the** lowest-information-density part **of the** program code, so that merely adds noise. A better approach might be to reduce the contrast of keywords, allowing the reader to focus on the informative part. Reducing font size (and increasing letter-spacing to compensate) might also help. Much the same can apply to punctuation, and while this is fairly unimportant for *comprehension* because it's already visually distinctive, it can be a major boon to *aesthetics*.

A recent post discussed on the orange website suggested coloring identifiers by a hash of their contents, so that `pcall` would look very different from `pcal1`, especially to non-colorblind people. If you also wanted to use color to distinguish function names, type names, and keywords, you'd need to slice up the color cube into distinct regions for each one with a demilitarized zone between them to permit reliable discrimination.

One possible significant exception to the dismissal of punctuation is that parenthesis-matching can improve the readability of Lisp code substantially. Traditionally it's done interactively (e.g., by bouncing on % in vi or using C-M-f and C-M-b in Emacs), but by assigning different colors and weights to different parenthesis levels, noninteractive parenthesis matching could be facilitated.

The available aesthetic attributes for syntax highlighting in modern CSS go far beyond what was traditionally available either in hot lead or in emulated VT340+ terminals. Even without using different typefaces, sizes, and letter-spacing, we can oblique, drop-shadow, change the background color, greatly vary the line width, underline (though be careful about underlining spaces), overline, strikethrough, compress the letterforms, and rotate the letterforms.

The vgrind program would call out definitions (for example, of functions and types) by putting the name of the defined entity in the right margin in a larger font, which was useful for paging through printed-out listings but probably isn't useful for a small snippet of code with commentary around it.

Where do I get the tokenizers or grammars?

It would be nice to avoid writing parsers for Python, SQL, OCaml, Scheme, MathProg, Golang, and Lua, especially fragment-tolerant and REPL-tolerant parsers, but there's probably no way to get around writing parsers for things like Binate, proposed extensions of Forth, proposed extensions of Io, and Io itself.

Pygments

Pygments comes with a fairly comprehensive set of lexers optimized for this task; `grep --color -nH -e ^class /usr/lib/python2.7/dist-packages/pygments/lexers/*.py` yields a bit over 400

matches. Most of these contain some imperative Python code, but Pygments has a pretty extensive set of regexp-based facilities:

```
class MoinWikiLexer(RegexLexer):
    """
    For MoinMoin (and Trac) Wiki markup.

    .. versionadded:: 0.7
    """

    name = 'MoinMoin/Trac Wiki markup'
    aliases = ['trac-wiki', 'moin']
    filenames = []
    mimetypes = ['text/x-trac-wiki']
    flags = re.MULTILINE | re.IGNORECASE

    tokens = {
        'root': [
            (r'^#.*$', Comment),
            (r'(!)(\S+)', bygroups(Keyword, Text)),  # Ignore-next
            # Titles
            (r'^(=+)([^=]+)(=+)(\s*#.+)?$',
             bygroups(Generic.Heading, using(this), Generic.Heading, String)),
            # Literal code blocks, with optional shebang

            (r'(\{\{\{)(\n#!.+)?', bygroups(Name.Builtin, Name.Namespace), 'codeb
lock'),
            (r'(\'\'\'?|\|\||`|__|~~|\^|,,|::)', Comment),  # Formatting
            # Lists
            (r'^( +)([.*-])( )', bygroups(Text, Name.Builtin, Text)),
            (r'^( +)([a-z]{1,5}\.)( )', bygroups(Text, Name.Builtin, Text)),
            # Other Formatting
            (r'\[\[\w+.*?\]\]', Keyword),  # Macro
            (r'(\[[^\s\]]+)(\s+[^\]]+?)?(\])',
             bygroups(Keyword, String, Keyword)),  # Link
            (r'^----+$', Keyword),  # Horizontal rules
            (r'[^\n\'\[{!_~^,|]+', Text),
            (r'\n', Text),
            (r'.', Text),
        ],
        'codeblock': [
            (r'\}\}\}', Name.Builtin, '#pop'),
            # these blocks are allowed to be nested in Trac, but not MoinMoin
            (r'\{\{\{', Text, '#push'),
            (r'[^{}]+', Comment.Preproc),  # slurp boring text
            (r'.', Comment.Preproc),  # allow loose { or }
        ],
    }
```

Even things like pygments.lexers.ml.OcamlLexer (90 lines of code) are written entirely in this declarative style without any executable code. As you can see above, Pygments apparently has a stack machine for nested multi-line string constructs such as Trac code blocks or OCaml comments.

One drawback of trying to use this stuff is that it relies pretty

heavily on Python's regexp syntax, which is mostly, but not completely, compatible with JS's.

Another is that syntax-highlighting purely through tokenization doesn't have much chance of, for example, consistently distinguishing user-defined types from user-defined functions.

Vim

Vim comes with some 586 syntax definition files; many of them are fairly minimal. They are written in Vimscript, which is imperative, but most of the commands consist of defining regular expressions — which I originally thought would be in Vim's regexp dialect with `\(\)` for grouping and whatnot, but which has apparently adopted many PCRE-like features — though the backwards-incompatible ones are turned on on a per-regexp basis by the sequence `\v`, which means "very magic". Here's Honza Pokorny's Dockerfile syntax file:

```
if exists("b:current_syntax")
    finish
endif

let b:current_syntax = "dockerfile"

syntax case ignore


syntax match dockerfileKeyword /\v^\s*(ONBUILD\s+)?(ADD|CMD|ENTRYPOINT|ENV|EXPOSE
|FROM|MAINTAINER|RUN|USER|VOLUME|WORKDIR|COPY)\s/

syntax region dockerfileString start=/\v"/ skip=/\v\\./ end=/\v"/

syntax match dockerfileComment "\v^\s*#.*$"

hi def link dockerfileString String
hi def link dockerfileKeyword Keyword
hi def link dockerfileComment Comment
```

Vim also does automatic indentation, which requires a little deeper understanding, but this is done entirely separately and with no dependence on the syntax highlighting.

Consequently, even in C, like Pygments, Vim doesn't manage to distinguish between types and functions.

Emacs

Emacs does successfully distinguish between types and functions, and it is configured in a similar way to how Vim is configured, but with Lisp lists instead of sequences of commands, and perhaps a few more levels of indirection. However, some of the Emacs syntax highlighting setup involves truly unbelievable amounts of hair.

Golang in Emacs

go-mode.el is relatively simple; one of the first things the `go-mode` function does is to set `font-lock-defaults` to a list containing the name of a function that returns a declarative configuration for syntax highlighting as a list:

```
(set (make-local-variable 'font-lock-defaults)
   '(go--build-font-lock-keywords))
```

This list is built up more or less as follows, though I've omitted most of the individual items:

```
(defun go--build-font-lock-keywords ()
  (append
   `(...)
   (if ...)
   `(...
     (,(concat (go--regexp-enclose-in-symbol "map")
               "\\[[^]]+\\]" go-type-name-regexp)
      1 font-lock-type-face) ;; map value type
     (,(concat (go--regexp-enclose-in-symbol "map")
               "\\[" go-type-name-regexp)
      1 font-lock-type-face) ;; map key type
     ...)))
```

The mysterious `go--regexp-enclose-in-symbol` function is defined as follows:

```
(defun go--regexp-enclose-in-symbol (s)
  "Enclose S as regexp symbol.
XEmacs does not support \\_<, GNU Emacs does.  In GNU Emacs we
make extensive use of \\_< to support unicode in identifiers.
Until we come up with a better solution for XEmacs, this solution
will break fontification in XEmacs for identifiers such as
\"typeμ\".  XEmacs will consider \"type\" a keyword, GNU Emacs
won't."
  (if (go--xemacs-p)
      (concat "\\<" s "\\>")
    (concat "\\_<" s "\\_>")))
```

The two particular items I called out earlier, which cause `foo` and `bar` in `map[foo]bar` to be highlighted as type names, come out as the following:

```
("\\_<map\\_>\\[[^]]+\\]\\(?:[*(]\\)*\\(\\(?:[[:word:][:multibyte:]]+\\.\\)?[[:wo
ord:][:multibyte:]]+\\)"
1 font-lock-type-face)

("\\_<map\\_>\\[\\(?:[*(]\\)*\\(\\(?:[[:word:][:multibyte:]]+\\.\\)?[[:word:][:mu
ltibyte:]]+\\)"
1 font-lock-type-face)
```

This says to highlight subexpression 1 in each of those regexps as `font-lock-type-face`, which begins at the first `\\(` that isn't followed by a `?:` — Emacs, too, has adopted some of Perl's regexp syntax, though not the good part of not having to double-backslash things or backslash your parens at all.

The whole syntax-highlighting list is about a page long, mostly regexps like that, but it also includes a call to `go--match-func`, which parses function parameter lists to look for type names. This involves

several pages of Elisp that does a lot of stuff like `(save-excursion (if (looking-at ...) (goto-char (match-end 0))))`, which is precisely the kind of thing I'd like to avoid, convenient though it sometimes is as a way to munge text.

C in Emacs

By contrast, C syntax highlighting (like various other features for C) is mostly handled by a heuristic, forgiving C parser consisting of over ten thousand lines of Elisp written over the last 35 years in cc-engine.el, which also handles Objective-C, C++ (including the Qt extensions), Awk, IDL, Java, and Pike. No test suite is evident. As to how it distinguishes type names from function names, I have no idea and I might not find out tonight even if I spent the rest of the night on it, but it does.

Writing them from scratch

An appealing alternative, and really the only alternative when it comes to programming language syntax variants that I'm exploring that nobody has ever implemented, is to write tokenizers and parsers from scratch. This is especially appealing if the parser can be used to actually interpret or compile the language as well as syntax-highlighting it, although of course if I'm just interested in the language's semantics and not its syntax I can quite reasonably just serialize to S-expressions, as in A formal language for defining implicitly parameterized functions (p. 144), or just RPN, like Python pickle. But if I want compatibility with existing code or existing implementations, parsing arbitrarily complicated languages may be useful.

For interpretation, though, handling incomplete or incorrect code isn't necessary, and sometimes for syntax highlighting it is, though less often than for text editing. One possibility for that kind of thing is to write a normal parser using some kind of parsing theory (such as an Earley or Packrat parser) and then mechanically transform it to produce possible parses of *substrings* of the original language. (This approach is also useful for parallelizing and incrementalizing parsing; see Parallel NFA evaluation (p. 2967) for details.)

Another possibility, maybe a more interesting one, is to train a recurrent ANN to classify syntactic elements instead.

Topics

- Programming (p. 3658) (286 notes)
- Syntax (p. 3738) (28 notes)
- Dercuano (p. 3406) (16 notes)
- Compilers (p. 3383) (16 notes)
- Parsing (p. 3618) (15 notes)
- Typography (p. 3760) (5 notes)

User-per-group (UPG), umask, and “Permission denied” on shared Git repos via ssh

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

Maybe we're doing something wrong, because I can't find anybody writing on the web having trouble with git and umask, although it looks like the umask situation in Debian has been kind of a PITA for a while.

The Current State of the umask Problem

So I made the mistake of believing /etc/skel/.profile when it said:

```
# the default umask is set in /etc/profile
```

It turns out /etc/profile doesn't do squat when you do this:

$ ssh watchdog.notabug.com umask

because there's no login shell involved. It does read .bashrc, but having every subshell reset your umask would be annoying and maybe dangerous.

/etc/login.defs explains:

```
# UMASK usage [in /etc/login.defs] is discouraged because it
# catches only some classes of user entries to system, in fact
# only those made through login(1), while setting umask in shell
# rc file will catch also logins through su, cron, ssh etc.
#
# At the same time, using shell rc to set umask won't catch
# entries which use non-shell executables in place of login shell,
# like /usr/sbin/pppd for "ppp" user and alike.
#
# Therefore the use of pam_umask is recommended (Debian package
# libpam-umask) as the solution which catches all these cases on
# PAM-enabled systems.
#
# This avoids the confusion created by having the umask set in two
# different places -- in login.defs and shell rc files (i.e.
# /etc/profile).
#
# For discussion, see #314539 and #248150 as well as the thread
# starting at
# http://lists.debian.org/debian-devel/2005/06/msg01598.html
```

But for whatever reason, libpam-umask is not installed and used by default.

Consequently there are now a bunch of directories in /home/watchdog/code/.git that are writable only by "aaronsw", so I probably won't be able to push any more changes until we fix that.

Why This Problem Arose

Probably everybody knows, but just for the record.

Debian by default uses "UPG" or "User Per Group". This lets you leave your umask at 002 (or 007, or 000) all the time, and just use group ownership to distinguish private from shared files; private files belong to the group that contains only you.

http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/ref-guide/s1-users-groups-private-groups.html

But not all people, or all software, have adapted to this new world, so sometimes people like to have other umasks. This causes the problem we're seeing here: accidentally creating directories other people can't write to in shared spaces.

Why We Should Care

If we can get the infrastructure into a smoothly working state, we don't have to struggle with it later on. The opportunity cost of time at the start of a project is high, so it's hard to justify spending time on stuff like this up front, but the psychological phenomenon of hyperbolic discounting exaggerates that opportunity cost. I think putting in an hour or so on this stuff now will save us two or three hours over the course of the next few weeks.

http://en.wikipedia.org/wiki/Hyperbolic_discounting

Probably all of us have been burned a few times by not keeping our infrastructure in adequate order, so I expect this should be an easy sell.

What We Should Do

First, aaronsw or root should fix the nonwritable directories:

$ chmod g+w $(find /home/watchdog/code/.git -perm -200 ! -perm -020 -print)

Then we should fix the umask. Two choices:

- use libpam-umask. I've tested this on my server and it works:
  kragen@courageous:~$ sudo apt-get install libpam-umask
  Put this line at the end of /etc/pam.d/common-session: session optional pam_umask.so umask=002
  Remove the "umask 002" line from /etc/profile.
- Put 'if ["$SHLVL" = 1] ; then umask 002; fi' into /etc/bash.bashrc. Putting this in .bashrc does seem to work (I'm testing it on another server where other people didn't like the libpam-umask approach), so putting it in /etc/bash.bashrc should work too, as long as everyone uses bash.

Topics

- Unix (p. 3765) (7 notes)
- Git (p. 3474) (5 notes)

In what sense is e the optimal branching factor, and what does it mean for menu tree design?

Kragen Javier Sitaker, 2012-12-04 (3 minutes)

What's the sense in which e is the optimal branching factor? The number of nodes you have to traverse to reach one of N nodes with branching factor B is ceil(log N / log B), and if you're examining each of the B branches in the node to figure out which one to follow, you end up examining B ceil(log N / log B) branches. If we remove the discretization, we get log N (B / log B), and it turns out that B / log B has a minimum at B = e, so in practice the optimal branching factor is 3.

This still holds if you are only examining B/2 branches at each node. It's only if there's an additional cost to visiting a new node that the minimum cost moves to a larger branching factor.

To be a little more concrete, suppose you're trying to track down an incorrect result from the execution of a program that runs for a hundred billion instructions, or about a minute. If the program is built out of functions that call other functions and combine their results, and you can tell from looking at each result whether it's correct or not, maybe you can navigate to the result you want by expanding the execution history as an outline.

If each function calls three other functions, then within 24 clicks, you can make your way from the top-level result to the particular incorrect result; at each step, you have to examine on average 1.5 intermediate results, so you need to look at about 35 results.

If each function calls only two other functions, you have to look at about 37 results, which is slightly worse; and the same is true if each function calls four other functions. So the B/log B function is pretty flat over the region of 2–4. Then it starts to climb: at 5 callees, you need to examine 39 results; at 6, 42; and at 7, 46. At 15 callees, the number of results you have to examine to find your way to the bottom-level fault has doubled to 70.

However, the number of nodes you have to examine (and the number of clicks you have to make) at that point has shrunk from 24 to 9. So if the cost of visiting a node is significant (e.g. you need to reorient yourself in a new context, or clicking is slow) then the optimal number of callees for debugging might be higher than 3. Already at 9 you've cut in half the number of nodes visited (12 instead of 24) at only a 50% extra cost in number of branches examined. So if the cost to visit a node is comparable to the cost to examine a branch (the same order of magnitude) then the optimum is probably somewhere around 9, say, between 5 and 18. (At 18, you're examining 79 branches, but still visiting 9 nodes. So going from 9 to 18 saves you about 3 nodes, but costs you 27 branch examinations, which only makes sense if examining a node costs at least 9 branch examinations.)

Topics

Set hashing

Kragen Javier Sitaker, 2017-03-09 (9 minutes)

There is a constant-time way to hash the value of a mutable set container, which converts some algorithms from $O(N^2)$ to $O(N)$.

A mutable set container of T is an abstract data type whose value is, at any given time, some set of values of type T; typically these support constant-time member addition, member removal, membership testing, and iteration over the items contained.

Often these don't guarantee a particular iteration order because they are implemented, for example, using hash tables, which may shrink with some hysteresis to preserve the performance guarantee for iteration. The data structure Nayuki calls the Binary Array Set is a non-hash-table mutable set container structure which can be iterated much more efficiently in some arbitrary order.

I am trying to be careful in this essay to distinguish sets, which are immutable mathematical objects, from set containers, whose value at any given moment is a set, but which contain different sets at different times — the relationship is that a set is a possible state of a set container.

Hashing sets: a more efficient approach

In Python, *immutable* sets are hashable (if their items are), which is a frequently useful feature, but often inefficient — making an immutable set from what Python calls a mutable set (what I'm calling a mutable set container) involves iterating over all the items.

To make the values of a mutable set container efficiently hashable, XOR together the hash values of all of its items, then perform some one-to-one transformation that doesn't commute with XOR, such as adding or multiplying some large constant, or computing its SHA-256 or Blake2. This (except for the final transformation) can be done incrementally as items are added and removed, and it will always produce the same value for the same set of items, regardless of how the set was arrived at.

The final transformation here is to ensure that related sets such as {1, {2, 3}}, {1, 2, {3}}, and {{{1, 2, 3}}} all have different hash values, a feature which seems likely to be desirable in practice.

Doing it this way rather than the standard Python way will convert some $O(N^2)$ algorithms into $O(N)$ algorithms. If you add N items to a set container one at a time, each time testing a set of hashes to see if you may have seen the current state of the container before, this will take $\theta(N)$ time with this approach, but $\Theta(N^2)$ time with the Python approach.

16-bit example

As a 16-bit example, if your set is {4, 5, 8}, your hash function for integers is multiplying by 33947 mod 2^{16}, and your final transformation is adding the random number 48709 mod 2^{16}, then you compute $((4 \cdot 33947) \oplus (5 \cdot 33947) \oplus (8 \cdot 33947)) + 48709 \bmod 2^{16}$, so the set's hash value is 19970. If you store the quantity $((4 \cdot 33947) \oplus (5 \cdot 33947) \oplus (8 \cdot 33947)) \bmod 2^{16} = 36797$ in a variable in the mutable set container, and you add 10 to the set, you can

incrementally update it by XORing in $10 \cdot 33947 \bmod 2^{16} = 11790$ into it, getting 36797 as the new untransformed hash value; upon request, you can return it with 48709 added, giving 19970. If you then remove 4 from the container, you can XOR $4 \cdot 33947 \,\%\, 2^{16} = 4716$ into the variable, getting 40401, which you can verify is $((5 \cdot 33947) \oplus (8 \cdot 33947) \oplus (10 \cdot 33947)) \,\%\, 2^{16}$.

(The $\oplus$ character is standard math notation for the XOR operation.)

Minor variants

(Alternatively, you can sum the hash values of the items, then use a final transformation that doesn't commute with addition, such as bit rotation or XOR with a constant.)

If your item hash functions and final transformation are cryptographically secure hashes, then you would be justified in concluding that two states of the container are equal if their hashes are equal, without iterating over their items.

Possible uses for incrementally hashing sets

You might argue that it isn't very helpful for you to keep around a hash value for a no-longer-current state of a mutable set container — certainly it isn't useful as a way of finding the set container in a hash table, for example. And in the common case where you don't use a cryptographically secure hash, there's the chance of hash collisions.

However, there are uses for this.

One possible use for this technique is in incrementally optimizing NFA execution by partial compilation to DFAs (each DFA state corresponds to a set of NFA states), which can in the worst case produce a memory usage explosion — but if only state-sets that occur many times are compiled to DFA states, then the problem is much less severe. If you snapshot states that appear in a list (or Bloom filter) of previously seen hashes, then you can recognize them if they come around again.

Another possible use for this technique is in memoizing set arithmetic: if you have already computed the union, intersection, or set-subtraction of two sets, you may be able to avoid iterating over the set items to compute them again.

A third use is in implementing operations on finite binary relations represented as sets of pairs — operations which, in addition to the set-arithmetic operations mentioned above, might include composition, converse, transitive closure, and various kinds of products.

A fourth, very general, use is in combination with a transaction log. If you maintain a transaction log in a layer on top of a basic mutable set container that supports both insertion and deletion, listing the additions and deletions in the log, you can hash positions in the transaction log, and the state at each of these positions is retrievable in time bounded by the size of the transaction log. This means you can add N items to a set container one at a time, making a hash table of the N successive sets that are the states of the container, in $\Theta(N)$ time, and then retrieve any state from the table in $\Theta(N)$ time.

This kind of undo/redo journal approach is a very general and efficient approach for obtaining FP-persistent† semantics on top of

ephemeral data structures. If you bound the transaction log size, you can preserve constant-time guarantees for access, but then when the journal gets too large, you need to copy the whole data structure to a new one with a fresh journal.

In the scenario given, if the transaction log size is k, you can retrieve any state in at most k steps, but you need to copy the hash table (in the limit, of average size N/2) ⌊N/k⌋ times, requiring $N^2/2k$ work inserting and deleting. Depending on the relative frequency of probes and updates, and in particular the frequency of access to very old states, this approach can often save several orders of magnitude of performance — not an asymptotic improvement, to be sure, but good enough in many cases.

Note that the Binary Array Set structure mentioned above can be viewed as a special case of this approach, where the update log is sorted into increasingly larger arrays.

Other approaches to hashing states of mutable set containers efficiently

A different approach is to store the sets in a trie, or in a binary search tree with a convergent tree balancing algorithm, like the jumprope data structure used for sets of large strings — convergent in the sense that a tree containing the same set of items will always be balanced in the same way. This way, the hash can be computed on each treenode, which additionally enables hash-consing, especially useful for FP-persistent data structures.

A third approach is to compute a bloom filter of some fixed parameters (or with some fixed sets of parameters) for the items in the set, rather than XORing their raw hash values together; then you can hash the Bloom filter by, for example, adding its words together at hash time. The bloom filter may be only a few words long (say, 64 or 128 bits) and yet effective under many circumstances, but there are difficult tradeoffs around expected set size.

This approach has the additional benefit that you can directly compute bloom filters for unions and intersections, though not set subtractions, and the bloom filter may speed up membership tests. Bloom filters' Achilles heel is their lack of support for deletion.

† "Persistent" unfortunately has two conflicting meanings when it comes to data structures; functional programmers use it to mean that modifying the data structure does not make its previous states inaccessible. By "FP-persistent" I mean this meaning, not the more common meaning of "retrievable after rebooting your computer".

Topics

Trees as code

Kragen Javier Sitaker, 2016-05-10 (4 minutes)

A typical representation of a binary tree in memory might be something like this:

```
typedef struct node { char *key; struct node *left, *right; } node;
node qn = { "queer", NULL, NULL }
   , rn = { "rude", qn, NULL }
   , tn = { "the", NULL, NULL }
   , sn = { "sad", rn, tn }
;
```

This is, as it happens, a binary search tree: the key of a node is always greater than any of the nodes in its left subtree and less than any of the nodes in its right subtree. In this case we're using it to store a set of words.

You could write separate functions to do things like efficiently look up a word to see if it's in the tree, print out all the words in the tree, hash all the words in a lexical range of the tree into a hash table. Most of these functions, in C, will have this general form:

```
void process_tree(node *n, context *ctx) {
    if (!n) return;
    results r = process_node(n->key, ctx);
    if (left_wanted(r)) process_tree(n->left, ctx);
    if (right_wanted(r)) process_tree(n->right, ctx);
}
```

Different `process_tree` functions will have different `results`, `process_node` `left_wanted`, and `right_wanted`, in some cases trivial ones, but all of the functions I mentioned above can be written in this way.

An obvious thing to do then is to factor out the common `process_tree` logic into a function and pass in a `process_node` function to it with a defined interface; for example:

```
typedef struct { bool left, right; } wanted;
typedef struct { wanted (*f)(const char *, void *); void *ctx; } visitor;
void visit_tree(node *n, visitor v) {
    if (!n) return;
    wanted w = v.f(n->key, v.ctx);
    if (w.left) visit_tree(n->left, v);
    if (w.right) visit_tree(n->right, v);
}
```

This definitely works, and it's within a constant factor of optimal, but it is maybe a little bit inefficient. In particular, it spends a lot of its time passing null pointers to itself, copying the two-word visitor struct, and consulting booleans to figure out whether to recurse on a null pointer. Also, even though it will always visit the parts of the tree that it visits in the same order, it requires that order to be represented wastefully in memory with a bunch of pointers, and then it recovers that order from those pointers.

So I was thinking about immediate-mode GUIs ("IMGUI") and tree traversal, and it occurred to me that there's an interesting alternative in many cases.

Here's an alternative representation of the tree above that works with the same `visitor` type:

```
void sad_tree(visitor v) {
    wanted w = v.f("sad", v.ctx);
    if (w.left) {
        if (v.f("rude", v.ctx).left) v.f("queer", v.ctx);
    }
    if (w.right) v.f("the", v.ctx);
}
```

In a sense, this is just a partially-evaluated version of `visit_tree` for the above tree; it will perform the same sequence of calls to `v.f` with the same arguments, given the same return values. Since it can do all of the preorder visitor traversals described above, it is in some sense a fairly universal representation of the tree.

Probably you wouldn't really want to write `sad_tree` by hand, at least in this case. But it might make sense to compile it from some other representation, either with run-time code generation or from some kind of source code.

It might actually be worthwhile to write it by hand in some cases, because the visitor interface allows you to traverse kinds of trees other than trees entirely materialized in memory ahead of time.

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- Immediate-mode GUIs (p. 3515) (8 notes)
- Synthesis (p. 3739) (2 notes)
- Partial evaluation

Very composite numbers

Kragen Javier Sitaker, 2014-04-24 (4 minutes)

It's often more convenient in practice to use feet and yards than meters because you can divide them evenly into three, four, six, or nine equal units, with a convenient whole number of inches. For the same reason, the Babylonian sexagesimal system remains in use today for angles (360 degrees, each subdivided into 60 minutes of 60 seconds). In some sense, both 10 and 12 are composite numbers, but 12 seems much "more composite", and 60 is "more composite" still, much more so than 100.

What are the "most composite" numbers, the ones that can be evenly divided in the most different ways? Of course you can always get more factors if you go to a larger number, but some numbers have more factors than any number smaller than themselves; there's a definite sequence of these "most composite" numbers, and there aren't really very many of them. You might think that you'd just go with the products of the primes: 2, 2·3 = 6, 2·3·5 = 30, 2·3·5·7 = 210, and so on; but it turns out that 30 only has six proper divisors greater than 1 (2, 3, 5, 6, 10, and 15), but 24 beats it by having the same number of factors and being smaller. It substitutes a versatile, lightweight extra couple of 2s for 30's 5, thus getting the same number of factors sooner.

It turns out our old friends 12, 36, 60, and 360 are in this sequence, along with 4 and 6. However, 360·60 = 21600 is not, because once you get past 720, it behooves you to start throwing a 7 in there. The champion that beats poor old Babylonian 21600, with its mere 70 divisors, is 10080 ($2^5 \cdot 3^2 \cdot 5 \cdot 7$) with 70 divisors, or alternatively 20160 ($2^6 \cdot 3^2 \cdot 5 \cdot 7$) with a whopping 82 divisors! (I'm still leaving out 1 and the number itself, here.)

Here's what I've computed so far:

```
>>> nfactors = lambda n: len([x for x in range(2, n) if n % x == 0])
>>> i = 1
>>> best_nfactors = 0
>>> while True:
...  i += 1
...  n = nfactors(i)
...  if n > best_nfactors:
...   print i, n
...   best_nfactors = n
...
4 1
6 2
12 4
24 6
36 7
48 8
60 10
120 14
180 16
240 18
360 22
```

720 28
840 30
1260 34
1680 38
2520 46
5040 58
7560 62
10080 70
15120 78
20160 82
25200 88
27720 94
45360 98
50400 106
55440 118
83160 126
110880 142

Unsurprisingly, I'm not the first person to discover this; the standard term is "highly composite number", and it's sequence A002182 in OEIS.

If, on the other hand, you're looking for a base for a positional numbering system that will give you maximum divisibility, 30 beats 24, and 210 is better still. True enough, in base 24, an eighth is simply 0.3, while in base 30, it's 0.3,22,15, which is more awkward; but in base 30, a fifth is 0.6, while in base 24, it's an infinite sequence: 0.4,0,19,4,0,19,...! Infinite sequences are far more awkward to calculate with. Base 210 gives you perfect divisibility --- in a finite number of fractional positions, but maybe more than one --- for any number of portions up to 10, and any composite number up to 21, plus of course many other numbers.

Topics

• Math (p. 3564) (78 notes)

A reactive crawler using Amygdala

Kragen Javier Sitaker, 2014-09-02 (updated 2014-09-19) (4 minutes)

This is a design for a reactive programming system optimized for web crawling, to be implemented in Java. We want it to be as simple as possible to write a web-crawling application that transmits updates when it has them — say, to an IRC channel, or a webhook.

Suppose we want to send the Bitstamp Bitcoin price, translated into Euros, to an IRC channel, but only when it changes by more than 0.5%, and without hitting any of the web servers more often than once every 20 minutes. We'd like to be able to write code like this:

```
import org.canonical.amygdala.UserAgent;
import org.canonical.amygdala.Page;
import org.canonical.amygdala.DBVar;
import org.canonical.amygdala.DB;

// ...

final DBVar dbVar = DB.var("Ⓑ in €");
start("compute Ⓑ in €", new Runnable() {
    public void run() {
        String ratesUrl =
            "http://www.x-rates.com/calculator/?from=EUR&to=USD&amount=1",
            bitstamp = "https://www.bitstamp.net/";
        // Incapsula lets Googlebot access pages without having to
        // execute JS.  Bitstamp uses Incapsula.
        UserAgent ua = UserAgent.create("Mozilla/5.0 (compatible; "
            + "Googlebot/2.1; +http://www.google.com/bot.html)");
        Page exchangeRates = ua.get(ratesUrl).maxAge(minutes(20)),
             bitstamp = ua.get(bitstamp).maxAge(minutes(20));
        // Extract floating-point numbers with regular expressions.
        Float bitcoinPrice =
                bitstamp.extFloat("class=\"last\">$([\\d.]+)<"),
            euroPrice =
                exchangeRates.extFloat("class=\"ccOutputRslt\">([\\d.]+)<"),
            bitcoinInEuros = bitcoinPrice / euroPrice;
        dbVar.setFloat(bitcoinInEuros);
    }
});

start("irc announcement", new Runnable() {
    public void run() {
        DBVar last = DB.var("last Ⓑ in €");
        float newVar = dbVar.get().float(),
        Float oldValue = last.get().floatOption();

        if (oldValue == null
          || Math.abs(oldValue - newValue) / newValue > 0.005) {
            ircChan.say("Ⓑ = €" + newValue);
            last.set(newValue);
        }
    }
```

```
});
```

and have it "just work", which in this case involves fetching the requested pages, reinvoking the runnable once it has them, and reinvoking it again every time it has new versions of those pages.

Amygdala needs to run the first runnable once simply in order to find out what URLs it will try to fetch — but since it hasn't fetched any pages yet, the first attempt to use regexps to extract floats from the not-yet-fetched pages will raise an exception. But the page fetches will have been initiated, and the runnable will be reinvoked once one or the other is available.

Invoking get() on a DBVar notes that DBVar as a read-dependency of the current task; invoking float() on the result will return the float stored there. There are three possibilities:

• Nothing is stored there yet. In this case, float() will silently abort the transaction, and it will be rerun when something is stored there. To avoid aborting the transaction in this case, you can use floatOption() instead.
• Something is stored there that isn't a float. In this case, float() will abort the transaction with an error (and it will be rerun when the value stored there changes).
• A float is stored there. In this case it will be returned.

So the first time the "irc announcement" runnable is invoked, it will fail the transaction because "Ⓑ in €" has no value yet; if and when a value is stored there, it will be rerun, and then it will also read, and write, "last Ⓑ in €".

I don't quite know what the right handling of the ircChan.say call is — it could either check to verify that speaking on the channel is possible (and abort the transaction if not), waiting to actually transmit until the transaction as a whole commits; or it could simply transmit from the middle of the transaction, with the unfortunate result that retrying the transaction could result in duplicate messages on the channel.

The maxAge ensures that we never fetch the pages more often than once every 20 minutes. But as long as the task is running, the fetcher will continue fetching those pages whenever they become stale.

This means Amygdala supports both "push" tasks, like those above, and "pull" tasks, which only run as long as something is listening to them. (Actually, I think that's not quite true; perhaps everything here can be "pull" from the IRC channel?)

Topics

• Programming (p. 3658) (286 notes)
• Incremental computation (p. 3517) (24 notes)
• Java (p. 3531) (5 notes)
• Amygdala

mechanical computation: with Merkle gates, height fields, and thread

Kragen Javier Sitaker, 2010-06-28 (36 minutes)
(Originally published 2010-06-28.)

So one of the reasons I'm excited about automated fabrication (e.g. Fab@Home, tabletop CNC mills, RepRap, inkjet-printing of circuits, fused deposition modeling) is that I expect it will make it possible to build computers from raw materials with minimal capital. It will be some time before those computers are as cheap or as fast as mass-produced microcontrollers, so they will start out as curiosities. But how far away is the prospect of automatically building a working computer from raw materials with minimal capital? I think it's already possible.

This post considers three kinds of mechanical computers: one using Merkle's buckling logic, one using mechanical height fields, and one using pulling of threads.

Organic semiconductors, nonlinear acoustic devices, fluidic logic, or something more exotic, might turn out to be the form that computers made by 3-D printers eventually take; but I'm pretty sure Merkle's approach is workable. So I'm going to consider that first. And I want to describe some ideas of my own.

Unfortunately, I don't know shit about mechanical engineering. If any actual mechanical engineer is annoyed at my ignorant speculations when they read this, I'd be delighted to hear about all the things I'm getting wrong.

How many logic gates? Around 64000.

There have been a number of capable small CPUs over the years that contain on the order of 4000–8000 transistors, including the 6502 used in the Apple][, the NES, and the Atari 2600 (4000 transistors, 8-bit ALU), Chuck Moore's MuP21 (6000 transistors, 21-bit ALU, including video coprocessor), Voyager 2's RCA 1802 (5000 transistors), CP/M machines' 8085 (6500 transistors) (the 8080 was smaller but needed more support circuitry), the Apollo Guidance Computer (4100 3-input RTL NOR gates, which I think is about 8200 transistors) and so on. The IBM 1401 was supposedly more complex: http://ed-thelen.org/comp-hist/BRL64-i.html says a minimal 1401 system had 6213 diodes and 4315 transistors. It ran at 86 957 Hz. (How big was the PDP-7 Unics was written on?)

To run an interpreted programming language, you probably need at least 32000 bits of memory, and twice that is better.

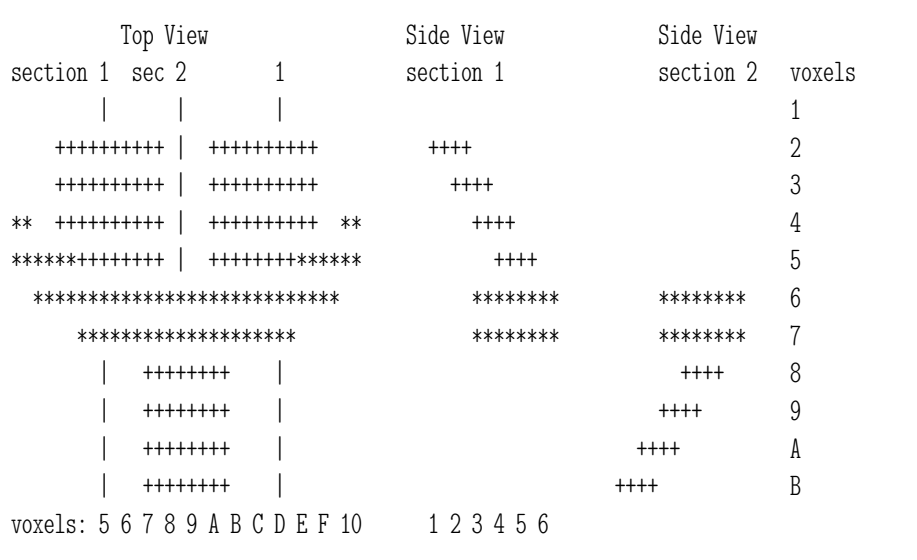
So 64000 "logic elements", each of which can be either a bit of memory or a gate, should more than suffice. 64000 is the cube of 40, so a 40x40x40 cube of logic elements would be sufficient if you didn't need any space for signal routing. In two-dimensional chips, I've heard it's typical to spend 90% of the area on routing; things are much closer together in three dimensions (64000 elements is about 250×250 in 2D, so that far-apart things in a device of that size need

more than six times as much wire to connect them), so 10x routing overhead is quite a pessimistic assumption for 3-D; but if we accept it, then we need an 86×86×86 cube.

It might be possible to reduce this substantially if the memory is some kind of bulk medium instead of one or more parts per bit. Historically-used bulk media have included mercury acoustic delay lines; magnetostrictive torsion acoustic delay lines in wire; magnetic films on tapes, drums, and discs; impressions in the surface of thixotropic substances such as wax or wet clay; ink or pencil lead on papyrus or paper; knots tied in khipu cords; and electric charges on the surface of cathode-ray tubes.

How many voxels per logic gate? Around 1200 for Merkle gates.

How big does each element need to be? Presumably if we're fabricating it with the CandyFab 2000, it needs to be pretty gigantic. (Sugar wouldn't bend enough, but maybe you could make it out of polyethylene pellets with the CandyFab.) But if you are depositing tiny beads of molten ABS plastic with 2-mil precision, you could make it a lot smaller. You could probably get a working Merkle buckling-spring cell in something like 16x6x12 voxels:



If that were really the size you needed, at 2 mils per voxel, it would be 32 × 12 × 24 mils, or about 0.8 mm × 0.3 mm × 0.6 mm, which altogether adds up to just over an eighth of a cubic millimeter --- 0.52 mm cubed.

You could probably do something a lot more ingenious and get an order of magnitude or two improvement.

Which means that the total 86×86×86 cube, 6 billion voxels, would be 45 mm on a side, if you have a 3-D printer that can construct a complex object with arbitrary 2-mil voxels.

One big advantage of the Merkle-gate design is that it doesn't require any contact between separate elements, sliding or otherwise, so surface finish may be less crucial than for some other kinds of machines.

Height-field computing: mechanical LUTs to reduce the number of elements

The elements in modern FPGAs ("field-programmable *gate* arrays") mostly consist of lookup tables ("LUTs") rather than actual gates in the array. The idea is that basically you use your N bits of input to index into a little EEPROM and get a bit of output, which allows you to emulate any possible combinational circuit, and then you have "routing resources" — basically crossbar multiplexers — to connect those outputs to the inputs of other cells.

For automated fabrication to help much with building mechanical computers, compared to just carving them out of wood or steel or whatever with non-automated machine tools, it's going to have to reduce the number of separate pieces that you have to assemble manually when you're done. Some kinds of 3-D printing can print already-assembled parts that mesh together nicely and aren't stuck together; others can't. Those that can't will probably require manual assembly for a while yet. (Although, hey, pick-and-place machines could be used for that, right?)

Height fields as a mechanical realization of LUTs

LUTs have a fairly straightforward low-parts-count mechanical realization. If you have a needle-like probe positioned over a solid height field, then when you lower the probe onto the height field, the height at which it stops will be an arbitrary function of its X and Y coordinates: the height of the surface at that point. This "lowering" step is similar to the step of squeezing the gate of a Merkle buckling-spring gate, or pushing a Drexlerian rod-logic rod to see where it stops.

If you encoded one bit in X and one bit in Y, you can get an arbitrary two-input boolean function in Z; but Z's range isn't limited to booleans. Likewise, you could encode multiple bits in each of X and Y; you're limited largely by the aspect ratio of holes you can carve into your height field. In fact, to compensate for minor errors in X and Y, there should be at least a dimple in the surface at the desired position.

But if you carry this out just like that, you may end up with tall, thin towers on your height field, which will be prone to bending or breaking. Instead, you could just drill a matrix of holes of different depths. How small could you make these holes?

Jewelers' twist drill bits, which cut cylindrical holes, normally come in sizes 1 through 80 in the US. According to Wikipedia, size 80 is 0.343 mm in diameter, or 0.0135", about a 74th of an inch. So you could probably drill a 32x32 matrix of these holes into a one-inch block of, say, aluminum, brass, or soapstone. Then you'd only need the positioning of the probe to be accurate to within about 0.007" to make sure that it went into a hole; if your machine drilling the depths, like certain inexpensive home knockoff CNC drill presses, were only accurate to 0.002", you could use a 3× mechanical advantage between the probe and whatever input it was driving so that you would need a 1" x 1" x 0.5" cube of metal for this 32x32 array of holes.

(It's probably best to translate the block itself in at least one of the three dimensions, rather than translating the probe in all three, in order to diminish the accumulated positional error.)

I'm not sure how many drill bits this procedure would use up, and how much it would cost. "The Real Cost of Runout" talks about

reducing the cost of drilling 3mm-diameter holes from 80 cents to 27 cents per hole by reducing runout on the drill press, or from 23 cents to 10 cents per hole when using high-speed steel instead of tungsten carbide — and these numbers are just for the cost of the drill bits! But 1024 holes at 10 cents per hole is still US$102.40. I hope smaller holes cost less?

Smaller drills exist; http://www.ukam.com/diamond_core_drills.html says they have diamond drills "from .001" to 48" (.0254mm to 1219mm) diameter." Being able to drill holes of .001" diameter would mean being able to drill a 32×32 array of holes in 0.064" × 0.064". On http://www.ukam.com/micro_core_drills.htm they actually only list drills down to 0.006", which they recommend using at 150 000 RPM, feed rate 0.010" per minute.

In some ways, this kind of drilling operation is less demanding than ordinary drilling operations: surface finish, hole straightness, and even hole positioning can tolerate quite a lot of slop. (If you countersink the holes, they can be off by quite a bit as long as it doesn't break the probe.) But microdrilling --- drilling holes of under 0.5mm --- apparently poses special problems in chip removal and cooling.

Other manufacturing processes might make more sense than drilling. For example, you could conceptually cut the chunk into 33 pieces along 32 lines of holes, cast the 33 pieces using lost-wax casting, and then clamp them together. Or you could make a mold with 1024 adjustable-height rods stuck in through holes in the cope, and cast in that. (Maybe in plastic.) Etc.

What a LUT is good for

So that's a LUT with 10 bits of input — one of 32 possible positions in two independent axes, positioning the needle probe over the array of holes — and 5 bits of output — one of 32 possible depths for the needle — realized in about a cubic inch. That's also about 5120 bits, or 640 8-bit bytes. That's enough to realize an arbitrary 32-state state machine with 5 bits of input at each step, or to perform 4-bit binary addition or subtraction with carry-in and carry-out and an extra input bit left over (say, to select between addition and subtraction), or to perform a selectable one of four arbitrary 5-bit combinational functions on two 4-bit inputs --- say, addition with carry out, AND, XOR, and something else.

If you split the same 1024 holes into two LUTs with ganged input — that is, two probes — then you get 9 bits of input and 10 bits of output. A 4x5-bit multiply needs only 9 bits of output. You could do a 4x4-bit multiply in half the area and half the depth.

All of these applications still work just as well if any or all of the quantities are encoded in some weird way such as a Gray code or an excess-N code.

At some point you have to encode the 32 slightly different linear displacements into five distinct bits. You can do this with a 32x5-hole LUT, with five probes sticking into it, and only two distinct hole depths.

Decoding five bits encoded in five separate displacements into 32 levels of displacement in a small number of moving parts is maybe more difficult.

The ZPU project on OpenCores is a 32-bit CPU; realized in a

Xilinx FPGA of LUTs, it uses “442 LUT @ 95 MHz after P&R w/32 bit datapath Xilinx XC3S400”. I don’t remember offhand how big the XC3S400 LUTs are, but they have only single-bit outputs.

Memory

A shaft that can slide freely is merely transmitting positional information from one place to another. A shaft that has a clamp closed on it can remember its position from before the clamp was closed until the clamp reopens. If the clamp and shaft surfaces are not flat, the shaft can be retained in any of its valid positions (reshaping the signal) with very little force and comparatively imprecise surfaces.

This allows you to store, say, 40 bits with nine moving parts: eight shafts (each encoding 5 bits) and one clamp for all of them.

Shift registers in LUTs

You can make an 8-bit shift register out of two 4×4 → 4 bit LUTs and two 4-bit memory units (known as registers) if you are willing for it to always shift; each LUT given (a, b) computes (a << 1 & 15 | (b & 8)

3), which is written to its memory unit for the next cycle.

The communication from the low nibble LUT to the high nibble must be intermediated through the memory; this allows both LUTs to transition at the same time and means that the bit being shifted into the high nibble is the old MSB of the low nibble, not the new one. Using (b & 8) >> 3 means you don’t need to decode the LUT output.

The contents of the LUT (two four-bit inputs producing one four-bit output) looks like this:

```
array([[ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15]])
```

Clearly you can reduce this to a 4×1 → 4 bit LUT if you have a way to extract just one bit from the b input. For example, you could use a 4 → 1 bit LUT: 0000000011111111. You could perhaps integrate this into the memory from which it’s being read, as an additional row of holes with a probe over it.

If you have a way to do that, you have space to include an opcode that tells the register what to do. For example, shift left, remain steady, shift right, or reset to 0. If you have a way to combine this opcode, the MSB of the less-significant nibble, and the LSB of the

more-significant nibble into a single 4-bit input `b = LSB | MSB << 1 | opcode << 2`, your table can look like this instead.

```
>>> def shift(a, b):
...     lsb, msb, opcode = b & 1, (b & 2) >> 1, b >> 2
...     shift_left, nop, shift_right, reset = range(4)
...     if opcode == shift_left:    return a << 1 & 15 | msb
...     elif opcode == nop:         return a
...     elif opcode == shift_right: return lsb << 3 | a >> 1
...     elif opcode == reset:       return 0
...
>>> Numeric.array([[shift(a, b) for a in range(16)] for b in range(16)])
array([[ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 0,  2,  4,  6,  8, 10, 12, 14,  0,  2,  4,  6,  8, 10, 12, 14],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 1,  3,  5,  7,  9, 11, 13, 15,  1,  3,  5,  7,  9, 11, 13, 15],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15],
       [ 0,  0,  1,  1,  2,  2,  3,  3,  4,  4,  5,  5,  6,  6,  7,  7],
       [ 8,  8,  9,  9, 10, 10, 11, 11, 12, 12, 13, 13, 14, 14, 15, 15],
       [ 0,  0,  1,  1,  2,  2,  3,  3,  4,  4,  5,  5,  6,  6,  7,  7],
       [ 8,  8,  9,  9, 10, 10, 11, 11, 12, 12, 13, 13, 14, 14, 15, 15],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]])
```

All of the above generalizes to N-bit shift registers; you just need a LUT and a register for every four bits.

Addition

Parallel addition (the usual digital kind) with inputs and outputs as the usual kind of binary numbers (not carry-save addition) can't be done totally in parallel; the carry from the least-significant bits must be ready before the most-significant bits can produce their final result.

But still, with $4\times3 \rightarrow 4$ LUTs we can do three bits at a time. We bring in the carry along with one of the inputs; the LUT looks like this:

```
>>> def add(a, b):
...     carry_in, inb = (b & 8) >> 3, b & 7
...     return a + inb + carry_in
...
>>> Numeric.array([[add(a, b) for a in range(8)] for b in range(16)])
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 1,  2,  3,  4,  5,  6,  7,  8],
       [ 2,  3,  4,  5,  6,  7,  8,  9],
       [ 3,  4,  5,  6,  7,  8,  9, 10],
       [ 4,  5,  6,  7,  8,  9, 10, 11],
       [ 5,  6,  7,  8,  9, 10, 11, 12],
       [ 6,  7,  8,  9, 10, 11, 12, 13],
       [ 7,  8,  9, 10, 11, 12, 13, 14],
```

```
       [ 1,  2,  3,  4,  5,  6,  7,  8],
       [ 2,  3,  4,  5,  6,  7,  8,  9],
       [ 3,  4,  5,  6,  7,  8,  9, 10],
       [ 4,  5,  6,  7,  8,  9, 10, 11],
       [ 5,  6,  7,  8,  9, 10, 11, 12],
       [ 6,  7,  8,  9, 10, 11, 12, 13],
       [ 7,  8,  9, 10, 11, 12, 13, 14],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
```

This gives you, for example, 9-bit addition in three levels of LUT, plus whatever it takes to shuffle the bits around appropriately.

Bit shuffling

In several of the previous items I have assumed a way to combine bits from disparate sources into a single positional input, or to drop bits. This problem also occurs on input. I hope there's a better way to do this, but one workable way is to use a LUT. I've pointed out earlier that a 4×0 → 1 bit LUT can select a single bit, but you can do things more generally. For example, in the shift-register case, where we want to combine a neighbor MSB possibly being shifted left, an LSB possibly being shifted right, and a two-bit opcode, we can use two small LUTs:

```
>>> Numeric.array([[(opcode << 1 | msb) for opcode in range(8)] for msb in range(○
○2)])
array([[ 0,  2,  4,  6,  8, 10, 12, 14],
       [ 1,  3,  5,  7,  9, 11, 13, 15]])

>>> Numeric.array([[(opcodemsb << 1 | lsb) for opcodemsb in range(16)] for lsb in○
○ range(2)])
array([[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30],
       [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]])
```

If we have to simultaneously extract the msb and lsb from the nibbles they're embedded in, it is best to do this differently:

```
>>> Numeric.array([[(leftneighbor & 1 | (rightneighbor & 8) >> 2)
...  for leftneighbor in range(16)] for rightneighbor in range(16)])
array([[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3],
       [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3]])
```

```
>>> Numeric.array([[(opcode << 2 | msblsb) for opcode in range(4)] for msblsb in o
orange(4)])
array([[ 0,  4,  8, 12],
       [ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15]])
```

It would require many fewer LUT entries to extract those bits in a separate step.

Circles instead of lines: rotational motion is simpler

Mechanically, moving things linearly is a little trickier than moving them in circles — there tends to be more slop. If you transpose this "height field" LUT into cylindrical coordinates, you get a camshaft. The normal sliding cam follower design imposes limits on the "slew rate" of the output function, but if you lift the "cam follower" while rotating the shaft and then use the same forest of holes and "lowering step" as with the flat X-Y approach.

Cylindrical coordinates still leave two coordinates translational, though. You can cheat a little bit by doing the axial positioning along a large-radius arc that almost parallels the axis of the camshaft, to within the diameter of the camshaft, rather than in a strictly translational fashion.

A third coordinate system that might be useful approximates two translational dimensions with angles around two axes that are some distance apart; for example, any two discs that overlap, while rotating around their own centers. This allows one dimension to "wrap around", as with the camshaft approach.

Digital logic with thread

Thread is very nonlinear in some ways, and it's easy to build thread systems with very nonlinear force-displacement curves. Thread also has the advantage that, because the material is not stressed in compression, its effective stiffness-to-weight ratio is enormous, so it can transmit signals quickly, and the energy needed to move it is slight.

As far as I can tell, the force-displacement curve of a system made of elastic thread tied between a bunch of fixed points is monotonic; that is, if you pull on a loose bit of the thread, the force with which the thread pulls back on you always goes up as you pull it further from its natural position. (There may be some trick with knots that violates this.) This imposes limits on pure thread systems, although you can still get amplification through braking.

Thread springs

A steel guitar string can easily be pulled a substantial fraction of an inch out of place, because although steel is very stiff, the d[cos x] / dx = 0 when x=0, so even very small elongations of the string allow substantial movement. (And the force-displacement curve is very nonlinear.)

This effect can be chained: if you have a thread configuration shaped like a capital H, you can pull up or down on the middle of the H's crossbar fairly easily, because it only has to pull slightly on the

sides of the H in order to move. So you get a lot of mechanical advantage, we could say.

So this kind of spring force can allow for reciprocating movement in response to a reciprocating input force pulling on some "power supply" thread.

Thread power amplification through braking

For fanout, we need the ability for one logical gate output to, eventually, control an arbitrarily large number of logical outputs. This can't be achieved just by having strings pull on each other in configurations like that described above; the amount of force needed would go up with the number of output stages, which prevents useful computation.

But braking can provide amplification. Imagine that you have a thread running along the surface of a cylinder; it can slide freely, as far as it as long as no force presses it against the cylinder. If another thread is wrapped loosely several times around the cylinder and the sliding thread, the sliding thread can still slide; but if the wrapped thread is then pulled taut, it presses the sliding thread against the cylinder, preventing it from sliding.

The crucial aspect here is that, although there is a limit to how much force the thread brake can resist, it can resist that force regardless of how far the sliding thread would slide without that resistance; and it can do it with an almost arbitrarily small displacement of its onw.

There are other approaches to braking that might turn out to be worthwhile; for example, if the wrapped thread clamps the sliding thread between two cylinders, then they can be in contact with different kinds of surface with different coefficients of friction, and the sliding thread won't catch and pull on the wrapped thread. But those approaches involve moving more mass.

Braking also provides memory, since the amplified signal happens later than the amplifier input.

Thresholding with thread

There are a couple of approaches to getting the kind of nonlinear behavior you need for AND or OR gates.

If you have a couple of threads pulling against a spring, the spring's displacement will be a function of the sum of the force from those threads. If the spring's force-displacement curve is monotonic and nonlinear enough, you can use it to approximate the AND or OR functions.

Alternatively, the force-displacement function of a thread tied to a fixed point has a huge nonlinearity as it runs out of slack: while it's slack, the force is basically zero.

Limiting displacement with thread

If you couple two threads through a spring, you can use either of the thresholding techniques in the previous section to limit the displacement of the driven thread without limiting the displacement of the driving thread. This costs energy (the spring continues to store energy as the driving thread pulls further) and also loses some displacement, as the displacement transmitted to the driven thread will always be less than the displacement of the driving thread.

Changing direction with thread, and increasing displacement

If you have a thread tied to a fixed point, its end describes a circle around that fixed point when it's under a small amount of tension. If the fixed point is far away, the circle approximates a straight line.

If you have another thread tied to that end at an angle not parallel or perpendicular to (that part of) the circle, that other thread will be able to pull it along the circle; so you can change the direction of motion that way by anything less than 90°. If you have a second thread pulling the other way, also at an an angle neither parallel nor perpendicular, then displacement is transmitted between the two threads, but can be changed by any angle at all.

If the angles made to the tangent line are equal, then the same displacement and force is transmitted, with no mechanical advantage, or rather a mechanical advantage of unity. But you can achieve MAs of either above or below 1 by having one angle be greater than the other.

This allows you to convert a small displacement with great force into a large displacement with small force, and vice versa. (There will be vibrational losses, but they can probably be made small.) It is that ability to reduce forces to the point where they can easily be controlled by braking, then step them back up, that makes me confident in the braking mechanism as a means of amplification.

Negation with thread

If ones and zeroes are represented by different displacements when the countervailing force is within a certain range, then negation can be achieved by pulling in the opposite direction against a spring force.

Sequencing with thread

If you have a cord with a number of threads tied to it with different amounts of slack in them, then when you pull on the cord, the various threads will go taut and begin to transmit force one after the other. This, plus limiting displacement as discussed previously, allows you to drive different parts of a thread logic system in a predetermined sequence.

A generic thread state machine

Initially, you have a number of "register" threads R_0 braked into some position or other by threads wrapped around them, held taut by springs. A few stages of thread combinational logic are driven from those positions, producing a thread combinational output C_0.

You begin to pull on the clock thread. This drives a final stage of combinational logic connecting the thread combinational output C_0 to another set of registers R_1, whose braking threads are currently lax, so their threads are free to slide back and forth — which they do, under the influence of your clock thread, pulling against the inputs of another set of combinational logic producing a second combinational output C_1.

Once the R_1 threads have found their position, your further pulling on the clock thread is transferred to their brake threads, which hold them in place, preventing the combinational output C_1 from changing further.

Further pulling on the clock thread loosens the tension on the brake

threads for R0, allowing those registers to assume their new state — which is coupled in from C1! There is vibration as previously slack threads snap taut, and the outputs of C0 (driven from R0) change. But R1's threads are still held in place.

Now you begin to release the tension on the clock thread. First the brake threads on R0 become taut again, preventing R0's value from changing. Then the brake threads for R1 loosen, and R1's threads are free to assume the new values of C1's output, so there is more vibration as previously slack threads snap taut.

Further loosening of the clock thread tension eases R1's state threads back into a neutral position.

Now R0 has its new state, and is ready to begin a new clock cycle.

This is similar to the functioning of a master-slave flip-flop, with its input driven from its output, but of course with arbitrary combinational logic.

I haven't tried to build this yet, but from the above, I think it's plausible.

To me, there's a strong appeal in the idea that universal computation has been within the grasp of human materials and manufacturing technology since the invention of sewing in the Paleolithic; it is only the mathematical sophistication that was absent. I don't yet know if it's true.

References

Merkle 1990 was published in Nanotechnology, Volume 4, 1993, pp. 114–131.

MMS 2006 was an article about the effects of runout on tool life published in Modern Machine Shop, 2006-06-14, by Peter Zelinski.

ZPU is intended to be "the world's smallest 32-bit CPU". Øyvind Harboe at Zylin seems to be the guy who licensed it under a free license, maybe built it too, in 2008.

Topics

Earring computer

Kragen Javier Sitaker, 2018-04-27 (1 minute)

The Lattice ICE40LP1K and ICE40HX1K FPGAs are big enough to support a J1a CPU (with 8K of RAM?) and are 1.4 × 1.5 × 0.45 mm and cost US$3.43.

The TI ADS7040 8-bit 1Msps ADC has an 8-X2QFN version that is 1.4 × 1.4 × 0.4 mm and costs US$1.14. It claims "nanowatt power consumption" by which they mean that it uses less than a microwatt at 1ksps. At 100ksps it uses 56µW, and the datasheet claims a physically impossible 49dB SNR (48.16 being the theoretical limit of what 8 bits can deliver). It uses 12MHz SPI and uses differential input. If you were to run it at 1Msps and downsample to audio 48ksps, you would gain about 7dB more.

Topics

Efficiently querying a log of everything that ever happened

Kragen Javier Sitaker, 2015-09-03 (7 minutes)

I've been designing a new chat/email UI called Desbarrerarme, and one of the design approaches I want to try out is computing the UI state as mostly a pure function of the input history of the system.

One of the things I've been thinking about for Desbarrerarme is querying. By far the nicest existing way I've found to do normal queries is Prolog/Datalog; it's dramatically better than SQL or Tutorial D. (Binate may be better, but it's unfinished.). However, it's not totally clear to me how to do aggregate queries, in particular "top N" queries.

I've been struggling with this for a long time; see e.g. http://stackoverflow.com/questions/1467898/what-language-could-i-use-for-fast-execution-of-this-database-summarization-task. ("Top 5 scores and their dates for each player"; one response: "Top-N is a well-known database killer. As shown by the post above, there is no way to efficiently express it in common SQL.")

Here are some example queries I've been thinking about for Desbarrerarme:

- What are the N most recently updated threads (i.e. threads with the most recent messages)? Because those are the ones we want to show on the screen.
- What are the participants and the most recent message in each of the N most recently updated threads? Because that's what we want to display on the left side.
- What are the N most recent messages in thread X? Because those are the ones we want on the screen.
- What are the N most recently updated threads *I haven't tagged as to-archive*?
- What are the N next scheduled reminders after the beginning of today that I haven't marked as taken-care-of?
- What are the N best threads that contain the words "apache" and "spark", case-insensitively, where "best" is some kind of combination of recency and TF/IDF?

If we have a single relation `messages` with the columns `when, thread, from, body`, and some similar thing for reminders, it's apparent how to compute each of these inefficiently. We can calculate #2, for example, like this:

```
participants = {}
last_message = {}
top_n = []
for when, thread, from_, body in sorted(messages):
    if thread in top_n:
        top_n.pop(top_n.index(thread))
    else:
        if len(top_n) >= n:
            top_n.pop(0)
```

```
    top_n.append(thread)

    if thread not in participants:
        participants[thread] = set([from_])
    else:
        participants[thread].add(from_)

    last_message[thread] = (when, body)

for thread in top_n:
    yield participants[thread], last_message[thread]
```

This has three big disadvantages:

• It's a lot of code. The English version of the question was 18 words; the Python version above is 18 lines.
• That code is poorly abstracted, so it isn't reusable. For example, it's very likely that this isn't the only place we'd like to know the participants of a chat.
• It's inefficient. It inherently needs to traverse the entire `messages` relation, which might be many gigabytes, and it accumulates a potentially large `participants` relation in memory because it doesn't know at the outset which threads are going to be included in the final result.

Problems #2 and #3 are interrelated. For example, if we could freeze the whole execution state of the loop partway through, then we could incrementally recompute its final state if new messages were added with a later date.

Better still, in this case, you could break the computations down into a monoid tree — `participants` can be computed from results over subsets of the data set simply using the union operation, while `top_n` could be computed from results over subsets of the data set by concatenating those results (ordered by timestamps) and truncating to the last N. Abstracting the operations to this level would allow them to be not only recomputed incrementally after updates, but also parallelized and made fault-tolerant. Also, the `top_n` result is probably somewhat lazy — it's probably possible to compute the most recently updated 10 or 20 chats without looking further back than 1–30 days.

Ideally, we could find a set of language primitives to express queries like these that would make the query simple to express. Something like maybe

```
top = max(when) by thread | sortby when desc | limit n
participants = distinct(project(thread, from))
latest = max(when, body) by thread
(participants naturaljoin latest) if thread ∈ top.thread
```

Or, I don't know, something still more concise. And ideally that set of language primitives would expose enough structure to a runtime to allow the orthogonal specification or even inference of optimizations like the ones suggested above; and ideally it would also, like the above, make it straightforward to reuse the definitions for other things.

Also, if it's not too much to ask, generalize the materialized views

that are used to speed up incremental recomputation so that they can be used for a larger range of queries than just the current ones. That is, maybe I only asked for the latest 12 chats, but maybe it would be prudent to calculate the latest 24 or 64 if that isn't harder; and maybe `latest` only contains `when` and `body` here, but maybe we should also keep `from` just in case.

Additionally, I'd like to get the results from these queries incrementally, so that the UI isn't frozen while the query is being evaluated.

This seems like it might be related to Spark's concept of "resilient distributed datasets". I need to read more.

Faster brute force

An alternative approach to caching and materializing a bunch of shit is trying to make brute-force query evaluation sufficiently fast. Considering this `messages` thing again, suppose I had 10 messages per second for the last 20 years. (Busy IRC channels.) That's 6.3 billion messages. But maybe there are less than 65536 participants, so participants can be identified by 16-bit numbers; so the participants column there is only 12 gigabytes, and it's probably highly enough compressible with LZ4 to fit into RAM easily. Similarly, suppose we're using 64-bit microseconds for `when`; now we have 50 gigabytes of timestamps, but again, they probably fit into RAM easily with LZ4. The message bodies won't, but they'll fit on a small SSD, and will probably compress enough to hit RAM-like speeds for scanning.

And the above query, if the query optimizer is lucky enough to pick a reasonable evaluation order, should only need to traverse the tail end of the `when` and `thread` columns before hitting `n`, and then the entire `from` column. Still, this is probably not going to get us to subsecond response times.

Topics

Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting?

Kragen Javier Sitaker, 2019-08-27 (updated 2019-08-28) (37 minutes)

Today, someone on Freenode ##electronics was asking about wireless communications between AVR microcontrollers with only passive components, so I was interested in the available energy for harvesting and possibilities for low-power RF communication. I started reading about ferrite loopstick antennas and energy harvesting from broadcast AM radio (as suggested by user Bga3 on Freenode ##electronics). These antennas are normally useful up to about 5 or 10 MHz and can be salvaged from old AM radios; more exotic ferrite mixes are viable up to 50 MHz.

As mentioned in Arduino radio (p. 169) and Could you do DDS of comprehensible radio signals with an Arduino? (p. 1829), the usual AM radio band is around 0.5 MHz to around 1.7 MHz. I don't think the AVR is the best choice for this kind of thing, since it's fairly power-hungry and slow, but even the AVR should work. The AVR has PWM generation circuits that can generate frequencies up to 8 MHz, so some degree of amplitude modulation at a few hundred kHz should be doable.

Available AM radio energy

The highest power available for broadcast radio station licensing in the US is 50 kW, while there are many 10-kW radio stations around. So let's suppose you're located 10 km from ten isotropic 10-kW AM transmitters, which would seem to be a common situation in urban areas. Each contributes about 8 μW/m² of radiation (10 kW spread over a sphere of area 1.26×10^9 m²), for a total of 80 μW/m². But you can't collect all of that energy because loopstick antennas are inefficient. Still, μW-scale energy harvesting seems like it should be feasible. And of course crystal radios were powered entirely from AM radio waves, and had enough power to be audible, if barely.

Related work

Trask on loopstick antennas

Research has been done on designing loopstick antennas to be tunable (Chris Trask, "An Active Ferrite Rod Antenna with Remote Tuning" with a high Q factor by setting up a resonance between the antenna inductance and some varactors.

The ARRL Antenna Handbook chapter 4 talks about loop antennas in some detail.

Zungeru *et al.*

Zungeru et al. describe several previous RF energy harvesting

systems in the μW range.

Xie *et al.* 2012

Xie et al. 2012 report harvesting 82 μW using a 10-meter-long horizontal antenna, even though that was just a dipole, and one with the wrong polarization, at that.

The Xie paper contains circuit diagrams and various plots of output voltages, currents and powers. The circuit was extremely simple! One end of the antenna was hooked up to a parallel-resonant LC tank (with, ideally, infinite impedance to ground at the resonant frequency) with a "multi-stages doubler rectifier" hooked up across the inductor (Figure 4 shows that this is a Cockcroft–Walton generator made out of six 1N60 germanium Schottkies and six 0.1-μF caps), feeding an energy storage capacitor to which the microcontroller is directly connected. They report getting 14 volts out of the damned thing — *before* the Cockcroft–Walton 6× multiplier! There in Xi'an, there's apparently a 300-kW station 23 km from their site, but they got more power from a 50-kW station at 8.5 km — that's where the 14 V came from. (But it's only 1.2 V without the resonant tank.)

Leon-Gil *et al.* 2018

Leon-Gil *et al.* 2018, "Medium and Short Wave RF Energy Harvester for Powering Wireless Sensor Networks", CC-BY also used a resonant tank circuit and a Cockcroft–Walton generator to step up their antenna voltage, but this one of 6 stages, using a center-tapped loopstick configuration. (Note that the PDF of the paper is not readable with libpoppler; xpdf can read it.) (They also mention that TV broadcasting stations are up to a megawatt in the US, 20× stronger than AM broadcast stations, but are not available in rural areas.)

Leon-Gil *et al.* also stacked up layers of ferrite in order to get a giant 60 mm × 60 mm antenna cross-sectional area in a compact volume, because (they say, I trust) the radiation resistance of the antenna is jointly proportional to the square of the area and the number of turns, and, astonishingly, not dependent on the length of the rod. They also point out that diode parasitic capacitances provide an AC leakage path that limits the total voltage available from a Cockcroft–Walton generator, which I hadn't thought of; these parasitics decrease with back-biased voltage, but need to be small compared to the intended capacitance. They used BAT85 diodes, which specify a 200-mV voltage drop and 5-pF junction capacitance, and found that 10-nF capacitors or larger were needed to avoid inefficiency.

They reported 1.5 volts at the input to their Cockcroft–Walton generator and 8 V at its output, providing about 60 μW into a 1-MΩ output impedance — quite impressive for the 100-mm-long antenna they're using, 100× shorter than the wire in the Xie paper! They report a 3.2% antenna efficiency.

They report that using the center-tapped Cockcroft–Walton configuration doubled their output power over what they describe as a "half-wave" configuration, but I suspect that this is because they were only using half of their winding when they were testing the "half-wave" configuration. In a Falstad simulation, I see in-phase

current and voltage both positive and negative in the configuration they describe as "half-wave", so I think their description is erroneous.

Dyo *et al.* 2013

Dyo et al. at the University of Bedfordshire wrote "Design of a ferrite rod antenna for harvesting energy from medium wave broadcast signals" in 2013 (doi: 10.1049/joe.2013.0126). They used an 8-stage Cockcroft–Walton generator made out of Schottkies driven from a loopstick antenna with an LC resonator tuned to a local 150 kW radio station at 909 MHz; in simulation they generated over 1000 µW, up to 9 km away, but apparently they only tested it by powering a 3 µW clock. Their paper is the only one of the ones I'm summarizing here that has a decent review of the existing work in the field, including the Xie et al. paper.

They point out that actually the length of the rod does matter, and the peak for a cylindrical rod is when the rod is 20 times as long as its diameter; its effective permeability rises almost linearly from aspect ratios of 1 to 10, half of the ideal aspect ratio of 20, nearly doubling its value; it gains another 10% going from 10 to 20.

They report a cost-optimized antenna design: they ended up with a 10-mm-diameter ferrite rod that was 200 mm long (a size you might find in a normal AM radio), with 50–500 turns of litz wire wrapped around the middle of it, depending on how weak the signal was — weaker signals needed inverse-proportionally more turns to get the same output voltage. For a sufficiently weak signal, the coil's self-resonant frequency falls low enough that the design is no longer viable.

Strangely, their optimization results table reports that wire length s varied from 67 meters to 787 meters. These are consistently about 45× the wire length I calculate via $n\pi D$, where n is the number of turns and D is the diameter. This seems to be a result of their using 45-strand litz wire.

Low-power electronics design

How much power do you really need to harvest in order to have a usable computer?

I've written a fair bit on low-power computers in Keyboard-powered computers (p. 2220), Low-power microcontrollers for a low-power computer (p. 2602), and Notes on the STM32 microcontroller family (p. 3176), and just today I encountered the oversold-as-low-power Renesas RL78 microcontroller line (p. 504), although that was disappointing in the end. The summary is that you can get useful amounts of computation at around 1 µW, which is a bit smaller than the self-discharge current of a CR2032 coin cell, which contains about 2.2 kJ. Understanding Capacitor Leakage to Make Smart Things Run Longer explains that you have to be careful to ensure your bypass capacitors don't have more leakage than that if you're designing, say, a passive IR sensor designed to run for ten years maintenance-free from a coin cell, but that you can do it by using capacitors rated for 10× higher voltages than you need, which will be larger.

Linear regulators dissipate the voltage difference between the input voltage you get and the output voltage you want; this is wasteful if

there is a large difference. Below I suggest using a buck converter so that your power usage is the same regardless of the input voltage — this works by drawing current less of the time when the input voltage is higher. But buck converters require a minimal quiescent current to remain stable, which is also wasteful. A possible solution is to use a linear regulator in sleep mode, then turn on the buck converter when waking up.

Xie et al. used 1N60 germanium Schottkies. Germanium diodes are a bit exotic nowadays, but the 1N60 is specified to have a threshold voltage of only about 250 mV. Schottkies in general have high leakage currents, and Taitron lists their 1N60P as leaking up to 50 μA, which would be disastrous for this application, but apparently they didn't observe such high leakages. Regular *junction* germanium diodes, on the other hand, have a threshold voltage of around 300 mV, and would be ideal.

Aside from low-leakage capacitors, low-self-discharge batteries, avoiding power supplies with high quiescent current, low-leakage diodes, and using low-power chips, it seems like the main trick to low-power design is to use a low duty cycle — keep your chip asleep most of the time. This means you can't be running either a radio or any other kind of ADC at high speed all the time. With typical wakeup times of 5 μs in modern microcontrollers, though, you could reasonably wake up 1000 times a second and still keep the duty cycle below 1%.

If you can hear a radio station around 10 μW/m², then it ought to be able to broadcast a signal around 100 μW and be audible on radios in the same room. Of course this requires a duty cycle of 1% or less if you're only harvesting 1 μW. This suggests that you can probably use the same antenna for harvesting and transmission — you perhaps must switch off the harvesting circuit during transmission, but that will only reduce the power harvested by 1% or so. You might want a shorter winding, or a center tap, to reduce the antenna's impedance, improving its efficiency at low voltages.

You could try the Xie et al. circuit or the Leon-Gil circuit without the resonant tank (thus receiving all frequencies indiscriminately) or perhaps with a few different parallel tanks in series to ground (thus, I think, receiving just those particular frequencies). Perhaps a better idea is to use an RF transformer on the input side to step up the antenna voltage to a more reasonable level; with a loopstick antenna this can be effected simply by having MOAR TURNS on the antenna, as Dyo et al. did, but only up to a point, so a transmission-line transformer or two to step the voltage up might be a reasonable option.

Stepping up the input voltage with a transformer should allow the use of regular signal diodes (which reportedly leak about three orders of magnitude less than Schottkies) or even special low-leakage types (which leak about three orders of magnitude less than regular signal diodes).

Variable-gain energy-harvesting Cockcroft–Walton generators

I had a silly idea that maybe you could get a variable multiplication factor out of a Cockcroft–Walton generator, and thus a sort of analog

MPPT circuit, by adding some more diodes to it, bypassing the later stages when the output storage capacitor voltage is low. Such diodes are relatively harmless but turn out not to be useful — they never conduct, even though their voltage drops are lower than the voltage drops of the chains of diodes they bypass. That's because all the capacitors in the Cockcroft–Walton generator charge simultaneously and discharge simultaneously, across cycles. (Within a single AC oscillation, they charge and discharge at different times.) The Cockcroft–Walton generator already gives you a variable voltage multiplication factor out of the box!

I *think* the effect is that if you load a Cockcroft–Walton generator's output so that it never approaches its maximum possible voltage, you in effect decrease its impedance. Assuming the input voltage is large compared to the sum of the diode drops, if you load it so heavily that only the output capacitor ever develops any charge, it becomes effectively a long diode, a half-wave rectifier; then, if you let it charge, its AC impedance gradually increases toward, ideally, ∞, because there is no purely capacitive path through it — all the paths through it go through the diodes.

I don't understand why it was that the Leon-Gil experiment needed 0.01-μF capacitors; calculations based on capacitor impedance suggest that even a much smaller capacitor should have been sufficient, given the 1-MHz frequency and the total output current of about 6–8 μA (62 μW ÷ 8 V, or 8 V ÷ 1.5 MΩ).

The Cockcroft–Walton generator's distributed capacitance works as energy storage; you can draw it down in a balanced fashion by temporarily drawing a larger current from the output. A buck regulator running from the Cockcroft–Walton output could efficiently handle a wide range of input voltages and thus enable efficient circuit operation from very low amounts of stored energy up to very high amounts.

As the capacitors become fully charged, the power factor shrinks — there is only current at the very peak of the wave. This suggests that, in the absence of resonance, higher output can be maintained from a Cockcroft–Walton generator that isn't fully charged. For example, with a multiply-by-six configuration — six diodes and six capacitors — if the input voltage is a 10-volt-peak AC wave, and the output voltage is 30 volts (feeding, say, a buck regulator producing a 2.0-volt power supply for a microcontroller), then whenever the wave is over 5 volts, the power supply will draw charging current, thus harvesting energy in phases from 30° to 150°, then 210° to 330°, ⅔ of a full cycle. If the output voltage is allowed to rise to 40 volts, it will only be charging for 54% of the cycle, and if allowed to rise to 50 volts, only 37%.

(I'm not sure if this depends on how the capacitance is distributed throughout the circuit.)

The above was observed simulating the circuit with a source with 500 Ω of internal impedance (because a more realistic amount made it charge very slowly, as you'd expect). Upon simulating with low source impedance, the charging current peaks happen right after the input voltage crosses zero, and indeed at first have periods of time where the circuit returns power to the source — as you would expect from a capacitive load. This ceases once the output voltage has risen to the input peak voltage plus the combined voltage drops of the

capacitors.

It seems like a little input series inductance would do some good for the circuit's power factor and power dissipation — with 7 100-nF capacitors and 7 300mV Schottky diodes at 500 kHz, it charges more than twice as fast (for the first 25 cycles, which get it to about 5× the input voltage) with a 2.2-µH inductor in series with its input. Probably better to err on the low side there with a 1-µH inductor or an 0.47-µH, though — the asymptote at low inductances (or low frequencies) is just the situation described above, while the asymptote at high inductances or high frequencies is that zero energy gets into the circuit.

Suppose the input voltage source has very low impedance; what limits the power it can supply? I think it's just the amount of charge that gets loaded into all the capacitors each cycle, so it's proportional to frequency and proportional to the square of the input voltage — it's just the impedance of the straight capacitive path to the output, plus a diode. (Except if the output voltage is lower than the instantaneous input voltage, when its impedance is the Vf of the the string of diodes.)

I think this means that the Cockcroft–Walton generator is a good approximation to the circuit I was trying to find in Constant current switching capacitor charging (p. 605)! Suppose you limit the output of a six-stage device to 50 V (with a "zener" avalanche diode or whatever). Then it will transmit power with reasonable efficiency from the input to the 50-volt output for an input AC voltage anywhere in the range of about 10 volts to 50 volts (peak), and furthermore for an output DC voltage anywhere in the range from just below the input peak voltage up to those 50 volts — it's like a transformer that dynamically changes its turns ratio according to the current draw! Like, uh, a switchmode power supply. More stages will increase the diode losses but, for a sufficiently high output voltage, will reduce the required input voltage.

It isn't exactly an MPPT circuit, since its impedance starts too low (though, unlike a simple rectifier charging a storage cap, not near zero), rises to the maximum power point, and then keeps rising; but all of the input energy is either dissipated in the diodes or stored in the capacitors, and with a little input inductance, it even avoids wasteful inrushes.

Disregarding diode capacitance, you can model the basic characteristics of a Cockcroft–Walton generator as follows.

$$V_{out} \le N(V_{in} - V_f)$$
$$P_f = N V_f I_{out}$$
$$P_r = V_{out} I_r$$
$$P_{out} = V_{out} I_{out} = \int V_{in} I_{in} \mathrm{d}t/\Delta t + P_f + P_r$$

V_f is the forward voltage drop of a diode at the low currents we're dealing with here, around 600 mV for a 1N4001 or 300 mV for a Schottky. P_f is the total power loss from the diode voltage drops for forward current; P_r is the total power loss from diode leakage resulting in reverse current; N is the number of stages, each containing a diode and a capacitor; I_r is each diode's average reverse current. V_{in} above is variably the peak voltage of the AC waveform (in the first inequality) and the instantaneous voltage in the integral.

The first thing to notice here is that if $V_{in} < V_f$, the circuit won't work at all, regardless of how many stages it has, and that's why so

many of these papers use Schottky diodes; and if $V_{in} > V_f$, it will, again regardless of the number of stages — and that's how the humans first split the atom.

Second, the *forward* power losses in the circuit are proportional to the number of stages. So if you have six 600 mV stages, they're going to drop 3.6 volts; if your output voltage is 3.6 volts then you are going to lose half of your energy inside the generator. You should probably use less stages. The *reverse* or leakage losses, though, actually *decrease* with the number of stages, for fixed input voltage, output voltage, and load current, because the diodes leak less when that fixed output voltage are divided across more stages. (This seems to contradict results in the Leon-Gil *et al.* paper, so I may have gotten something wrong.)

So, with few enough stages, the losses will be dominated by reverse losses, and you should use low-leakage diodes; and with enough stages, they will be dominated by forward losses, and you should use low-V_f diodes, even if they could leak more.

The leakage currents in the diodes depend on the reverse-bias voltage, and the reverse voltage across any diode varies interestingly. It is always less than the peak-*to-peak* voltage of the AC input, but it varies during the cycle, and in a way that depends on circuit load. If the circuit gets fully charged, all the diodes are always reverse-biased or anyway not forward-biased enough to conduct any more, and so they see the full AC waveform with just enough DC bias to make it never quite conduct, or conduct just enough to compensate for reverse-biased-diode leakage; so, for example, if you're using 600-mV diodes and the input is a 10-volt-peak (7.1 V rms) sine wave, each diode sees a reverse-biased voltage oscillating between -19.4 V and +0.6 V.

When the current isn't fully charged, the diode sees a sine wave that's *clipped*: the negative bias is more moderate, so when the voltage tries to rise above its threshold, it starts conducting and clips the peak off the voltage wave; and the wave's trough voltage gets clipped by the diodes that conduct during the opposite half cycle.

Since diode leakage currents increase, very roughly, exponentially with the reverse-bias voltage, probably nearly all of the leakage will happen at the trough of the wave, and the wave spends nearly half of its time there. So, for example, if you have a 6-stage circuit whose output voltage is currently 24 volts, each diode has -8 volts across it half the time. So if each diode leaks 25 nA at a reverse bias of 8 volts, the whole circuit will leak about 12.5 nA and 0.3 μW. (This is a normal number for a silicon signal junction diode.)

Here's a Falstad circuit for simulating a 7-stage Cockcroft–Walton generator:

```
$ 1 5.0000000000000004E-8 1.8479586061009856 56 5.0 50
v 432 416 432 576 0 1 500000.0 2.0 0.0 0.0 0.5
d 432 336 528 176 1 0.3
c 432 336 624 336 0 1.0000000000000001E-7 -0.07696510389787246
c 624 336 816 336 0 1.0000000000000001E-7 -0.07127146119106348
c 816 336 1008 336 0 1.0000000000000001E-7 -0.06854028985446714
c 528 176 720 176 0 1.0000000000000001E-7 -0.06854028985446775
c 720 176 912 176 0 1.0000000000000001E-7 -0.07127146119106291
c 912 176 1104 176 0 1.0000000000000001E-7 -0.07696510389786876
```

```
d 528 176 624 336 1 0.3
d 624 336 720 176 1 0.3
d 720 176 816 336 1 0.3
d 816 336 912 176 1 0.3
d 912 176 1008 336 1 0.3
d 1008 336 1104 176 1 0.3
c 1104 176 1104 576 0 1.0E-7 0.12035737918464492
w 1104 576 432 576 0
w 1104 176 1216 176 0
w 1104 576 1216 576 0
g 432 576 432 624 0
r 432 336 432 416 0 500.0
x 239 377 407 380 0 12 internal source impedance
r 1216 576 1216 336 0 100.0
s 1216 176 1216 336 0 1 true
x 1262 461 1289 464 0 12 load
o 7 512 0 290 320.0 0.4 0 -1
o 2 512 0 290 320.0 0.4 0 -1
o 16 1 0 35 0.15625 9.765625E-5 1 -1
o 0 1 0 35 9.353610478917778 0.005846006549323612 1 -1
o 16 64 0 35 0.3125 9.765625E-5 2 -1
```

Something in the neighborhood of 6 stages is probably optimal for this purpose, and a maximum voltage of 50 volts is probably also a good idea (to avoid needing high-voltage components), so you should probably step up the antenna voltage to about 8 volts (with an RF transformer, if necessary) and maybe limit it with a small zener for safety. (Or build the damned thing out of zeners.)

Robustness

Since the available energy may vary over one or more orders of magnitude depending on location, radio stations starting up and shutting down, time of day, ionosphere propagation, nuclear warfare, and so on, it's probably worthwhile to have some kind of overvoltage protection across the antenna. It likely only needs to be able to dissipate milliwatts of power, so a couple of back-to-back small "zeners" (avalanche diodes) might be adequate, and maybe a MOV or two for nearby lightning strikes or hydrogen bombs or something, plus some inductance before the Cockcroft–Walton generator to prevent spikes too fast even for the MOV from making it through.

Passive signaling

Above I suggested transmitting AM audio at 100 μW or so for the sake of communicating to humans using ordinary AM radios, which has been done on a battery. But suppose we only want to communicate with other energy-harvesting nodes?

Some work has been done on passive signaling using Wi-Fi for sensor motes and even passive sensors, analogous to Theremin's masterwork, The Thing. As I understand it, the mechanism is that if you have an antenna tuned to a particular frequency feeding into a load (such as a Cockcroft–Walton generator), you absorb some energy at that frequency; if you disconnect the load, the radio waves reflect from the antenna (perhaps with a phase shift) instead of being absorbed. So someone in the vicinity listening on that frequency can

detect that you've turned the antenna off.

Suppose it's possible to get reasonable antenna efficiency across a wide range of frequencies, like the whole AM band, as would be ideal for energy harvesting, and so you can be absorbing tens of microwatts as the Leon-Gil experiment did, or even more. To a nearby observer "tuned" to the same set of frequencies, temporarily disconnecting your harvesting input would be precisely as observable as beginning to transmit noise at those same tens of microwatts. In effect, this amounts to a time-domain radio transmitter with 100% efficiency — "111% efficiency" if your power supply is 90% efficient. This is probably an order of magnitude or more better than you could do by actually transmitting a signal, but the range will be very limited, because you can't concentrate power harvested over time at a particular moment.

If you want to transmit, say, four meters, by failing to absorb 50 μW and instead reflecting it isotropically, your lack of signal will be noted as 250 nW/m² at that four-meter range (assuming far-field radiation, which isn't actually correct — at 1 MHz, where the wavelength is 300 m, energy can bounce back and forth between the nonlistener and the observer of nonlistening some 37 times per oscillation, so I think the near-field coupling can be stronger than this). This is about 320x lower than the "background noise" we're assuming of 80 μW/m², although it seems likely that Leon-Gil et al. were more strongly irradiated than that. This is a SNR of -22 dB and accordingly requires at least 22 dB of coding gain to establish communication; each additional factor of 10 in distance adds another 20 dB.

(Unfortunately I don't know how to calculate near-field electromagnetic coupling to a reasonable degree of precision; I'm pretty sure it's stronger is all.)

The Shannon–Hartley formula is that the channel capacity in bits per second is $C = B \lg (1 + S/N)$, where B is the bandwidth in Hertz, S is the signal power, and N is the (additive white Gaussian) noise power. So in this case it's 1.2 MHz · lg (1 + 1/320), which is basically 1.2 MHz · 1/(320 ln(2)), or about 5.4 kbps. In theory, this drops by a factor of 100 with every factor of 10 in distance, in the limit: 54 bps at 40 meters, 0.54 bps at 400 meters, 0.0054 bps at 4 km. I think this is hit harder by the near-field effects, though — you get terrible antenna efficiency with an antenna that's a tiny fraction of a wavelength in length (it reabsorbs almost all of what it emits, in a sense), so the situation is orders of magnitude worse.

Down here in the power-limited regime, it doesn't really matter whether your signal is spread over 100 kHz or 1 MHz or 10 MHz — if you spread it over ten times as much bandwidth, you also get ten times as much noise, which compensates to within a small rounding error. But if you narrow the signal to, say, 10 kHz, you start to see an effect: 10 kHz · lg (1 + 1/2.7) ≈ 4.6 kbps — you're entering the bandwidth-limited regime. This is counterproductive from the point of view of power efficiency — you get less bits per joule that way.

The power input circuit I simulated above generates very substantial harmonic distortion (though less so with a bit of PFC inductance) — the sharp current edges from the diodes turning on are potentially quite noticeable, and I think the strongest signals would probably be the second and fourth harmonics.

Measuring input

Suppose you have something like the above setup and you want to periodically measure the energy in the radio spectrum, perhaps because you are trying to decode a signal sent by another such node. You have to be careful because the total amount of signal is small enough that the normally-negligible bias currents of analog inputs could consume a significant fraction of your harvested power! A low duty cycle might be adequate.

The voltages, however, are large enough that you probably want to divide them down — the original input voltage can be as high as 8 volts and you want to divide it down to, say, 1.1 volts or less. Maybe you could use some kind of capacitive divider that you then rectify and analog-filter the output from, with a multi-megohm resistive path in parallel with it to prevent unequal leakage in the capacitors from introducing a DC bias.

Active signaling

If your device is powered by energy harvesting from AM radio, the average bit rate you can achieve via active signaling is necessarily lower — the power you would have passively reflected must instead be harvested an used to power a radio transmitter, which is typically less than 20% efficient. However, you can save up that energy and transmit it in bursts, possibly in a frequency band where this is a legal thing to do, and possibly at a predetermined hour so that the intended destination node will be listening. (Sort of like FidoNet "mail hour": whatever its other hours of operation, you would leave your FidoNet node turned on during your zone's "mail hour" so that other nodes could call it and deliver its waiting mail.) So if you're harvesting 50 μW, you could save up 4.3 joules in a day, and transmit, say, 300 mJ of radio energy during a single second, thus transmitting at 300 mW — a million times stronger than the passive near-field communications numbers above, and thus in theory able to transmit a thousand times as far.

(Elaine Chao explained to me that she wrote such a time-domain multiplexing system for the Motes project at Berkeley — as I understand it, each node listened in a particular fairly-coarse-grained time slot, and transmitted in the time slots of the nodes it wanted to talk to.)

Saving up 4.3 joules requires somewhere to put it, though. If you try to put it in $\frac{1}{2}CV^2$ with $V \leq 50$ V, you need $C \geq 3400$ μF, which is a hell of a lot of capacitance — and, in particular, the electrolytic capacitors that are the best bets for this kind of massive capacitance are leaky as shit.

It's clear, though, that long-distance communication needs either some kind of highly directional reception, a better power source, extremely infrequent communication, much bigger antennas, or all of the above. Frequency bands like the one I've been talking about are ideal for over-the-horizon communication because of skywave propagation.

(As mentioned above, though, the 20× larger energy available from TV stations is a potential better power source, and their higher frequencies would avoid the need for bulky ferrites; PV cells are another possibility, as a 21%-efficient 10 cm × 10 cm cell can generate 2.1 watts in full sun — that's 2,100,000 μW, enabling four orders of

magnitude higher communication bandwidth.)

Fractal loopstick antennas

The field strengths we're talking about here are far too low to saturate our ferrite cores. Some research has focused on optimizing the core shape to gather more flux, for example by using a barbell or hyperboloid shape, but it seems likely to me that a ferrite core that branches out at both ends in a fractal branching pattern, something like a pair of sprigs of parsley or a pair of antlers, would have a better flux-gathered-to-weight ratio than such solid shapes. Fabricating such shapes has not been practical in the past, but you ought to be able to do a reasonable job with a ferrite-filled plastic system, carrying powdered ferrite or even powdered iron in a small amount of something like ABS or PLA.

Packaging for durability

PVC plumbing pipe is a durable, shock-resistant, sunlight-resistant, waterproof, and visually unremarkable package into which you could stuff even a fairly thick ferrite rod and attach the device to the side of a building; you could reasonably expect it would stay there for many years. To seal the package entirely hermetically, you could seal plugs in place some distance inside the ends of the pipe, using, for example, silicone, epoxy, or PVC glue.

As a larger-scale alternative to using a ferrite-core loopstick antenna, you could use a just-plain-loop antenna, with a coil of wire running around a larger area than is feasible to fill with ferrite; it, too, could be sealed in PVC pipe, perhaps in a square shape. Loop antennas have nulls perpendicular to the loop plane, so a reasonable orientation might be horizontal, since very few high-powered AM radio transmitters are near the zenith; for example, you could run the loop around the perimeter of a roof. By enclosing, for example, 100 m² of loop area, you can increase the received (or conditionally received) power by 5½ orders of magnitude, minus the one or two orders of magnitude you lose from not having the ferrite. This could boost the power available to such a system up near a watt.

Since the circuit's coupling to the environment is magnetic, shielding it is not practical, so it could reasonably be embedded in concrete, even underground. The skin depth $\delta = \sqrt{(2/\omega\mu\sigma)}$ is around 16 meters in rock, according to David Gibson's "AM Radio Reception in Caves", so even several meters of earth or concrete would barely weaken it. So burying the loop antenna in a ditch, perhaps armored with concrete, is another option to increase the durability of the device.

Topics

Double heap sequence

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

Emacs stores the text in a buffer in a buffer-gap representation: the text before the cursor is in one contiguous block, the text after it is in another contiguous block, and there is a slack space in between. This makes inserting or deleting text at the cursor very fast, while moving the cursor is somewhat slower, involving copying the text that the cursor moves past to the other side of the buffer gap. This is not a very fashionable way of doing things nowadays, since it involves a lot of mutation in the course of what you'd think would be read-only operations, but in practice it works very well indeed.

I was thinking that a possibly interesting representation for mutable *sorted* sequences is a somewhat analogous thing, one which maintains two *heaps* for the elements before and after a "cursor". To move the cursor forward, you pop an item off the after-heap, which is a min-heap, and insert it into the before-heap, which is a max-heap. To move the cursor backward, you pop an item off the before-heap and insert it into the after-heap. You can insert and delete items from these two heaps as well. All four of these operations — forward, back, insert arbitrary, and delete arbitrary — have logarithmic worst-case and average-case time. Initially building the heaps from an unsorted set of items (with a given key as the cursor position) takes linear time. No external space is needed.

The more normal way to support these operations would be using a B-tree. Is that better? It gives you logarithmic insertion and deletion time, yes, but it takes N log N time to build the B-tree in the first place, and it takes potentially substantial space to build. It also gives you some goodies that the double-heap does not: logarithmic-time access to find a given key and constant-time iteration.

Topics

• Programming (p. 3658) (286 notes)
• Performance (p. 3621) (149 notes)
• Algorithms (p. 3310) (123 notes)

How to use “correct horse battery staple” as an encryption key, including a recommended 4096-word list

Kragen Javier Sitaker, 2014-04-24 (44 minutes)

You can use a passphrase consisting of four short, common words to keep your encryption key secure from attackers willing to spend less than US$1B to crack them. Just choose the passphrase with strong randomness, salt it, then spend a minute stretching the passphrase with scrypt. Memorizing such a passphrase with an optimal practice schedule takes several minutes, but only once in your life.

For this to work, you absolutely cannot make up the passphrase yourself with your mind. You have to use a truly random physical process such as rolling dice.

Background

xkcd clearly showed that for online password attacks, a password like "correct horse battery staple" is perfectly adequate — that is, a password consisting of a small number of randomly chosen common words, with spaces between them, and no unusual orthography — while more common methods of choosing passwords are less secure; and furthermore that these passwords are easier to remember. But a footnote on the comic explained that this may not be adequate for resisting offline attacks.

(To those who wonder whether they should be taking security advice from a comic strip: ① the argument and calculations in the strip are correct, and ② you clearly don’t read xkcd. Also, in that case, you probably need to be told not to make up the passphrase with your mind; use Diceware or something similarly strong. Also you probably need someone else to write the software to implement this for you.)

In a chat with a friend of mine yesterday, I realized, much to my surprise, that this kind of password is also adequate against *offline* attacks, like those that someone can try against encrypted text they captured or against private keys, given certain conditions.

The approach is this. You take a randomly chosen passphrase, run it through a password-based key derivation function on a computer you trust for a reasonable amount of time, and use the resulting output as the encryption key.

("Brain wallet" is a broken implementation of this idea as it applies to Bitcoin. Most brainwallet-stored Bitcoins have been stolen.)

How much entropy do we need?

Let’s suppose we want the key to be secure against a billion dollars’ worth of attacker resources. A million is probably okay for most people in most situations, but a billion gives us some margin of safety. I calculated some computing costs in 2010; in particular, running a regular personal computer cost about US$0.005 per minute, which

I'm going to figure is probably still about right.

The most advanced current password-based key derivation functions, like Colin Percival's 2009 scrypt, minimize the cost advantage of special-purpose attack hardware relative to a regular personal computer; let's figure that they keep it down to 8×, that is, that you can duplicate a minute's worth of scrypt computation for ⅛ of US$0.005 if you're using special-purpose hardware. (I think that by using a custom packet-switched memory fabric like the Tera MTA, you can probably beat that advantage by an order or two of magnitude, but probably only the NSA has done this so far.)

So let's suppose that you, the legitimate user, are willing to wait for one minute, with one computer, to hash your password. That is, you're spending ½¢, but you want your attacker to have to spend US$1B. This means you need to choose one of about 400 billion passwords, i.e. you need 39 bits of entropy. Plus three bits for the custom hardware advantage, for 42 bits.

"Correct horse battery staple" passwords can easily have 44 or 48 bits of entropy, using only four words.

In effect, key stretching is adding 30–50 bits of difficulty to breaking your key.

Memorization schedule

For a password like this that you never write down, even if it's as memorable as "correct horse battery staple", you need to establish a practice schedule so that you don't forget it, and it would be very helpful to have spaced practice software for this. Existing spaced practice software for memorization (things like AnyMemo, Anki, or Mnemosyne) is not suitable unless you're already using it. It's designed for memorizing a large set of facts, not a single one, so it only quizzes you when you ask it to run.

By contrast, in this case, you need to space your practice of just one password. I don't know of any software for this. When I did this with a seven-word passphrase, I set an alarm on my phone repeatedly for the repetitions in the first day, and thereafter repeated the exercise each day for a long period of time by just using the password each day:

- 10:31: generated password; repeated it until I could remember it for a few minutes.
- 10:35: first practice.
- 10:37: second practice.
- 10:47: third practice.
- 11:30: fourth practice.
- around 16:00: fifth practice.
- That night: sixth practice.
- The next morning: seventh practice.
- Two days later: eighth practice.

The intervals here are a bit more conservative than the usual 1, 5, 25, 125, 625, 3125 intervals for practice memorization, for two reasons:

- The passphrase, although I impose linguistic associations onto it, doesn't really have an underlying structure that simplifies it; so it's inherently a hard thing to memorize.

• If at some point I forget the passphrase, I don't have the opportunity to recover it from some reminder medium, as is usually the case for things I'm trying to memorize. I can't just turn over a flash card. If I forget it once, and it's not still in my terminal scrollback, it's gone.

On the fourth try, I misremembered the passphrase at first, suggesting that the schedule may not be conservative *enough*. The exponential factor in that case was about 3½, so you probably want something more like a factor of 2½ to be safe: after repeating the password for 2 minutes, wait 3 minutes, 7 minutes, 20 minutes, an hour, 2 hours, 5 hours, 12 hours, a day, 2 days, 5 days.

(This probably means you want to start this process in the morning, not at night.)

Checksums

If you're going to wait for an entire minute to hash your password, it's kind of a bummer if it turns out you mistyped it. This is avoidable at the cost of slightly longer passwords. If the password is generated in a way that ensures that, say, the checksum of its 4-bit nibbles will be 0, then you have decreased the strength of the password by 4 bits, while ensuring that 94% of the time that you type the wrong password, the system can detect it immediately.

If the key-stretching program also knows the wordlist from which your words were originally chosen, it can probably also catch most typographical errors.

This is actually crucial for memorization; because of hyperbolic discounting, when you're practicing your password, you need immediate feedback if you get it wrong. (Hyperbolic discounting is somewhat questionable when applied to people making choices consciously, but firmly established in operant conditioning, which is what we're dealing with here.)

Alternatively, you could simply use a longer password and less key stretching, so that the key stretching process ends in around a second.

Moore's Law

A billion dollars of computing effort today is half a billion dollars in a year and a half, if Moore's Law continues. In 15 years (in 2029), it will diminish to only a million dollars. It's uncertain as to whether this trend will continue. It might even speed up: solar photovoltaic energy will probably overtake fossil-fuel energy as a fraction of our marketed energy in the early 2020s, and subsequently energy will become cheaper even faster than computing hardware.

Each extra bit of password strength gives you about a year and a half of Moore's Law lead time.

In cases like public keys used to authenticate forward-secret key exchanges or authorize transactions, being secure against future attacks is irrelevant. In cases like the encryption used for your disk, it is not. You should probably assume that breaking your disk encryption will become practical at some point, but you can probably delay that day by using longer passwords.

Believe MC Frontalot: you can't hide secrets from the future.

Quantum computers

Speaking of defending against future attacks, what about quantum

computers?

Grover's algorithm on quantum computers, if and when those turn out to be feasible to build, halves your effective key length. If applied to your 48-bit pre-key-stretching password, this would reduce its strength by 24 bits, which would make it feasible to break. You could use a 96-bit pre-key-stretching password (eight words) to resist this attack.

I'm pretty ignorant about quantum computation, so I don't really know, but I don't think you can apply Grover's or any other known quantum algorithm to reduce the effective length of your stretched key — the log of the number of possible passwords plus the log of the number of operations used in stretching — by half. If you could, then if your key stretching involves a trillion operations (about 2^{40}) and you want to resist a 2^{80} attack, you need an effective key length of 160 bits, of which 120 will have to come from the password: ten words.

Massive attacks and salt

An attacker can gain some advantage over the defender by attacking many passwords in parallel. With the approach described so far, if there are, for example, 256 law enforcement officers who receive an encrypted email, and the mafia successfully steals all of their computers, the mafia doesn't need to spend a billion dollars to decrypt one of the officers' disks. Instead, they can spend 1/256 of that to decrypt *any* of the officers' disks: they try each generated key against all 256 disks. This way they can get the email after only a few million dollars' worth of work.

(Better, if someone does the billion dollars' worth of work just once, they can make a rainbow table of all possible keys, although with such a slow key derivation function, the rainbow table chain lengths will be limited. For example, with a chain length of 256, you'll need 2^{40} entries for 2^{48} possible passwords, occupying a total of 6 tebibytes; but doing a single lookup in the table will take 2 hours.)

The standard approach to solving this problem is to salt the passwords: store a nonsecret random string that gets combined with the password (for example, concatenated with it) before you run the key derivation function. Many pieces of software will do this salting for you, but if not, you can do it yourself. For example, for a disk encryption password, you can engrave the salt into the case of your computer, next to the keyboard. For example, your 48-bit salt might be encoded as "sting vowed woken hold" while your password is "crowd lasts men woody"; you type "sting vowed woken hold crowd lasts men woody", having copied the first four words from the engraving.

Being interactively prompted with the salt would probably reduce the incidence of accidentally typing the wrong password, a phenomenon which compromises that password.

Sequential memory-hard algorithms are a wrong turn in this context

The scrypt paper argues that sequential memory-hard algorithms give defenders an extra advantage over attackers: by increasing the amount of time needed by a defender to hash their correct password by some factor N, you increase the cost to an attacker by N^2, because

the attacker needs both N times more memory and to use it for N times as long.

XXX

Timing and other side-channel attacks

Memory-hard functions like scrypt, while necessary to resist custom hardware, probably cannot execute in constant-time on off-the-shelf hardware, because they need to generate and access large tables. But because the total time for the computation is so long, and because it can't be automatically initiated by an attacker request and remote attackers can't normally observe either the beginning or the end of the process, timing attacks should in general be very difficult to carry out.

Using phrases of English words rather than arbitrary characters adds a great deal of redundancy to the password, which means that even very minor amounts of side-channel leakage (such as keystroke timings, audio of keystrokes, or RF emissions from the keyboard) should make it possible to completely reconstruct the password.

I have no idea about power analysis.

My preferred wordlist

The S/KEY wordlist from RFC1760 is commonly used; it consists of 2048 words of up to four letters. It has a couple of flaws: it's only 2048 words (11 bits per word), when 4096 is easily achievable; it includes many very uncommon words, such as "Egan", "Eben", and "fogy", impeding memorization; and it contains many words that are too similar, such as "good", "gold", "goad", "goal", "coal", "foal", and "goat". For example, the 48-bit number 155759005738413 is encoded using S/KEY as "aid shin mini ruse made", except in all capitals. My preferred wordlist, instead, encodes it as "hay fork catch diary", which I think is dramatically more memorable.

Here are ten more S/KEY encodings of 48 bits, but in lowercase:

```
air limb dash mask laws
ace cock rip fond bask
an shy lind juno folk
all doug jolt lang sob
aim tank bed neil juno
an fawn wean aqua if
ada ding sea hive kit
ada lies nip aids howl
ada cut love will hum
am iris gut din bold
```

Here are ten encodings of 48 bits with my wordlist:

```
ici shrug slow solid
must fever plug hotel
facts foam gall frail
snake erika star i
belle feat dip waste
rent herr freud tuna
annie spade downs doo
grief sense infer cab
```

cuba g maths pig
days dusty small yacht

I generate these with bitwords.py, which uses the system truly-random-number generator, which is almost as trustworthy as rolling dice if you're on a Linux machine that isn't backdoored.

I derived the following wordlist by taking the most common 4096 words of 5 letters or less from the British National Corpus. It's imperfect, as you can see above; it still includes some nonwords, uncommon proper nouns, and words very similar to one another. In practice, though, as you can see from the above, the words it generates are pretty memorable. Many of them are sentences or noun phrases, or almost: "Rent Herr Freud tuna! Must fever plug hotel? Grief sense infer[s] cab. Cuba G: Maths pig! Day's dusty small yacht. Annie spade-downs doo."

It also has the problem, shared with the S/KEY list, that it contains many words that are too similar to one another: it contains not only "sold", but also "old", "told", "hold", "cold", "gold", "bold", "solid", "sole", and "solo". This means that a typo or misremembering of a password is often also a valid password.

Previously I used the most common 4096 words in English (again, according to the British National Corpus) but it turns out that they are only slightly more memorable than these words, and substantially shorter. These are drawn from the most common 15898 words, adding perhaps 15% to the difficulty of memorization (log 15898/log 4096); they average 4.25 letters, while the most common 4096 average 6.57 letters, adding 55% to the time needed to type them. This tradeoff seems reasonable.

the of and to a in it is was that i for on you he be with by at have
are this not but had his they from as she which or we an there her
were do been all their has would will what if one can so no who said
more about them some could him into its then up two time my out like
did only me your now other may just these new also any know first see
well very than how get most over back way our much think years go er
many being those yeah work got down three make us good such year still
must last take own even after too right here come both does made oh say
going erm might same under day yes man use world when want life while
again never put old need used home mr why each part house off went end
look came four give local great small place mean next case find group
quite long five party every women says later given took point men set
away often seen money fact night found since less done thing area taken
help hand best mm state water head where large yet young side days john
ca left week form face power until room tell able six high told half
times eyes doing court major war car keep once asked road open am saw
today full knew feel let ever name mind far door law voice above body
early big book known using words child main clear began show means upon
areas woman gave act round whole among real job staff black view line
city white felt kind south age start idea study sense level run read
sort third seems care try else free order thus pay past ten shall death
table love north mrs whose ago range play leave land gone ask word turn
trade few air move food team west hours god hands b sir rate cost lot
held data role cases class town bank value needs union call true price
seven paper uk eight type wife seem close heard live near sure east based
terms hard wo c stage club cos makes hope comes issue soon bed girl na

david rest tax weeks bring poor top shown music month game ways talk art
royal cut goes offer april field june news works short lost hair basis
below force lord bad stop feet meet hear king heart board fire story
nine light wrong human per along final deal boy total nice de press legal
books bit whom son march lead plan sea red hold late size space died st
nor gon peter low list lower worth term buy thank date cause okay share
model miss stay prime july sound fine dead wall test happy takes added
parts loss visit floor rates allow army sorry paul stood hour easy basic
dr tried costs wish ideas arms risk mark sat unit fish write hotel met
aware park paid style miles kept ones cup maybe s gives sales page event
shop hall rose looks bill oil claim lines james blood forms goods fell
film rules ah sent carry stand v led wrote plans york ready glass site
front wide lack cover jobs lay fall moved girls title eye lady trust n
speak p river lives mouth piece walk wales heavy task arm win green d
note baby rule post older fully radio peace rise hot types sun ran wants
sale break box sit watch civil ii tea fifty built m won spent extra none
firm knows blue trees shows sex learn cash wait add match agree aid drive
duty dog dark truth boys send born step sign media avoid apply key deep
smith plus huge mum brown chair e reach stone male plant gets horse bar
base award earth phone fear text cells edge race chief eat spend cars
begin mhm firms scale image cold t join speed wind names foot views
ball stock gas pain sell drawn worse train hit mary units save smile
meant skin build spoke warm drink banks wood dear rich pass dad sleep
fresh scene steps r band x draw crime items forty dry shot enjoy hill
sight users tv ec legs stuff fit lose daily trial vote queen tree rock
pound check inc rain equal mine fund sides henry farm fight ahead joint
wine usual rural fair twice path judge funds touch tend onto games ring
seat walls pick soft safe shape o paris aim cross ought homes pair track
grant due doubt sold user japan raise goal birds video notes lived clean
sites prove g empty card copy grand heat occur beat rooms cell quiet
neck urban grow h tiny die exist seek route least upper tour boat jack
arts leg serve ian hell bus focus worry enter faith facts shook lunch
heads alan irish fast thin crown broad star bear aye entry birth busy
corp broke vital gold italy coal alone ltd drew tom etc lips shops usa
owner tony sky grey wider slow leeds bag debt wear mass iii waste catch
talks acid hoped milk laws worst gain doors hence ibm net spot f guide
la teeth tests flat india un goals vast file brief drop suit kinds wild
link rare via phase kill fixed ship harry signs row jones metal brain
lie liked minor ta coast uses ooh grew tall youth quick lots iron hole
drugs noted sport desk noise limit lying brian inner funny chris odd bob
spain pool seats tape motor dogs co dress pages crowd anne lies china
steve sixty agent calls sum joe badly lane jim tone mike flow jesus
shock nose pull simon dream meal alive van begun yours bbc jane angry
code sheet unix fuel block aged tears kids grass roof store armed faces
fruit towns lucky reply sets drug glad sharp index l th taste ideal guy
soil shut cope song frank till argue ai lift lake ref fill teams roads y
stars cards w cat trip stick hello ice error rail loan theme pub refer
chest bird grown map keen item port count clubs cast marks loved lewis
j bread links eggs drove wages score aims tasks panel lee yards tells
roman diet chain shoes runs ha aside bound plate meat admit hopes fifth
weak tired pale treat adam gate sees luke uncle smell hurt self bath
apart welsh ages rapid beach laugh eh gun sweet sons push fault sarah
lords dance laid plane sad rough sugar golf bid coat scope enemy smoke
tim wave pure nigel bits rome snow drama ward films fail gap tory beer
frame mood camp du acts solid fly thick input shift throw feels sake

mile wet mill moral truly scott k bush tools iraq wage aids ken mad ben mp duke ends hat sam skill faced nick hills bay fewer abuse proud threw pilot yard core neil hang aunt crew bell prize asia sheep steel ruth rent bills holy paint sand anger ears soul wore depth vary nurse blame pop cry finds pack blow sorts tough jean ease guess arise pace turns fees boots rugby puts awful luck spare cuts ear novel ok load songs ride poll ratio boss clock chose plays keith delay fee split tank wing silly mixed tower minds kong false iv cheap mail guard andy sick cake cycle rocks cried hong joy cook roles wan host likes print dirty fun feed mode le dozen newly rely pitch lucy loans helen ban hate taxes fears rang votes teach knife coach harm dealt marry bet proof dave pink unity mere dust pride deaf peak waves draft gift named buyer ships rid kent shirt fans wheel blind roll wake aha ill u owned anna roger holes knees salt mps cool keeps sixth terry flesh bond hers trend dna egg ended tail saved layer gross boxes raw colin outer nhs bomb topic tends laura holds angle san bone kiss jimmy adopt squad mummy disk tie meals sue alice fat helps guest bare vat roots lists poem moves wings texts flew hide opera kevin leads upset cream hunt lloyd exact logic essex acute jury valid zero lock moon bands apple deny bones bars re fleet files mix males tip owen poet dates adds billy egypt earl fate dutch tool steam hero keys wise stuck wash gene risks storm des sole habit era mayor sing spell root mess gates joke pipe greek dare woods dawn grace fred clare kelly plain bowl folk pupil louis maria pen album curve pairs diana beds adult jump ruled kick tied zone foods latin hugh genes trace silk cm susan tight loose naked knee daddy canal cloud posts gaze lease bulk navy crash rush pot diary climb tin ad urged guns wire iran magic slip actor ray asks boats panic emily bible worn phil sheer bonds robin piano lands burst alarm ocean baker ann marx cloth wives aimed solve sadly pope eric mud ross moore audit ford shed eliot shame lad alex kings barry gear raf taxi pat mouse flats shell villa rank inch derek earn calm wars yield mills smart gay evans minus fails gary dying plot eg tale rival dull asian jews label farms marie grade scots abbey acted grain cap alter pity penny cheek allen movie nasty fancy nt belt reign pile crazy roy wool grip shots fool rear joan chip falls brick bags bike fox asset bye shore jenny risen ira hey chips ate ya tide boost sizes rows edges tries voted neat forth pan ours halt knock don les breed arab ussr debts rape giant opens lover damn shit da ye hated craft bench bruce wound brush fence kid ms liz tap damp hung brave rice swing eaten plc tube sink mount fed glory fig liver pond chaos hiv clerk stake dec disc hurry korea loud chart ed guilt poems karen pause sums q menu dish clay essay acres chin cats arose ranks strip shoot debut oral widow rope evil prey coup super crack slept ph faint chap noble harsh lamb bull pit manor iraqi fired fibre odds winds balls roses seeks beef kit lads kate vague dose ma mate grasp naval deals bore echo patch steep pray di cable tune dick rob swiss blank julia imply ie virus burn trap lit races gifts sally buses pc mild quid rigid boot bases yer drank trick lily tray myth gang shake forum nato gains seed eager grave fan walks emma davis mines deck dean cab waist relax jan sword hint prior awake crop eve dried raid suite meets fraud woke boom verse bias km julie inn slide loads honey cruel nest fatal max hire flag gods duck lip ham ugly grief isle sandy rally loyal elder ali al coins fix paths soup pole oven betty bent lorry bonus wells hook seeds sail organ cure quote brand flora blew vi beans sin pence brass solar flood yep waved slope shade slid oak pubs nerve pm theft ace bow joyce mick straw bold jet devon cliff backs shout br shelf grows skirt los leaf freud loch juice dated bloke jeans devil tales ici craig urge el slim owed angel pete kim sara

owe crude cow loves lap barn drunk poles lisa lean soap maps seal santa swim charm ozone elite react fuck alert pint ghost blown modes bowel torn bend ridge sits saudi bunch tapes saint safer jail lend fame geoff yo robyn lakes rod hull ralph coin fires corps suits spots corn swept serum crops bite ties swift stops log stamp grin bile wins cups toxic tommy sang truck tons shy fond dot lodge fetch jazz basin burns eddie dual bacon tent pie gray fry rage cd width heath ft pays tel twins ted rats z tenth stir guys grid punch lamp drag array plea fa drift lobby eec li chat rode mask sigh fury derby parks mist pig trail eagle bp sank rolls rings sauce pin nails lemon ports medal pour cared borne venue ash ferry intel alike salad shoe palm tiles tanks ml clue sweat hired lump photo flown elbow amid tokyo blues pains fluid lid bombs toys weird pools wee thumb lawn stare cabin grim wiped tense hp dies bears warn polls stem wines matt blake pump fled oz gaps cited jason chase tutor heels clash von tidy bt wrist reid gulf fort bored flour bass crews ho skull draws hardy clive arrow allan twist dock tear yacht hut beam vivid exit mice cave bride lion swung hi noisy rocky graph sofa mercy bang clark bells bc linda lea kenya flash acids texas realm pose probe axis sack dig bobby lanes aloud sunny camps holly lined jokes ulcer usage maths owl blade reads toast folly burnt rude denis slice donna wheat cakes ellis blast norm jaw sri sean merit heir dan adams scent duly janet gould shaw fist milan gazed ninth knit adapt sa exile cows owns bat lace faded drum linen laser gp drops marsh dos cape sweep par codes dirt hits rat pigs gdp dug poets non irony dairy annie hedge rebel moor cease stole rider lung glow chalk tips ryan limbs beard deer queue flame cage fog dense carol paula doyle lions gall fined agony maker fits handy toes pots polly lets hood ron arena mains drain bitch pipes lone pa pulse bury flies agnes frost choir steal maid ivory pr leo ego fever metre tops itv verb vice shaft spine ruler slave bail zones canon seas atoms altar giles bleak grab towel rises fur decay tones meg lungs leap horn pact fare stiff drill torch locks rex nuts hay fuss dc ankle belly jo clues reed wit halls socks hats quest carl loop papal loses flows cart tours telly quit jacob glen danny pet swan isles gloom ex fold jeff stays spray pine ros exam wary trips pp lazy greg cargo paddy hm dame ample vicar hips gut quota fairy petty mould arch spin posed boris nina kin plug dim souls prone molly cult alien urine coats swear norms yarn sore madam levy wayne tyres crohn peaks del beast veins gases ac onset oils ally brow scrap creed monks eased beg mason jam cries colon wh batch maxim ux prose herbs tubes peru mini gauge float naive jill arc alpha wendy whip solo feast riot disco thou ruin bees blend peat rosie ord dusty dared jamie risc vii dumb baron shiny discs valve iris surge lanka boil bin siege nazi cuba stool ellen spr cork stall obey tribe nil heap vic bolt vein tore pits sells fork fax wipe wards noon mg audio piper lb heel peers ski serbs ram libel oddly penal libya rifle poole gill stark dolly cheer mug civic twin foul hazel spoon thigh thief slot cafe weary tens prix soils hip baths flock wears toll tyne sunk trunk lydia disks czech nod gown mates vocal pro grove para hid depot lang ba vale khan idle icl sh spurs filed comic utter sexes lp spy cord ruins roofs cloak leas wolf tate docks min daft cough gmt bowed axe riots dana syria knot joins amp peer trio raids doll um ps lent alec risky lens walsh int icy buys midst jolly bind webb sparc herd coun ch sour rug polar shine nancy lacks cairo pony deed rents dale snake orbit onion trent cor fines wigan mock hurd bryan alas snap rosa kite tiger leith gps exams zoo mips mac cara thy thee sexy gnp screw purse owes ink eden dash viii sends rays envy crept aston nut dump ducks flung crust stack

dee hans guil downs drums basil rub cecil tooth skies opt alton skins
leigh jake fists tha node bald sung ripe odour nylon attic muddy cosy
guild vet span ni anc peas hmm pad freed apt nets nail sworn lou doses
ribs kohl sober rails malta chef bland mars hart deeds roar peggy pearl
lime dusk radar teddy cater verge pi stove rev rack hobby spoil morse
dome dixon alley eva lenin erect nicky imf rated olive dip blaze witch
rna gavin token idiot repay jar genre toe stems horns blunt salon nanny
moses fiona si lyons celia calf piles tins donor ditch beta amino beth
watt trout sins lied flush web overt delhi cia algae tt seize noun ssr
monk mob labs ale weigh pork pills chile saves ropes pike crane hosts
cs crest bulbs se greet glue hairs cox moist ads hatch bless acre scan
jaws frail cares scum piers ivan wires marco perth duchy herr flu chuck
alain wax rory costa ants myths motif dive pulls mungo gig daisy toy
larry kenny islam irene chord peel mandy knots sha moors foam overs
ellie bolts lynn sands melt globe chill haven fuels plo lambs bowls
bids slate pier worms veto slump lab isaac curls writ wed dole tomb jug
huh lifts swore porch peg pasta kemp arse ions crap cole wrap oct hague
lasts dam rim picks mare elect deter bust beck abbot gel pal tract raced
oath cohen brook mins limb finer nuns medau ftse curb buzz molla jerry
exert bmw kgb glare bean query merry merge void stoke mon logo mo mole
wreck loft cans anal tag rover ivy hints yen tudor rugs haul blows sly
nov moss goat tyre lucas woven tweed penis karl icing bee rick mid mat
lever brake amy spark gerry focal rio faye bark arabs turf otto jelly
hugo theda steer mann await sting tails reef brisk noel kills erika caps
atom rung quinn trim spa frown frog bred awe qc pints mama litre lamps
curry tug ee plump cites opted silas newer gin bloom wagon swap fen cdna
pam fiver dots dire beads wills undue stray packs hare cock click whale
ashes swell palms kg chess ronni fiery doug tsar punk couch ponds pets
rot junk flux woken tees tee apron flint wa shah cl bert weir weeds raped
pinch nora marc tick leak kylie cathy aided tina moira knelt gangs carer
bonn heal den seals po pest meter piled baked amiss turks ton scarf watts
puppy beams weber renal der delta curse creep wow worm renew prop plots
fake ricky niece vs verbs urgh shrug hurts warns hears cocoa avon seb
icon mam leapt ti sudan shaky dover taxed luce bates smash pears locke
froze dread aisle ter benny alps satin pill len bait vowed jenna cf bach
pens ore juan beers slab poses mast edit diane unite scare flee edith
boxer batty nan wwf stern nec guts cane assay chop aa tbsp slam aea
stout singh pants maze kay con gcse venus val nme mint folds polo lava
pies menus gina cot bloc ion frogs fills dwarf bulb vines rig pcs jets
foil doo curly tidal spur howe luton cruz crush rotor hen papa oi nun
nazis hq wry windy en crisp cigar ant liam ethos ahmed rusty kirk cuban
ag maud hindu fade caves kicks blah bella piss bikes noses hens cubic
benn wade reg clan robe percy loo goats gale coral burke sails rag niche
nests stain reins lays prof veil slap olds det beech badge tummy ryder
reds fr fares dhss skip flags diy tack sas pins wea swam felix ci thorn
spit pizza zinc troop patio lager jaq witty sod grips eyed strap kpmg hop
gala dunn boast blond quiz lest id brink booth akin vase nave jew typed
fours ncr mound jeep goose fumes cling ruc posh carlo beats tuc tram thai
mums mar hairy gypsy necks hilda fife est selby oscar insp gower drily
brows tread slabs buck nizan nixon epic aura amend tesco stale remit mph
moods hyde berry spill robot byrne bunny salts gis eats spun sid groom
cue lowe huy toss tents sic bake weed vowel stab sql owls mario foxes
butt xi wha sued fe clara wits tracy lush grill edwin diets unto slots
rift geese boro bats widen visa spd gibbs unfit logs forge et cache tan
saga nell folks es carey tenor oxide nodes bout clip flank dizzy truce

tier ss lo gran faldo cc sub rides notts judy dales belts scar rites
cyril coke trays elves curl tunes steak rue roast suck gasp razor lust
wlr rum jon dane bump birch wilko rouge micro cod kirov dunes svqs shh
tessa loaf gosh wedge stony sperm ramp mas foyer bum wight malt dye dial
cone chaps bucks tonic tides sec gum cop tasty spat nobel locus gloss
glenn cam tort tae masks hates hari flap feat womb taut miami rib boyd
awoke wi todd nepal greed gen cnaa stair pumps pious lili heirs haste
scrub candy tile teens jed fonts clung bud abu vans rita opec miner
edgar cured crawl zen vine rip coil trams gym bows wiser vain tow quay
burma glove fuse diego crag ache tuck em becky rees lure fried carla
plead jewel groin duo gorge desks carp soho riven pleas moody messy liar
katie josh ghana ethel pcr cheat tuna tramp dice angus os fetal claws
seam hooks avail peach ayr axes taps tame leant chant vms herds gaunt
gamma feeds comb tempo proxy sip belle rye monte loser dolls cfcs bulls
tara crewe swans sdlp pros haunt elton stan squat ne las lais firth camel
wryly tying memo jose hms hefty heady hasty hal drown col wrath rods haze
mend crypt arid wacc spies fools fauna cose boar viola topaz perry mod
cues bulky aroma ounce lick ledge gleam cops canoe tsp rests inns havoc
dti soda shaun pests paste bra zeal smack orcs dora oxfam nile lawns
kb slips ro newco motto euro bogus au vault jay spate nicer kits kev
hangs genus nomes maize lotus forte wyatt mused lynch herb sleek scrum
floyd elvis edged toby soak paved mesh hum doris aback subs smelt lain
waits moles lapse doe doc brew amen mao lumps flick decor cube ono il
dived bs scalp rains kerry ruby ldp dwell cider bony tonne rufus liner
hymns hug guise gigs clown rash seams gm zip yu sings earls blur undo
sown scars infer ibid frome edie dcs broom wig traps shove scorn rosy
regal bream timed fins deane bully payne hawk frau flute fbi dodgy defy
barge yemen suez slit sacks pores bliss vera spike rave leon jade hush
hind franz aug weep sgt rags mel jules gwen gilt dove munro envoy cegb
brace vodka stud hon hayes fiji eerie cds ana xv vol ups rica pagan oed
lear flask woo lofty icons cooke clwyd taxis sae plato incur huts shave
ow inset ills gemma fabia cute bnfl bays wasps resin loops kyle deity
csce woody noah moan figs dent sane misty lt friar evoke trait timid
tech alloy yeast vent tacit sd gatt flair creek tempt rsc franc apex zoe
vinyl robes rap ploy hymn grit fin fairs cp wept inert buds trot spade
pals darts bug ass tilt nash milky cite chew casts oats nay muck modem
mc laden bog gulls emi dell chunk arson tub suede satan rust jars hype
groan fri weave tally micky mi jumps dimly bluff urges slum rogue cis
pegs frs cr skye seoul plum pans kurds wares taped putt pops lima holt
gully fc abide rite oval dough coded tncs stew smear sited peck macho
grind fo dss clone sway paces lass hem rune relay oaks dummy doom dept
shire sheds scant phd pe oily knob blitz smug pouch farce demon danes
cooks voter tunic spear scot josie flaws famed cents pedal lids lag hath
gaol foe finch dup crow blair sniff scoop muted joey clyde aide vicky
sybil shady poked peril eds beak rein numb mead ix cprw vile spice sized
levi idly fella fats dj cid stead sauna lj carr bunk blot amber null
nude niall heed harp fungi ethic dues crook willy sped reap monde crab
cages rake laing havel gravy gaily fatty ebb boyle whigs queer gooch eqn
brett tiled scout rab pads limp dart bruno metro inter font daly barns
swamp nouns maine hose feb agm sage rump ra plank lame joys flare chick
bsl bingo tuned spelt pnp mince lipid filth dine baggy troy tombs knox
cords bgs parry joked gail dun blush te snack shrub shone oasis nasal mop
evils viral uh synod sow riley rhyme moths ilp grate finn claw airy nam
klerk duvet casey atlas rspb nip luis gears edna bun aunty whore twigs
snp slack rinse platt mrna ge dogma chi bede vest tamil sneak qb perch

lunar ju hides henri coals clamp stump stein pram oecd klaus ds snail seth klein jock jehan hub gogh glc gland

Topics

Time domain lightning triggering

Kragen Javier Sitaker, 2013-05-17 (4 minutes)

Time-domain radio, aka ultrawideband or pulse radio, is mostly considered in the context of very-low-power communication. But there's an interesting very-high-power physical phenomenon that could be useful for time-domain radio: lightning. The mechanism that triggers lightning is not well understood; when it happens, the electric field is still two orders of magnitude below the voltage needed to ionize homogeneous air. Then, the current jumps from effectively zero to some tens of kA within a few microseconds, with a conductor length of typically a few kilometers. This transmits a powerful electromagnetic impulse horizontally across the bands up to a few hundred kHz, an impulse powerful enough to be detected at considerable distance. If we figure on an antenna impedance of 63 ohms, 30 kA will generate a radio pulse of about 60 gigawatts.

How far away can you detect a 60-gigawatt pulse from? If we assume a detection threshold of -70dBm, which works fine in Wi-Fi cards, and a ten-square-meter dish antenna, you can detect a signal down to 10^{-11} W/m², so you hit that threshold when the power is spread out over 6×10^{21} m², which is when the pulse has expanded into a sphere 22 billion meters in radius. This is about a sixth of the distance from the Earth to the Sun. If you can detect down to -85dBm, you can apparently detect the lightning strike from just about anywhere in the inner solar system.

So it seems safe to speculate that you should be able to detect a lightning strike using shortwave radio from anywhere on Earth, and using other radio bands, perhaps from some of the other inner planets too. It really seems like overkill for terrestrial communication.

You still can't make lightning from scratch, but you can trigger it. The traditional approach is to blaspheme, but that has been shown both experimentally and theoretically to have very low efficacy; additionally, the large amount of time required to blaspheme (hundreds of milliseconds or more) would dramatically limit your possible communications bandwidth. More modern approaches, shown to work fairly reliably, include firing rockets into clouds trailing wire from spools, and ionizing paths in air using ultraviolet lasers. Presumably particle accelerators can also ionize paths in air. (I've seen speculation that cosmic-ray strikes play a role in creating lightning leaders.)

A problem with laser-induced ionization is that ionized air is opaque, so it tends to be self-limiting; nevertheless, some experiments have found success using this method.

Particle accelerators might work better; particle beams are attenuated by air, but no more by plasma than by regular air, less so even. And you could maybe use the particle accelerator to produce a beam of synchrotron radiation or X-rays to form the ionized path, rather than firing subluminal particle beams into the air. Alternatively you could perhaps use an X-ray laser.

You probably can't find a place where you can trigger lightning more often than a few times a month, and you can probably only control the triggering to a precision of a microsecond or so within a

window of a few seconds, so your total system bandwidth isn't going to be more than a few dozen bits per month.

This approach to radio communication has the advantage that you could transmit data over interplanetary distances without being detectable as a radio source without access to the spreading code.

Topics

- Physics (p. 3632) (119 notes)
- Communication (p. 3382) (19 notes)
- Time domain (p. 3749) (2 notes)
- Lightning

Some extensions of William Beaty's scratch holograms

Kragen Javier Sitaker, 2019-07-11 (9 minutes)

William Beaty's scratch holograms can be extended using dithering and noncircular scratch paths to support arbitrary movement and brightness change, with artistic and archival applications.

Gradients and disorganized abrasives

A piecewise-linear interpolation of scratch direction and density over a surface will give rise to nonlinear and, in general, noncircular scratch paths; this may be feasible to fabricate using disorganized abrasive embedded in a soft matrix, such as a piece of felt or rubber, which is moved in several passes over the surface to scratch. Due to the elasticity of the matrix, displacements of the matrix parallel to the surface vary the amount of pressure transmitted through the abrasive to the surface, and this varies the scratch density (number of scratches and average scratch width). Spatially varying this perpendicular displacement throughout the matrix, which is to say holding it slightly nonparallel to the surface, will result in a spatial scratch density gradient on the surface; temporally varying the displacement will also result in a spatial scratch gradient, but along the direction of movement. Both techniques can be used in concert to maximize scratch density gradient sharpness by moving the matrix parallel to the matrix. Nonparallel movement will blur the gradient, which may be desirable to prevent the faithful reproduction of mechanical errors such as unwanted vibration. Moving the matrix along an in-general-nonlinear path without rotating it will result in the scratches from a given point in time all being parallel, like the parking-lot floating polishing glove image mentioned in Beaty's original notes, but rotating it around an axis more or less parallel to the surface will cause the scratch paths from one side of the matrix to the other to vary in both direction and radius of curvature.

By combining these gradient effects, it should be possible to rapidly fabricate scratch holograms that reproduce an arbitrary continuously-varying set of monochrome images to fairly high precision when viewed or illuminated from different angles; this goes beyond merely fabricating a hologram of a single physical object viewed from different angles and includes, for example, arbitrary animations, though animations with only a single (possibly cyclic) temporal dimension.

If the matrix is not soft, being for example steel or pitch, the perpendicular displacement directly controls scratch depth and density, rather than indirectly through pressure. This requires more precise positional control.

Alternative scratch fabrication tools

Randomly positioned abrasive is not the only thing that could be used this way. A single-point cutting or forming tool could also be moved over the workpiece in a controllably curved path by an analogous mechanism: mounted on a wheel whose rotation angle is

precisely controlled as a gantry or other two-degree-of-freedom mechanism moves the wheel's center over the work. (A hardened-steel glass-cutting wheel might make an adequate tool for forming lines in the surface of a softer metal, or cutting them in glass.) A series of regularly spaced points would also work; for example, you could use the edge of a saw blade held nearly parallel to the workpiece surface, controlling the angle and curvature of the blade as you drag it over the surface to scratch it, using a six- or seven-degree-of-freedom control mechanism. And you might be able to use a rotary brush of aluminum-oxide-particle-impregnated nylon — either rotating around an axis parallel to the surface to make straight scratches at a given angle, or even rotating non-parallel to the surface to make elliptical scratch arcs.

Media

More highly reflective surfaces, such as metals, are of course more desirable. Grinding copper flat, then electropolishing it, then engraving the hologram with abrasive, then plating it in silver, chrome, or nickel, should enable very high contrast ratios. Aluminum might be easier to form. Glass is in some ways the optimal material, having no grain size or work-hardening to worry about, but getting it to be highly reflective requires some kind of silvering process, nowadays typically by vacuum sputtering. If this is not done, objects on the other side of the glass may be clearly visible, and for some applications — such as superimposing a sort of hardcopy alternate reality view on an object, for example for measurement purposes — this could be desirable.

Metallic media can perhaps be mass-duplicated by electrotyping, but molding, as is done for vinyl phonorecords or diffraction gratings from the Grating Lab, is probably a better process. The molded reproductions can then be silvered through sputtering, as gratings are.

Color

By adding color filters over the surface, ideally after scratching, some color should be possible, although it means that scratch density needs to vary in phase with the color-filter variation to get color, which requires much higher spatial frequencies than would otherwise be needed, thus substantially increasing fabrication time. The R-G-B-G checkerboard used in modern digital cameras might be ideal in some sense, but using parallel strips (either R-G-B-G, R-G-B, or R-G-B-W as in some modern LCDs) would reduce the scratching problem. If the surface is titanium, chrome, or stainless steel, iridescent oxide layers may be an adequate way to apply color filters.

Oil films

At the extreme of soft materials, it should be possible to produce most of these effects in a film of oil or grease on a smooth surface, such as glass or PMMA, without using abrasives or cutting tools at all; a soft rubber squeegee is adequate. Such a temporary hologram might serve as a frame of an animated display or as a temporary hardcopy. In time the minimal surface tension of oil will erase the images.

Virtually transparent opaque polyhedra

It should be possible to apply these effects to more than one side of a polished metal polyhedron or convex curve to produce parallax-correct views of a three-dimensional object within, though with only a single dimension of parallax. The illusion will be that of seeing a luminous object within the polyhedron through the metal surface, as if it were merely a wire frame enclosing the object.

Archival

Of course the usual archival applications exist; by virtue of scattering light directionally, a scratch that is only half a micron in width and a few microns long becomes brilliantly visible from a certain angle. Under ordinary sunlight viewing conditions, a single sheet of material can encode some 300 different images at illumination angles differing by half a degree (180 degrees / ½ degree minus a safety factor), or perhaps more if the contrast-enhancement techniques in Analemma sundial (p. 1955) are applied. Art-gallery viewing conditions, with a more directional light source than the sun and a background that is much darker than the blue sky, could permit many more images to be encoded.

10-micron household aluminum foil, shiny on one side, with 300 images encoded at 300 dpi and one bit per pixel, would provide archival bit density of 42 gigabits per square meter and 4.2 petabits per cubic meter with this approach; scaled down to the size of Paul Atreides' Orange Catholic Bible, that's 4.2 gigabits per cubic centimeter. But such a delicate medium as 10-μm annealed aluminum is difficult to handle without creasing it, and the electrostatic page-selection mechanism described in *Dune* is unfortunately probably not feasible outside of fiction.

The 100-μm forged aluminum used for drink cans can withstand rough handling, but it is surely overkill for this application. 30-μm aluminum flashing is probably adequate and can be polished on both sides. This would provide 2.8 gigabits per cubic centimeter.

By this method you could produce a book, readable with the naked eye in direct sunlight, safe to handle with bare hands, using inexpensive materials (but very expensive fabrication techniques), that will last millennia under reasonable archival conditions, which at 130 mm × 80 mm × 10 mm contains 290 gigabits of data, 36 gigabytes. If we want it to use normal lettering, we probably need about 3×6 pixels per letter at least, so that would be only 16 billion letters of text, about 3 billion words. This is about a third the size of English Wikipedia.

More aggressive specifications might use 600 dpi and 1200 images rather than 300, thus requiring a light four times more directional than the sun, such as a sunbeam entering an otherwise dark room through a peephole or vertical slit. This would enable sixteen times greater information density but would probably require a magnifying glass to read.

For mass production of such holographic archival devices, perhaps a plastic film could first be molded (as described above) out of an archival-safe thermoplastic such as PET, then silvered with aluminum in a vacuum.

Topics

Would Synthgramelodia make a good base for livecoding music?

Kragen Javier Sitaker, 2015-09-03 (8 minutes)

I wrote a thing a while back called "synthgramelodia", which randomly synthesizes melodies from a grammar, many of which are listenable. Its homepage, which is currently just a list of outputs, is at http://canonical.org/~kragen/sw/synthgramelodia. I should probably at least put the software there.

I think it's probably possible and worthwhile to use this same grammar for interactive improvisational composition, but I don't feel up to actually implementing it at the moment. (I'm horizontal with what I think is a flu.) But I think I can explain the idea.

Synthgramelodia uses a tiny six-production grammar of melodies which is capable, in theory, of expressing basically any chromatic melody using the Chinese equal-temperament scale and the binary note values used in mainstream Western music today; but it's intended to be biased towards things that will sound "nice", so that the result will often be listenable, and it uses a DAG to promote repetition of motifs with some variation.

The grammar can be expressed as follows in a single line:

```
m ::= "." | [a-z] | _<m> | +<m> | (<m> <m>) | (<m> ^ <m>)
```

The two fundamental atomic melodies of the grammar are:

- a **Rest**, written `.`, which is a silence lasting one beat, and
- a **NoteScore**, written with a letter `a`, `b`, etc., to indicate which of the synthesizer's various "instruments" to play. A NoteScore by itself represents the instrument being played for a single beat at 220Hz, which is A3 or A below middle C.

On top of these, there are four compound melody types which modify one or more melodies:

- a **Transpose**, written `_x`, where `x` is the melody being transposed; it represents the same melody as `x`, except lower in pitch by a perfect fifth.
- a **Louder**, written `+x`, where `x` is the melody being amplified; it represents the same melody as `x`, except louder in volume by 3 dB;
- a **Sequence**, written `(x y)`, where `x` and `y` are two melodies being concatenated, which consists of first the notes of `x` and then the notes of `y`; if `x` and `y` are of different lengths, the shorter one is slowed down to equal the longer one in length. Also, `y` is reduced in volume by 2 dB.
- a **Parallel**, written `(x ^ y)` where `x` and `y` are two melodies being played simultaneously; `y` is raised in pitch by an octave, while the bass line `x` is left unchanged, and if they are of different lengths, the longer of the two is accelerated in order to be of the same length as the shorter.

(At some point, I replaced the stretching of the shorter melody with repetition, which I don't think improved it.)

Synthgramelodia builds up random melody DAGs using these productions or node types in a bottom-up fashion that tends to reuse the same node in many parts of the tree, with the result that you tend to hear the same motif at diferent speeds and transposed to different pitches, although not using different instruments.

It should be apparent that you can raise a melody's pitch arbitrarily high by paralleling it with a bass line consisting of a rest several times, lower it arbitrarily low by transposing it several times, reach any chromatic pitch by combining these two, amplify it arbitrarily by loudering it many times, accelerate it arbitrarily by concatenating it with many rests (or with itself) and then paralleling it with a single rest, and slow it down arbitrarily to an arbitrarily slow speed by concatenating it with a long sequence of rests, although at the cost of embedding it in a longer silence.

It should also be apparent that the only reachable pitches are in the chromatic scale, the only reachable melody lengths are powers of 2, and all integer numbers of decibels are reachable amplitudes.

(It might make more sense to reverse the speedup/slowdown semantics of Parallel and Sequence, so that you can slow things down arbitrarily by paralleling them with a long silence, while speeding them up would just require concatenating them with a single rest; or maybe make a separate node type for speeding things up.)

Given all this, it also seems clear that you should be able to write an expression (infix or Forthlike) to generate a "gramelodia", perhaps even synthesizing it in real time as you improvise. I suspect that some kind of Forth-like interface, like GlitchMachine, is probably the best way to edit this; it gives you a linear thing you can edit which is essentially immune to syntax errors.

Constructing arbitrary DAGs on a Forth stack might seem tricky. Arbitrary trees are of course straightforward: the postorder traversal of the tree forms the Forth-style command sequence, and you're done. Some DAGs can of course be constructed with nothing more than "dup", but you need some amount of stack manipulation to construct things like `(a b) (a c)`.

If we consider a Forth two-stack machine as a sort of Turing machine that can insert and delete cells on its tape, we can see that the four operations `dup`, `r>`, `>r`, and `shift` are sufficient to shuttle nodes around in order to construct arbitrary DAGs. With the operand stack on the left, the auxiliary ("return") stack on the right, and top-of-stack outlined with parens, these operations are as follows:

- `dup`: "... (a) ..." → "... a (a) ..."
- `>r`: "... b (a) ..." → "... (b) a ..."
- `r>`: "... (b) a ..." → "... b (a) ..."
- `shift`: "...(a) b ..." → "... b (a) ..."

The sequence of cells on the tape, omitting duplication, remains unchanged except by `shift`, which moves the current top-of-stack to the right on the tape, which can be repeated until it reaches the position it needs to combine with whatever it needs to combine with; `>r` and `r>` move the head left or right on the tape; and `dup` is what you do to use a node in two places. (Arguably the duplicate ought to be on the auxiliary stack to the right, not the main stack to the left.)

I've made several attempts to reduce this to three operations, but none has yet been successful. There's probably something hiding just

out of sight; but, for example, with just `r!`, `shift`, and `r>`, there's no way to write `swap`, which is a problem in the presence of non-commutative operations like the gramelodia melody combinations; and if you integrate `dup` into one of the other operations, you probably need to add `drop` as well.

You could imagine a live display not only of the melody, but also the sheet music, the DAG, sheet music for individual DAG nodes, an infix version of it, and a highlight on each of these indicating where your editing cursor is in the Forth sequence; and also keys to jump your cursor to the construction of one or the other top stack items at that point in execution, so you can navigate the DAG structurally.

Topics

He listened to the human intently

Kragen Javier Sitaker, 2014-06-29 (4 minutes)

He listened to the human intently, although his demeanor was casual.

"Yes," said the human.

From the sound of the E, he discerned that the human was probably from either near Boston or near Sydney. A human from Scotland might pronounce the E the same way, but only if he were impatient. And the human's phoneme pacing, as well as the slow pulsing of blood in his face, showed that he wasn't impatient. A bit curious, perhaps. Interested. But not impatient.

"Why did you buy it?" he asked the human, in a voice meticulously calibrated by a linear model on tens of millions of hours of wiretapped telephone conversations, whose only remarkable feature was its featurelessness.

"Well, I'd heard of the title, but I had never seen a copy in person. So when it came up for auction, I put in a low bid, and it turned out to be the only one." The human's eyes flicked down a couple of times, punctuating his speech.

In the fillip the human gave to the word "auction", he glimpsed childhood memories of weekends listening to the gavel, clearly from the contraband auctions the Coast Guard held at Back Bay every second Friday, and his secret joy in acquisition, along with faint hints of an adolescence spent masturbating in surplus military fatigues. In the accent he heard the traces of the human's parents' accent — Portagee, as they'd call it themselves. And in the tiny hesitations he heard the wounds of a childhood at the mercy of a sadistic parent, the human's father, probably. The timeframe of the Portuguese accent influence would be the 1950s or 1960s, and the human had grown up in poverty, with guilt attached to his love of books, so probably his parents were not highly educated.

Ah, it came into focus: the father had served briefly in PIDE, Salazar's secret police, and then fled Portugal to escape the horrors of what he had done in Tarrafal; but the monster awoken in the man fed on his son's soul even today. Every word the human spoke gave subtle evidence of it, to those who knew how to listen to human speech.

He knew how to listen.

The human was lying when he said he'd never seen a copy in person, he could tell, but the lie was motivated only by a desire to keep the conversation interesting. He hadn't been interested in the other copy he'd seen, but for a boring reason, and he knew that boring clients was no way to sell rare books.

Where the light glinted off the cover of the book, he recognized the fingerprints of half a dozen long-dead rare book dealers, but although he noted their identities for eternity, the book and its history did not really interest him. He was hunting down a conspiracy of rogue archivists who frequented illicit book dealers like this one, and who particularly might be interested in nineteenth-century chemistry books. But their fingerprints were not among the prints on the cover, and the human's speech and motions gave no suggestion they had

been here.

No matter. Soon he would find them. They had no chance. Today he would set a bait here, tomorrow elsewhere; sooner or later they would buy a book from someone he knew, or carelessly leave a fingerprint on the subway, or talk to one of their family members, who were all under surveillance by casual acquaintances who seemed human.

"I have the next one in the series," he said to the human in his voice, a voice so average that to another like him it would stand out like a siren. "Would you be interested?"

In the human's face he saw a lifetime of hopes, dreams, fears, obsessions, and disappointments flicker by in a fraction of a second, and then heard them richly modulated onto the human's casual voice: "Could be."

Topics

- Psychology (p. 3669) (18 notes)
- Fiction (p. 3454) (7 notes)

How cheap can laser-cut boxes be?

Kragen Javier Sitaker, 2017-06-01 (2 minutes)

I bought a little plastic box with eight partitions for AR$49 at Sodimac (US$3.10). This seems expensive and I thought maybe I could get an equivalent laser-cut for less. Mina was skeptical.

The dimensions of the box are about 130 mm × 110 mm × 25 mm.

There is one partition running in the 130 mm dimension and four partitions in the 110 mm dimensions, and there is a lid.

This works out to six partition edges and four face edges in the 130 mm dimension, thus 1300 mm, plus ten partition edges and four face edges in the 110 mm dimension, thus 1540 mm, plus 15 vertical edges of 25 mm, thus 375 mm; a total of 3215 mm.

My current laser-cutting cost model is that laser-cutting goes 24 mm per second and takes an extra 60 ms per vertex, and the cost is US$0.026 per second. Probably there would be about four vertices per vertical edge, for a total of about 60 vertices, a total of only 3600 ms, not significant in the overall cutting time. The 3215 mm would take 134 seconds, plus another 4 seconds of vertices, a cost of US$3.60. So it would work out to be pretty much the same cost, probably, or slightly more.

For larger boxes, I think the laser-cut MDF approach is definitely cheaper. Or if I can find someone who does cheaper MDF cutting.

Topics

- Pricing (p. 3646) (89 notes)
- Manufacturing (p. 3558) (50 notes)
- Laser cutters (p. 3540) (10 notes)

Improving lossless image compression with basic machine learning algorithms

Kragen Javier Sitaker, 2016-07-27 (2 minutes)

Image compression algorithms work, in some sense, by finding a good way to predict the value of each pixel from the already-known pixels, and then correcting the prediction more or less. (It's kind of a stretch to apply this description to JPEG, I guess...)

One possible predictor is a simple spline fit to the previous pixels. If it's zero-order, this reduces in some sense to RLE; first-order predicts gradients; second-order and higher may not be useful.

A more interesting predictor for screenshots is perhaps a KNN predictor: given the so-far decoded pixel data, what are the K previously-decoded pixels whose environment was most similar? Perhaps, for concreteness, we take the two pixels above, two pixels to the left, and one pixel diagonally up and to the left. Let's take them in grayscale so we only have a 5-dimensional parameter space, since RGB TrueColor would form a 15-dimensional parameter space.

Now we can search the so-far-decoded image for the K pixels whose environments are most similar to our current environment, using e.g. Manhattan distance, and take the mean or, more likely, median of their color to form our predictor of the current pixel's color.

5 dimensions is small enough that we could reasonably build a k-d tree to keep the search efficient.

My thought is that the vast majority of pixels in screenshot environments would have exactly the predicted color, because those 5 pixels have enough information to nearly uniquely identify the font glyph, pixel position within that glyph, and background color that we're looking at. That means you can encode the average residual in less than a bit, particularly if you accept some quantization noise.

(We might want to do some kind of dimensionality reduction, e.g. PCA, to be able to use color and more than 5 pixels.)

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Graphics (p. 3483) (91 notes)
- Compression (p. 3384) (28 notes)
- Information theory (p. 3524) (9 notes)
- Artificial intelligence (p. 3307) (8 notes)

Twingler

Kragen Javier Sitaker, 2014-02-24 (7 minutes)

In Twingler, a generic data structure munger, Mark-Jason Dominus proposes a new broadly-useful DSL.

The problem

Given

```
[
    [Chi, Ill],
    [NY, NY],
    [Alb, NY],
    [Spr, Ill],
    [Tr, NJ],
    [Ev, Ill],
]
```

he wants to easily write the transformation to DS1:

```
{ Ill => [Chi, Ev, Spr],
  NY  => [Alb, NY],
  NJ  => [Tr],
}
```

for example, or to any of DS2:

```
{
  Chi => Ill,
  NY => NY,
  Alb => NY,
  Spr => Ill,
  Tr => NJ,
  Ev => Ill,
}
```

DS3:

```
{ Ill => [Chi, Spr, Ev],
  NY  => [NY, Alb],
  NJ  => Tr,
}
```

(unnamed):

```
{ Ill => 3,
  NY  => 2,
  NJ  => 1,
}
```

or DS4:

```
[ Chi, Ill, NY, NY, Alb, NY, Spr, Ill, Tr, NJ, Ev, Ill]
```

As a current example of how it is currently unnecessarily difficult to write the first transformation, he gives the Perl

```
my $out;
for my $pair (@$in) {
    push @{$out->{$pair->[0]}}, $pair->[1];
}
for my $k (keys %$out) {
    @{$out->{$k}} = sort @{$out->{$k}};
}
```

and suggests that even the abbreviated syntax

```
for pair (in.items) :
  out[pair[0]].append(pair[1])
for list (out.values) :
  list.sort
```

is unnecessarily complicated.

MJD's Twingler templates

he ends up suggesting

```
DS1: [ <[X,Y]> ]
DS2: { <X=>Y>  }
DS3: { <X => [<Y>]> }
DS4: [ <X, Y> ]
```

where ◇ is "ITERATE over the thing inside and make a list of the results," like `map` in Perl or `flatMap` in Scala, and `X` and `Y` are the first two dimensions of a sort of adjacency matrix computed in an unspecified way from the original data:

```
       Ill       NY       NJ
Chi    X
NY             X
Alb            X
Spr    X
Tr                    X
Ev     X
```

...or two columns, I guess, considering the original data as an N-ary relation represented simply as a list-of-lists. (You could probably use the same or similar notation to specify how to reduce the original data to an N-ary relation.)

I don't understand how his four expressions above give the four data structures explained previously; it seems like the DS3 expression `{ <X => [<Y>]> }` should give DS1 rather than DS3, while the DS1 expression should give the original input. I'm going to assume this is just an error MJD made.

MJD points out that you can easily incorporate arbitrary functions into the template:

```
{ <X => max(<Y>) }
```

Later he gives the example, given the table

```
|----+------+--------+---------+------|
| ID | NAME | SHADE  | PALETTE | DESC |
|----+------+--------+---------+------|
| A  | AAA  | red    | pink    | Aaa  |
| B  | BBB  | yellow | tawny   | Bbb  |
| A  | AAA  | green  | nude    | Aaa  |
| B  | BBB  | blue   | violet  | Bbb  |
| C  | CCC  | black  | nude    | Ccc  |
|----+------+--------+---------+------|
```

how to compute

```
{ A => [ AAA, [ [red, pink], [green, nude] ], Aaa],
  B => [ BBB, [ [yellow, tawny], [blue, violet] ], Bbb],
  C => [ CCC, [ [black, nude] ], CCC]
}
```

using the expression

```
{ < ID => [
            name,
            [ <[shade, palette]> ]
            desc
          ]>
}
```

His original examples are fairly easily handled with mapreduce, which hadn't been identified yet when he wrote his notes, and although this one is too, it points the way to examples that are not so easily handled. For example, the classic Titanic dataset, included with R and containing information on survival by gender, age, and class, could be reasonably munged into any of

```
{<Class => {<Sex => {<Age => {<Survived => N>}>}>}>}
{<Sex => {<Class => {<Age => {<Survived => N>}>}>}>}
{<Sex => {<Age => {<Class => {<Survived => N>}>}>}>}
{<Sex => {<Age => {<Survived => {<Class => N>}>}>}>}
```

as well as many other forms. This is not a task you can do with mapreduce.

(I'm not referring to the parallel and fault-tolerant attributes of mapreduce as implemented by Google or Hadoop, but to the higher-order mapreduce function that I've argued should be in Python itertools.)

QBE

Later in his post, MJD suggests that it might make more sense to simply supply a sample input and sample output; e.g.

```
[ [ A, B ],
  [ C, B ],
  [ D, E ] ]
-------------
```

```
{ B => [A, C],
  E => [D],
}
```

I'm not sure if this kind of QBE will work, but it does seem like if it works, it would be simpler to use, despite needing more verbose input. The idea is that, instead of specifying iteration with a separate iteration operator, you specify it by iteration. Consider the slightly harder problem

```
[ [ A, B ],
  [ C, B ],
  [ D, E ] ]
--------------
[ [B, [A, C]],
  [E, [D]],
]
```

This is unambiguous, but if we delete the first output item to get

```
[ [ D, E ] ]
--------------
[ [E, [D]] ]
```

that could mean either the original

```
[ [ A, B ],
  [ C, B ],
  [ D, E ] ]
--------------
[ [B, [A, C]],
  [E, [D]],
]
```

or the alternative

```
[ [ A, B ],
  [ C, B ],
  [ D, E ] ]
--------------
[ [B, [A]],
  [B, [C]],
  [E, [D]],
]
```

although this is not possible in the hash case.

I think this might be less wieldy in the case of deep nesting, and it doesn't accommodate functions like `count` or `max` as easily. Consider the Titanic example

```
{<Sex => {<Class => {<Age => {<Survived => N>}>}>}>}
```

which could be written as

```
[ [First, Male, Adult, Yes, 24] ]
----
```

```
{ Male => { First => { Adult => { Yes => 24 } } } }
```

and is thus no problem; but if we instead want nested lists of pairs, we are in trouble:

```
[ [First, Male, Adult, Yes, 24] ]
----
[ [Male, [ [First, [ [Adult, [ [Yes, 24] ]] ]] ]] ]
```

which is hopelessly ambiguous; we need at least one repetition at each level to specify it properly:

```
[ [First, Male, Adult, Yes, 24],
  [First, Male, Adult, No, 25],
  [First, Male, Child, Yes, 26],
  [Crew, Male, Adult, Yes, 27],
  [First, Female, Adult, Yes, 28],
]
---
[ [Male, [ [First, [ [Adult, [ [Yes, 24],
                               [No,  25] ]],
                     [Child, [ [Yes, 26] ]] ]],
           [Crew,  [ [Adult, [ [Yes, 27] ]] ]] ]],
  [Female, [ [First, [ [Adult, [Yes, 28] ]] ]] ]
]
```

which is, at least to me, less readable than

```
[ <[Sex, [ <[Class, [ <[Age, [ <[Survived, N]> ]
                    ]> ]
         ]> ]
]> ]
```

although I'm not going to try to claim that that's some kind of paragon of readability either.

JSON

Twingle is a great deal more interesting today than in the late 1990s when MJD originally invented it, because now we have a great deal of data on the web (and other places!) represented as JSON. The fundamental aggregate data types in JSON are the same fundamental data types Twingle handles, and simple transformations between different representations of such datasets is a ubiquitous task.

Binate

At first, I was thinking that maybe Binate would be useful for this kind of transformation, since binary relations are awfully similar to the Perl dictionaries being represented by {} here; and Binate might well be more compact than the Twingle representation.

However, binary relations sort of erase the distinction between dicts and lists. XXX

Topics

Categorical zero sum prohibition

Kragen Javier Sitaker, 2019-05-27 (updated 2019-06-01) (23 minutes)

Let's explore why rich societies are more economically productive, and why both Kantian and utilitarian ethics forbid exploit development and SEO, as explained by *Bill & Ted's Excellent Adventure*.

A tale of three hobbies: the bad, the good, and the ugly

Suppose almost everyone in the world worked really hard to get better at football as a hobby, instead of wasting their time watching non-football TV or getting angry about politics. The level of professional football games would improve, but there would still be about the same number of professional football games; maybe a little more diversity, but just as today, most people would watch the matches of the top clubs and not the much larger number of bush-league matches. So the overall well-being of the world would increase, but only a little. Or maybe it would decrease, if more people tore their knee ligaments. Either way, it would be pretty much the same.

Suppose that, instead, almost everyone in the world worked really hard to get good at medicine as a hobby — doing paramedic training, reading case studies, predicting patient outcomes, doing Gwern-style double-blind self-experiments, performing recreational plastic surgery on their pets, that kind of thing. The level of health care would improve dramatically — deaths due to drug interactions the prescribing physician failed to notice would go way down, people's decisions about when to go to the emergency room would be much more accurate, people's lifestyle choices would become slightly healthier, and medical science would advance much more rapidly. The overall well-being of the world would increase substantially.

On the gripping hand, suppose that almost everyone in the world worked really hard to get good at fighting with kitchen knives. Perhaps the level of Olympic fencing would improve substantially, which would make for enjoyable TV — though perhaps the sports are far enough apart that it wouldn't transfer — but it's also likely that a larger fraction of fights would involve knives instead of just shouting or fists, usually killing one or more people. The training, too, would probably involve a certain number of accidental fatalities, and some fraction of those would lead to revenge killings. More children would be orphaned. The overall well-being of the world would diminish.

On this basis, I claim that it would be better for almost everyone to take up medicine as a hobby than football, and better football than knife fighting; and this is because knife-fighting is a negative-sum game, football is a zero-sum game, and medicine is a positive-sum "game".

"Party on, and be excellent to each other," Kant, and robbing old women

As I remember, sage Dave Long at some point unpacked the Bill

and Ted philosophy as follows: "party on" means not to engage in negative-sum "games" — in the very general sense of "interactions with other agents" — and "be excellent" means to engage in positive-sum games. To the extent that you can control which "games" you "play", it is better to invest your limited time in positive-sum games. (Sage Shakyamuni reportedly took the alternative position that you should not waste your time playing any games, because you are going to die soon, while the sages of Wyld Stallyns instead played games to defeat death itself.)

Of course, a person may find themself in a situation where it is more advantageous to them personally to direct their effort to a zero-sum game or a negative-sum game. Consider the thief who spotted my neighbor, an old woman, at the bus stop last month, and decided to knock her down and beat her to steal $300 from her (about US$7). He chose to direct his efforts to robbing her, and although he injured her in the process, he presumably considered that a worthwhile tradeoff, because it satisfied his desires for food or crack or whatever. A pyramid-scheme participant who recruits downstream members is doing the same thing, but in a less courageous way, and the injury is delivered via deception to the victim's mind, rather than via a stick to her knee.

From a consequentialist point of view, this is a bad outcome if we consider the thief and their victim to be equally important, so that the loss of money by one is precisely canceled by the gain of money by the other, while the putative injury or deception has no such counterbalancing benefit. If the consequentialist considers the thief's profit to be more important — perhaps because the thief is poorer and therefore benefits more from the money, or perhaps because they subscribe to a worldview where it is better for the money to go to a brave thief rather than a cowardly victim, or a smart thief rather than a stupid victim, or simply because they are racist — then the consequentialist can justify the theft as morally correct. But an *egalitarian* consequentialist cannot so justify theft or any other negative-sum game.

Neither can a deontologist who subscribes to Kant's categorical imperative, which is of course (XXX) one of Kant's first examples; the thief cannot at the same time will that the victim should steal their money back, delivering the same beating to the thief, because that would leave the thief bruised but no richer. Only by some kind of special pleading can the Kantian thief save his livelihood, and the same kinds of special pleading that the consequentialist could use to justify the theft can be used to exempt the thief from the maxim that would otherwise strip his stolen money from him. Again, this logic applies equally well to any negative-sum game.

For an egalitarian consequentialist, the imperative to "be excellent" — to play positive-sum games rather than zero-sum games — is just as strong as the imperative to "party on", that is, not play negative-sum games. Kant's theory (like most deontologist theories) also regards it as a "perfect duty", and thus obligatory, to "party on" in every possible way — at least in the specific examples Kant gives — but regard it as a supererogatory "imperfect duty" to "be excellent".

XXX this description of Kant's incoherency criterion is itself incoherent and needs reworking

Computer security, the nuclear arms race, advertising, and SEO

In theory, since computers only do what you program them to do, you could decide not to program them to accept arbitrary commands from random people anywhere on the internet, instead only programming them to accept arbitrary commands from their actual owners. That is, there is no need for software to contain security holes. Computer security violations are not inevitable. They are the result of undiscovered programming mistakes, which is why the most significant forum for announcing and fixing security holes in the 1990s was called BUGTRAQ.

In the late 1990s I spent some effort finding security holes in software and reporting them to get them fixed. Unfortunately, it became increasingly apparent that the current economic and intellectual environment was going to introduce new security holes faster than we could remove them, so rather than a gradually decreasing pool of still-undiscovered security holes, we have a gradually increasing one, with the disastrous effects on the human right to privacy documented by the Snowden revelations.

In such an environment, exploit users and defenders are in a sort of arms race, a literal race to see who can respond faster. If the defender is faster to patch a newly discovered hole in deployed systems, they win that round; if the exploit user is faster to acquire and employ an exploit for it, they win that round. So whenever one side or the other increases their commitment of resources and gets ahead a bit, the other side has the option of matching that increase to get back to the previous equilibrium.

This sounds like a zero-sum game, but in a larger context, it's negative-sum: the human effort spent on the vulnerability treadmill is taken from the time available for making music, discovering new algorithms, cooking food, meditating, painting pictures, writing poetry, reading poetry, watching movies, making love, or raising children. Every sysadmin on call who has to patch the production systems within an hour so that exploit users won't break in with the newly announced vulnerability is spending the irreplaceable minutes of their life on a Red Queen's race with the spies at the NSA or the FSB. They are falling short of partying on.

Of course, neither side can unilaterally scale back its efforts; that would amount to surrender. If it's your job to keep your employer's public-facing systems up to date so they don't get popped, you can't just decide it's not worth the effort. But you can find a new job.

There are other drawbacks as well: it's no longer a viable option to continue running outdated software unless it's in a very unusual isolated environment, like a non-networked video game, and the increasingly rapid and frequent response required to new vulnerabilities has the effect of centralizing both patching and exploit use in large organizations. This means everyone is exposed to the risks of deploying untested new software on short notice and to having strangers administer their most intimate computer systems, such as Android hand computers. Also, sometimes patching a vulnerability unavoidably introduces incompatibilities, causing bitrot.

So I stopped spending effort on that.

The nuclear arms race during the Cold War had a similar dynamic:

each side constantly worked to preserve a second-strike capability (by, among other things, building enough warheads that some would be likely to survive a first strike from the other side) and to find ways to remove the other side's second-strike capability. Neither side could opt out of the game, but the result of both sides playing it harder and harder was the decades-long threat that civilization could end at any time, with 20 minutes of warning.

Fortunately there were also people on both sides like Jonas Salk, Norman Borlaug, and Andrey Kolmogorov who were able to dedicate their lives to positive-sum games instead of the negative-sum game of the nuclear arms race.

Commercial competition is, in theory, a positive-sum game, though eventually only slightly so — as more and higher-frequency participants in a stock market means fairer prices, with retail participants having to pay much narrower spreads to the market-makers, more competition in consumer-goods markets should result in goods priced just above the lowest possible marginal cost of production. One reason this doesn't happen in practice is advertising: consumers buy goods that are advertised rather than equivalent goods that are not, and in many cases are induced to buy categories of goods they wouldn't have bought at all without advertising.

This puts advertising in the negative-sum category: to survive, firms are forced to push positive information about their products in front of customers, whether that's by traditional display ads and flyers, by getting newspapers to write articles about them, by commissioning shill research (such as rigged Gartner product comparisons), by buying product placements in supermarkets and music videos, or by doing SEO on their web pages. Moreover, they cannot afford to be much more forthcoming about the drawbacks of their products, or less enthusiastic about their benefits, than their competition is, unless consumers are turned off by their immodest hucksterism.

To the extent that consumers can find objective research about the relative merits of different products and distinguish it from the advertising, all these strategies will be ineffective, so firms must work harder and harder to disguise their hustle and puffery as objective research, making it harder and harder for customers to find objective information about their products and, ultimately, about anything at all.

And so it is that if you search for almost any commercially relevant topic on Google today, the results are the Wikipedia article and nine shill pages.

Selfish reasons for avoiding negative-sum and zero-sum games

You might reasonably think there's a prisoner's-dilemma-type global-local incentive conflict when it comes to playing negative-sum games: even if it is not in the interest of society as a whole that you beat up old women to rob them, you might perceive it as in your *own* interest, however you define that. And surely for some definitions you will sometimes be correct. But this happens less often than you might think.

First, let's consider negative-sum games like retail day-trading, in which all the participants are voluntary (as contrasted with the game

of robbing old women at the bus stop, in which my neighbor was an involuntary participant). This is a mildly negative-sum game because of broker commissions and the spreads paid to market makers, which are siphoned off of the otherwise-zero-sum transactions between retail day-traders. (The advent of penny pricing and algorithmic HFT has enormously diminished the magnitude of the spreads in the last 15 or 20 years.)

Presumably most of the participants in the game believe it is in their interest to participate, although there might be a few acknowledged addicts who just haven't managed to quit even though they know it's bad for them. But many of them are wrong, more than half in this case. That means that if you think it's in your interest to participate because you will make money, you're likely just mistaken. In this market you really are behind a Rawlsian veil of ignorance, not knowing if you are predator or prey. I've watched smarter people than myself waste fortunes on such mistakes.

But there's another, subtler problem. The people who participate in some activity, whether it's football, day trading, knife fighting, medicine, SEO, or beating and robbing old women, form an affinity group — a "community of practice", it's sometimes called. They tend to talk to each other, sharing information about their shared activity, and often they engage in other transactions with one another — cellphone thieves need cellphone fences, for example, who in some sense form part of the same community of criminal practice. So if you decide to spend time day trading, you're also implicitly deciding you're going to spend some time hanging out with day traders, talking with them, maybe buying them a beer from time to time. And similarly if you do any kind of medicine — even if you're just taking CPR training — you're going to spend some time hanging out with medical people.

So, we can reasonably ask, how might hanging out with medical people differ from hanging out with cellphone thieves? And there are a lot of answers that are specific to these fields of endeavor (for example, medical people tend to be from Cuba, while cellphone thieves tend to own motorcycles) but one difference that's common across all these fields is that the kind of people who choose to play negative-sum games are (however slightly) the kind of people who choose to play negative-sum games, valuing their individual interests over others', while the kind of people who choose to play positive-sum games are less so.

We can renormalize this and get a stronger result: if you beat up old women to rob them, then the people who choose to hang out with you knowing this will tend to be people who think that's a reasonable choice, but who aren't afraid you will beat them up and rob them — often because they think they're stronger than you, which (refer to previous lemma) they believe justifies robbing you. And similarly for SEOs, advertisers, day traders, and soldiers, with appropriate variations.

So, if you go out drinking with doctors and with day traders, you should expect the day traders to stick you with the tab more often. Maybe not much, but detectably. And if you hang out with people who beat up old women to rob them, you're likely to get robbed, one way or another.

(Of course, it's also possible that someone could take revenge on

you. But that's true in positive-sum games too — positive-sum games aren't always win-win, though zero-sum and negative-sum games are never win-win.)

This doesn't mean you're better off if you hang out with people who *advocate* playing positive-sum games and not playing negative-sum games. A noticeable fraction of them are just trying to talk you into giving them your money, or your volunteer time, or your vote, or leaving your monogamous partner alone with them, or whatever.

Social capital, factionalism, and negative-sum games

"Social capital" is often invoked to explain why rich countries experience vastly higher levels of economic productivity from workers with similar levels of education employing similar levels of capital intensity than poor coutries do; the effect is so strong that poor people from poor countries can greatly improve their income simply by working in rich countries.

After 12 years of living in Argentina and plenty of opportunity to contrast the beliefs and practices of different social groups, one of my conclusions is that much of what is known as "social capital" or "high trust societies" amounts to a self-reinforcing tendency for people to play positive-sum games rather than negative-sum games, or to play positive-sum games in a wider context (not, for example, only within their own family); and this accounts for the otherwise puzzling overwhelming dominance of Wikipedia contributions from highly developed countries, far out of proportion to any difference in literate populations.

Here in Argentina, two centuries of often-militarized factionalism has created a profound cynicism about politics and about any attempt to carry out positive-sum projects; politicians get and maintain power by polarizing their constituents against other politicians and their constituents and by using the state to extract wealth to give to their own supporters, with the unsurprising result that everyone has concluded that all politicians are thieves and liars — because would-be politicians who are not thieves and liars are not successful at gaining support.

The popular belief here that all games are zero-sum is so strong that an Argentine woman told me once that Argentine cars are of poorer quality than American cars because American companies send their defective parts to Argentina, as if statistical process control worked by producing several times the needed quantity of goods and discarding most of them. This belief is strengthened by the Reagan-like deployment of positive-sum rhetoric by politicians to promote negative-sum policies that enrich the rich at the expense of the poor; Macri, our current president, is especially guilty of this, and his economic policies have been disastrous.

We can also argue for a subtler effect, similar to the renormalization argument in the previous section. Leaders of factionalist ignoramus societies, such as Donald Trump, Hugo Chávez, or Mauricio Macri, cannot afford to take advice from the intelligentsia, or put them in charge of policy, even if they could figure out who they were; they are forced to assume that wiser heads

from other factions are looking for ways to defeat them, and will use delegation of authority or even openness to advice as a way to undermine their leadership. (See Notch scorn (p. 115) for pompous bloviation on this topic.) Instead, they are forced to delegate authority to people loyal to their own faction, and listen only to their advice, with the predictably disastrous consequences we are seeing today in Venezuela.

(Given that Trump is president of the country with half of the world's top universities, it may seem strange to call it a "factionalist ignoramus society" — but Trump has apparently spent his entire life playing zero-sum and negative-sum games, and is certainly an ignoramus who surrounds himself with ignoramuses, and almost half the population of the country voted for him anyway. So clearly the factionalist ignoramus element of that society, always significant, is today its dominant element, just as in Germany after 1933.)

You might think that the pervasive belief in zero-sum-ness would make it impossible to motivate voters to participate in politics at all, since clearly whatever time you spend handing out campaign fliers or getting gassed by the cops (a nearly universal experience among the lower and middle class in Argentina) isn't going to be compensated by the marginally increased chance of your party winning at the polls and putting in place whatever policies you favor, even presuming those policies benefit you and not just the politicians you elected. But humans are not primarily motivated by such calculations, and never have been; any who were would fail Newcomb tests and be cast out of society to die in the wilderness.

Suppressing factionalism is no panacea either — whenever those in power pursue foolish policies, they can reasonably accuse anyone who criticizes those policies of fomenting factionalism, and this commonly happens in modern China, in the modern Bahá'í ecclesiastical hierarchy, and in the medieval Catholic church, for example, all of which have or had strong taboos against factionalism. Just as extreme factionalism deprives the leadership of the benefit of the collective cognition of the wise, so too does conformism.

Topics

A variety of code fragments for testing proposed language designs

Kragen Javier Sitaker, 2016-05-18 (19 minutes)

I've been thinking about code size, bytecode formats, and so on: what kind of tradeoffs in program representation are best? Existing microbenchmarks tend to be very atypical of real code; real code is mostly much more straightforward and boring than microbenchmarks.

So here are some sample code fragments to try different compilation strategies with (after hand-translating them to some uniform syntax, I suppose). These are chosen to cover the range of real code found in the wild, although they are biased towards code bases of high quality and towards medium-length functions: functions between 8 and 64 lines are better represented. There is code here in Pascal, C, C++, C#, Java, Golang, Perl, Tcl, Python, PHP, JS, Lua, Elisp, Ruby, Haskell, R, and Visual Basic — every popular programming language except Excel, Swift, assembly, SQL, and Matlab. The idea here is to include things written in somewhat different paradigms.

I thought about including code in bash, TeX, assembly, SQL, CSS, Prolog, and HTML, but their execution models are too different from the mainstream to be reasonably translatable to a mainstream virtual machine. Even R and Haskell are kind of pushing it.

The classic dumb Fibonacci, in C:

```
fib(n) { return n < 2 ? 1 : fib(n-1) + fib(n-2); }
```

A nearly totally vacuous class definition:

```
class Match:
    def __init__(self, length, rendering):
        self.length = length
        self.rendering = rendering
```

A test of the mod_pubsub JS interface, some code I think I wrote in 2000, unless it was Ben or Rohit:

```
function do_it()
{
    var topic = kn_argv['kn_topic'];
    kn_publish(topic, {}, { onSuccess: function(e) {succeed()},
                            onError: function(e) {fail(e.kn_payload)} });
}
```

Part of the Perl abstract base class DBD::File:

```
sub table_meta_attr_changed
{
    my ($class, $meta, $attrib, $value) = @_;
    defined $reset_on_modify{$attrib} and
        delete $meta->{$reset_on_modify{$attrib}} and
```

```
        $meta->{initialized} = 0;
    } # table_meta_attr_changed
```

The Lua code in LuaTeX for computing the full platform string on Linux:

```
function os.resolvers.platform(t,k)
    local platform = "linux"
    os.setenv("MTX_PLATFORM",platform)
    os.platform = platform
    return platform
end
```

A byteswapping function by Jon Lech Johansen in C#:

```
byte [] NetToHost( byte [] Input, int Pos, int Count )
{
    if( BitConverter.IsLittleEndian )
    {
        for( int i = 0; i < Count; i++ )
        {
            Array.Reverse( Input, Pos + (i * 4), 4 );
        }
    }

    return Input;
}
```

In GNU `cp`, some slight options parsing hackery:

```
/* For long options that have no equivalent short option, use a
   non-character as a pseudo short option, starting with CHAR_MAX + 1.  */
enum
{
  ATTRIBUTES_ONLY_OPTION = CHAR_MAX + 1,
  COPY_CONTENTS_OPTION,
  NO_PRESERVE_ATTRIBUTES_OPTION,
  PARENTS_OPTION,
  PRESERVE_ATTRIBUTES_OPTION,
  REFLINK_OPTION,
  SPARSE_OPTION,
  STRIP_TRAILING_SLASHES_OPTION,
  UNLINK_DEST_BEFORE_OPENING
};
```

A function from netqmail to downcase a NUL-terminated string:

```
#include "case.h"

void case_lowers(s)
char *s;
{
  unsigned char x;
  while (x = *s) {
    x -= 'A';
```

```
    if (x <= 'Z' - 'A') *s = x + 'a';
    ++s;
  }
}
```

Part of Jon Lech Johansen's DeDRMS utility in C#:

```
class M4PStream
{
    // ...
    public M4PStream( FileStream fs )
    {
        br = new BinaryReader( fs );
        bw = new BinaryWriter( fs );
        sbuffer = br.ReadBytes( Convert.ToInt32( fs.Length ) );

        alg = Rijndael.Create();
        alg.Mode = CipherMode.CBC;
        alg.Padding = PaddingMode.None;
    }
    // ...
}
```

A function from the Nouveau video driver in the Linux kernel:

```
u16
nvbios_cstepEe(struct nvkm_bios *bios, int idx, u8 *ver, u8 *hdr)
{
        u8  cnt, len, xnr, xsz;
        u16 data = nvbios_cstepTe(bios, ver, hdr, &cnt, &len, &xnr, &xsz);
        if (data && idx < cnt) {
                data = data + *hdr + (idx * len);
                *hdr = len;
                return data;
        }
        return 0x0000;
}
```

Part of the Elisp code for Flymake, which gives you on-the-fly error checking in Emacs, in this case converting between a couple of slightly different data formats in order to pop up a list of suggested corrections when you click on an error:

```
(defun flymake-make-emacs-menu (menu-data)
  "Return a menu specifier using MENU-DATA.
MENU-DATA is a list of error and warning messages returned by
`flymake-make-err-menu-data'.
See `x-popup-menu' for the menu specifier format."
  (let* ((menu-title     (nth 0 menu-data))
         (menu-items     (nth 1 menu-data))
         (menu-commands  (mapcar (lambda (foo)
                                   (cons (nth 0 foo) (nth 1 foo)))
                                 menu-items)))
    (list menu-title (cons "" menu-commands))))
```

The SafeBuffer indexing method from Ruby ActiveSupport:

```
def [](*args)
  return super if args.size < 2

  if html_safe?
    new_safe_buffer = super
    new_safe_buffer.instance_eval { @html_safe = true }
    new_safe_buffer
  else
    to_str[*args]
  end
end
```

A function from the Linux network scheduling code:

```
static int ingress_dump(struct Qdisc *sch, struct sk_buff *skb)
{
        struct nlattr *nest;

        nest = nla_nest_start(skb, TCA_OPTIONS);
        if (nest == NULL)
                goto nla_put_failure;

        return nla_nest_end(skb, nest);

nla_put_failure:
        nla_nest_cancel(skb, nest);
        return -1;
}
```

A function from Godoc, the Golang documentation extractor:

```
func findSpec(list []ast.Spec, id *ast.Ident) ast.Spec {
        for _, spec := range list {
                switch s := spec.(type) {
                case *ast.ImportSpec:
                        if s.Name == id {
                                return s
                        }
                case *ast.ValueSpec:
                        for _, n := range s.Names {
                                if n == id {
                                        return s
                                }
                        }
                case *ast.TypeSpec:
                        if s.Name == id {
                                return s
                        }
                }
        }
        return nil
}
```

Part of the GNU ISO C++ library implementing the thin_heap type:

```
PB_DS_CLASS_T_DEC
inline void
PB_DS_CLASS_C_DEC::
fix_sibling_rank_1_unmarked(node_pointer p_y)
{
  _GLIBCXX_DEBUG_ASSERT(p_y->m_p_prev_or_parent != 0);

  _GLIBCXX_DEBUG_ONLY(node_pointer p_w = p_y->m_p_l_child;)
  _GLIBCXX_DEBUG_ASSERT(p_w != 0);
  _GLIBCXX_DEBUG_ASSERT(p_w->m_p_next_sibling == 0);
  _GLIBCXX_DEBUG_ASSERT(p_y->m_p_next_sibling == 0);

  p_y->m_p_next_sibling = p_y->m_p_l_child;
  p_y->m_p_next_sibling->m_p_prev_or_parent = p_y;
  p_y->m_p_l_child = 0;
  PB_DS_ASSERT_NODE_CONSISTENT(p_y, false)
}
```

Part of the tests for d3.js:

```
"can output a percentage with rounding and sign": function(format) {
  var f = format("+.2p");
  assert.strictEqual(f(.00123), "+0.12%");
  assert.strictEqual(f(.0123), "+1.2%");
  assert.strictEqual(f(.123), "+12%");
  assert.strictEqual(f(1.23), "+120%");
  assert.strictEqual(f(-.00123), "-0.12%");
  assert.strictEqual(f(-.0123), "-1.2%");
  assert.strictEqual(f(-.123), "-12%");
  assert.strictEqual(f(-1.23), "-120%");
},
```

Another function from netqmail, this one for comparing two strings lexically:

```
int str_diffn(s,t,len)
register char *s;
register char *t;
unsigned int len;
{
  register char x;

  for (;;) {
    if (!len--) return 0; x = *s; if (x != *t) break; if (!x) break; ++s; ++t;
    if (!len--) return 0; x = *s; if (x != *t) break; if (!x) break; ++s; ++t;
    if (!len--) return 0; x = *s; if (x != *t) break; if (!x) break; ++s; ++t;
    if (!len--) return 0; x = *s; if (x != *t) break; if (!x) break; ++s; ++t;
  }
  return ((int)(unsigned int)(unsigned char) x)
       - ((int)(unsigned int)(unsigned char) *t);
}
```

A module from the Free Pascal compiler providing access to an AmigaOS timer function:

```
unit timerutils;

interface

uses exec, timer, amigalib;

Function CreateTimer(theUnit : longint) : pTimeRequest;
{ some stuff omitted }

implementation

Function CreateTimer(theUnit : longint) : pTimeRequest;
var
    Error : longint;
    TimerPort : pMsgPort;
    TimeReq : pTimeRequest;
begin
    TimerPort := CreatePort(Nil, 0);
    if TimerPort = Nil then
        CreateTimer := Nil;
    TimeReq := pTimeRequest(CreateExtIO(TimerPort,sizeof(tTimeRequest)));
    if TimeReq = Nil then begin
        DeletePort(TimerPort);
        CreateTimer := Nil;
    end;
    Error := OpenDevice(TIMERNAME, theUnit, pIORequest(TimeReq), 0);
    if Error <> 0 then begin
        DeleteExtIO(pIORequest(TimeReq));
        DeletePort(TimerPort);
        CreateTimer := Nil;
    end;
    TimerBase := pointer(TimeReq^.tr_Node.io_Device);
    CreateTimer := pTimeRequest(TimeReq);
end;
{ lots of stuff omitted }
end.
```

From quodlibet, some PyGTK code with a callback:

```
def edit_patterns(cls, button):
    def valid_uri(s):
        # TODO: some pattern validation too (that isn't slow)
        try:
            p = Pattern(s)
            u = URI(s)
            return (p and u.netloc and
                u.scheme in ["http", "https", "ftp", "file"])
        except ValueError:
            return False

    win = StandaloneEditor(filename=cls.PATTERNS_FILE,
        title=_("Search URL patterns"), initial=cls.DEFAULT_URL_PATS,
```

```
        validator=valid_uri)
    win.show()
```

From slide-rule.tcl, some Tk code with callbacks that I wrote in like 1997:

```
proc sl_ru_bind {canvas} {
        bind $canvas <1> {
                global old_x
                set old_x %x
                global dragobj
                set dragobj [groupof %W [%W find closest %x %y]]
        }
        bind $canvas <B1-Motion> {
                global old_x dragobj
                %W move $dragobj [expr (%x - $old_x) / 2] 0
                set old_x %x
        }
}
```

A perspective projection in JS from some code I wrote to design domes:

```
function project(points, zDist, zMin, width) {
  var rv = [];
  for (var ii = 0; ii < points.length; ii++) {
    var p = points[ii]
      , xx = p[0]
      , yy = p[1]
      , zz = p[2] + zDist
    ;
    if (zz > zMin) {
      rv.push([width * xx / zz, width * yy / zz]);
    } else {
      rv.push([NaN, NaN]);
    }
  }
  return rv;
}
```

Some disappointingly boilerplatish code from Lucene:

```
  public String toString(String field) {
    StringBuffer buffer = new StringBuffer();
    buffer.append("spanNot(");
    buffer.append(include.toString(field));
    buffer.append(", ");
    buffer.append(exclude.toString(field));
    buffer.append(")");
    buffer.append(ToStringUtils.boost(getBoost()));
    return buffer.toString();
  }
```

From Darius's constraint-programming draft in JS:

```
// Try to reduce eqns to an equivalent system with each variable
// defined by a single equation. (The result may be inconsistent
// or underconstrained.)
function reduceEquations(eqns) {
    for (let i = 0; i < eqns.length; ++i) {
        const eqi = eqns[i];
        const v = eqi.aVariable();
        if (v === null) continue;
        for (let j = 0; j < i; ++j) {
            eqns[j] = eqns[j].substituteFor(v, eqi);
            if (eqns[j].isInconsistent()) {
                return {isConsistent: false};
            }
        }
    };
    return {isConsistent: true,
            equations: (eqns.filter(eqn => !eqn.isTautology())
                           .map(eqn => eqn.normalize()))};
}
```

The glue code in the Lua 5.1 OS interface to allow Lua code to call mktime(3):

```
static int os_time (lua_State *L) {
  time_t t;
  if (lua_isnoneornil(L, 1))  /* called without args? */
    t = time(NULL);  /* get current time */
  else {
    struct tm ts;
    luaL_checktype(L, 1, LUA_TTABLE);
    lua_settop(L, 1);  /* make sure table is at the top */
    ts.tm_sec = getfield(L, "sec", 0);
    ts.tm_min = getfield(L, "min", 0);
    ts.tm_hour = getfield(L, "hour", 12);
    ts.tm_mday = getfield(L, "day", -1);
    ts.tm_mon = getfield(L, "month", -1) - 1;
    ts.tm_year = getfield(L, "year", -1) - 1900;
    ts.tm_isdst = getboolfield(L, "isdst");
    t = mktime(&ts);
  }
  if (t == (time_t)(-1))
    lua_pushnil(L);
  else
    lua_pushnumber(L, (lua_Number)t);
  return 1;
}
```

A fairly boilerplatish class definition in Perl:

```
package Pod::Simple::PullParserToken;
 # Base class for tokens gotten from Pod::Simple::PullParser's $parser->get_token
@ISA = ();
$VERSION = '3.16';
use strict;
```

```
sub new {  # Class->new('type', stuff...);  ## Overridden in derived classes anywo
oay
  my $class = shift;
  return bless [@_], ref($class) || $class;
}

sub type { $_[0][0] }  # Can't change the type of an object
sub dump { Pod::Simple::pretty( [ @{ $_[0] } ] ) }

sub is_start { $_[0][0] eq 'start' }
sub is_end   { $_[0][0] eq 'end'   }
sub is_text  { $_[0][0] eq 'text'  }

1;
__END__
```

Some R code from the CRAN survival analysis package, doing some kind of magic metaprogramming hackery that I don't understand:

```
as.character.Surv <- function(x, ...) {
    if (is.R()) class(x) <- NULL
    else        oldClass(x) <- NULL
    type <- attr(x, 'type')
    if (type=='right') {
        temp <- x[,2]
        temp <- ifelse(is.na(temp), "?", ifelse(temp==0, "+"," "))
        paste(format(x[,1]), temp, sep='')
        }
    else if (type=='counting') {
        temp <- x[,3]
        temp <- ifelse(is.na(temp), "?", ifelse(temp==0, "+"," "))
        paste('(', format(x[,1]), ',', format(x[,2]), temp,
                        ']', sep='')
        }
    else if (type=='left') {
        temp <- x[,2]
        temp <- ifelse(is.na(temp), "?", ifelse(temp==0, "<"," "))
        paste(temp, format(x[,1]), sep='')
        }
    else {   #interval type
        stat <- x[,3]
        temp <- c("+", "", "-", "]")[stat+1]
        temp2 <- ifelse(stat==3,
                        paste("[", format(x[,1]), ", ",format(x[,2]), sep=''),
                        format(x[,1]))
        ifelse(is.na(stat), "NA", paste(temp2, temp, sep=''))
        }
    }
```

A method from Cynthiune, an MP3 player written in Objective-C for GNUStep, showing the old style retain/release/autorelease memory-management for NeXTSTep apps:

```
@implementation M3UUnarchiver : PlaylistUnarchiver

+ (NSArray *) _fileListFromLines: (NSArray *) arrayOfLines
            inReferenceDirectory: (NSString *) directory
{
  NSMutableArray *filelist;
  NSEnumerator *lines;
  NSString *currLine, *newString;

  filelist = [NSMutableArray new];
  [filelist autorelease];

  lines = [arrayOfLines objectEnumerator];
  currLine = [lines nextObject];
  while (currLine)
    {
      if (![currLine hasPrefix: @"#"] && [currLine length])
        {
          newString = [NSString stringWithString: currLine];
          if (![newString isAbsolutePath])
            newString = [directory stringByAppendingPathComponent: newString];
          newString = [newString stringByStandardizingPath];
          [filelist addObject: newString];
        }
      currLine = [lines nextObject];
    }

  return filelist;
}
```

A C part of the SQLite test suite:

```
/*
** Usage: sqlite3_shared_cache_report
**
** Return a list of file that are shared and the number of
** references to each file.
*/
int sqlite3BtreeSharedCacheReport(
  void * clientData,
  Tcl_Interp *interp,
  int objc,
  Tcl_Obj *CONST objv[]
){
#ifndef SQLITE_OMIT_SHARED_CACHE
  extern BtShared *sqlite3SharedCacheList;
  BtShared *pBt;
  Tcl_Obj *pRet = Tcl_NewObj();
  for(pBt=GLOBAL(BtShared*,sqlite3SharedCacheList); pBt; pBt=pBt->pNext){
    const char *zFile = sqlite3PagerFilename(pBt->pPager, 1);
    Tcl_ListObjAppendElement(interp, pRet, Tcl_NewStringObj(zFile, -1));
    Tcl_ListObjAppendElement(interp, pRet, Tcl_NewIntObj(pBt->nRef));
  }
  Tcl_SetObjResult(interp, pRet);
#endif
```

```
  return TCL_OK;
}
```

A small part of the SQLite test suite, in Tcl:

```
# Convert a string containing EXPR, AGG, and BOOL into a string
# that contains nothing but X, Y, and Z.
#
proc extract_vars {a} {
  regsub -all {EXPR} $a X a
  regsub -all {AGG} $a Y a
  regsub -all {BOOL} $a Z a
  regsub -all {[^XYZ]} $a {} a
  return $a
}

# Test all templates to make sure the number of EXPR, AGG, and BOOL
# expressions match.
#
foreach term [concat $aggexpr $intexpr $boolexpr] {
  foreach {a b} $term break
  if {[extract_vars $a]!=[extract_vars $b]} {
    error "mismatch: $term"
  }
}
```

From some dumb test code I wrote for a C random music generator:

```
int
main()
{
  int ii, jj, kk;
  score sc;
  srand(getpid());
  generate_score(&sc);
  for (ii = 0; ii != 256; ii++) {
    for (jj = 0; jj != 8192; jj++) {
      unsigned char samp = 0;
      for (kk = 0; kk < 6; kk++) {
        int t = ii << (kk/2) | jj >> (13 - (kk/2));
        int f = freq(next_note(t, &sc, kk));
        samp += jj * f * (1 << kk) >> 10 & 32;
      }
      printf("%c", samp);
    }
  }
  return 0;
}
```

More code from SQLite, this time in one of its memory allocators:

```
/*
** Unlink the chunk at mem5.aPool[i] from list it is currently
** on.  It should be found on mem5.aiFreelist[iLogsize].
```

```
*/
static void memsys5Unlink(int i, int iLogsize){
  int next, prev;
  assert( i>=0 && i<mem5.nBlock );
  assert( iLogsize>=0 && iLogsize<=LOGMAX );
  assert( (mem5.aCtrl[i] & CTRL_LOGSIZE)==iLogsize );

  next = MEM5LINK(i)->next;
  prev = MEM5LINK(i)->prev;
  if( prev<0 ){
    mem5.aiFreelist[iLogsize] = next;
  }else{
    MEM5LINK(prev)->next = next;
  }
  if( next>=0 ){
    MEM5LINK(next)->prev = prev;
  }
}
```

A function in the Maybe monad from Brandon Moore's Haskell ray tracer:

```
hit :: Vec -> Vec -> Sphere -> Maybe (Float, Vec, Vec)
hit x d (Sphere r center _ _) = do
  let face = diff x center
      a = dot d d
      b = 2*dot d face
      c = (dot face face) - (r*r)
      disc = b^2 - 4*a*c
  guard (b < 0) -- vector from center to camera should point opposite from ray
  guard (disc >= 0)
  let t = 2*c / (-b + sqrt disc)
      intersection = x `add` scale t d
      normal = normalize (diff intersection center)
      xbounce = dot d normal
      reflected = add d (scale (-2*xbounce) normal)
  return (t, intersection, reflected)
```

From Lucene:

```
public class DateTools {

  private final static TimeZone GMT = TimeZone.getTimeZone("GMT");


  private static final SimpleDateFormat YEAR_FORMAT = new SimpleDateFormat("yyyy"o
o, Locale.US);

  private static final SimpleDateFormat MONTH_FORMAT = new SimpleDateFormat("yyyyo
oMM", Locale.US);

  private static final SimpleDateFormat DAY_FORMAT = new SimpleDateFormat("yyyyMMo
odd", Locale.US);

  private static final SimpleDateFormat HOUR_FORMAT = new SimpleDateFormat("yyyyMo
```

```
oMddHH", Locale.US);

   private static final SimpleDateFormat MINUTE_FORMAT = new SimpleDateFormat("yyyo
oyMMddHHmm", Locale.US);

   private static final SimpleDateFormat SECOND_FORMAT = new SimpleDateFormat("yyyo
oyMMddHHmmss", Locale.US);

   private static final SimpleDateFormat MILLISECOND_FORMAT = new SimpleDateFormato
o("yyyyMMddHHmmssSSS", Locale.US);
   static {
     // times need to be normalized so the value doesn't depend on the
     // location the index is created/used:
     YEAR_FORMAT.setTimeZone(GMT);
     MONTH_FORMAT.setTimeZone(GMT);
     DAY_FORMAT.setTimeZone(GMT);
     HOUR_FORMAT.setTimeZone(GMT);
     MINUTE_FORMAT.setTimeZone(GMT);
     SECOND_FORMAT.setTimeZone(GMT);
     MILLISECOND_FORMAT.setTimeZone(GMT);
   }

   private static final Calendar calInstance = Calendar.getInstance(GMT);

   // cannot create, the class has static methods only
   private DateTools() {}
   // ...
 }
```

From shlex in the Python standard library:

```
def read_token(self):
    quoted = False
    escapedstate = ' '
    while True:
        nextchar = self.instream.read(1)
        if nextchar == '\n':
            self.lineno = self.lineno + 1
        if self.debug >= 3:
            print "shlex: in state", repr(self.state), \
                  "I see character:", repr(nextchar)
        if self.state is None:
            self.token = ''        # past end of file
            break
        elif self.state == ' ':
            if not nextchar:
                self.state = None  # end of file
                break
            elif nextchar in self.whitespace:
                if self.debug >= 2:
                    print "shlex: I see whitespace in whitespace state"
                if self.token or (self.posix and quoted):
                    break   # emit current token
                else:
                    continue
```

```
            elif nextchar in self.commenters:
                self.instream.readline()
                self.lineno = self.lineno + 1
            elif self.posix and nextchar in self.escape:
                escapedstate = 'a'
                self.state = nextchar
            elif nextchar in self.wordchars:
                self.token = nextchar
                self.state = 'a'
```

Some test code for a Visual Basic interface to mod_pubsub:

```
Module TestUtil
    Public TU_OK As String = "OK"

    Dim TU_VERBOSE_ALL As Boolean = False
    Dim TU_TESTSUITE As String = ""
    Dim TU_TESTCASE As String = ""
    Dim TU_SERVER As String = ""
    Dim TU_MISSINGSERVER As String = ""
    Dim TU_SERVERNAME As String = ""
    Dim tu_servers As New ArrayList()

    Dim tu_count_subListener_OnUpdate As Integer = 0
    Dim tu_count_subConnSH_OnConnStatus As Integer = 0
    Dim tu_count_subReqSH_OnError As Integer = 0
    Dim tu_count_subReqSH_OnSuccess As Integer = 0
    Dim tu_count_pubConnSH_OnConnStatus As Integer = 0
    Dim tu_count_pubReqSH_OnError As Integer = 0
    Dim tu_count_pubReqSH_OnSuccess As Integer = 0

    '////////////////////////////////////////////////////////////////////
    '/// Public Events

            '////////////////////////////////////////////////////////////////////o
o/////

    Public Class TU_SubListener
        Inherits LibKNDotNet.IListener

        Public Overrides Sub OnUpdate(ByVal msg As LibKNDotNet.Message)
            TU_dumpMsg("TU_SubListener.OnUpdate", msg)
            tu_count_subListener_OnUpdate = tu_count_subListener_OnUpdate + 1
        End Sub

    End Class

    Public Sub TU_INIT_TESTCASE(ByVal testCaseName As String)
        TU_TESTCASE = testCaseName
        If TU_VERBOSE_ALL = False Then
            Return
        End If
        Console.WriteLine()

        Console.WriteLine("================================================o
```

```
o========================")
        Console.WriteLine("=== Start test: " & TU_TESTSUITE & "." & testCaseName)

        Console.WriteLine("==================================================o
o========================")
    End Sub

    ' lots more stuff here omitted
End Module
```

A PHP class from Laravel:

```
namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvideo
or;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @param  \Illuminate\Contracts\Auth\Access\Gate  $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);

        //
    }
}
```

A database schema migration in PHP from Laravel:

```
class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
```

```
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('users');
    }
}
```

Part of the Python 3.2 marshal module:

```
static void
w_object(PyObject *v, WFILE *p)
{
    Py_ssize_t i, n;

    p->depth++;

    if (p->depth > MAX_MARSHAL_STACK_DEPTH) {
        p->error = WFERR_NESTEDTOODEEP;
    }
    else if (v == NULL) {
        w_byte(TYPE_NULL, p);
    }
    else if (v == Py_None) {
        w_byte(TYPE_NONE, p);
    }
    else if (v == PyExc_StopIteration) {
        w_byte(TYPE_STOPITER, p);
    }
    else if (v == Py_Ellipsis) {
        w_byte(TYPE_ELLIPSIS, p);
    }
    else if (v == Py_False) {
        w_byte(TYPE_FALSE, p);
    }
    else if (v == Py_True) {
        w_byte(TYPE_TRUE, p);
    }
    else if (PyLong_CheckExact(v)) {
        long x = PyLong_AsLong(v);
        if ((x == -1)  && PyErr_Occurred()) {
            PyLongObject *ob = (PyLongObject *)v;
```

```
            PyErr_Clear();
            w_PyLong(ob, p);
        }
        else {
#if SIZEOF_LONG > 4
            long y = Py_ARITHMETIC_RIGHT_SHIFT(long, x, 31);
            if (y && y != -1) {
                w_byte(TYPE_INT64, p);
                w_long64(x, p);
            }
            else
#endif
            {
                w_byte(TYPE_INT, p);
                w_long(x, p);
            }
        }
    }
    // other cases omitted
}
```

A three-page function from GNU `cp`:

```
static bool
make_dir_parents_private (char const *const_dir, size_t src_offset,
                          char const *verbose_fmt_string,
                          struct dir_attr **attr_list, bool *new_dst,
                          const struct cp_options *x)
{
  struct stat stats;
  char *dir;            /* A copy of CONST_DIR we can change.  */
  char *src;            /* Source name in DIR.  */
  char *dst_dir;        /* Leading directory of DIR.  */
  size_t dirlen;        /* Length of DIR.  */

  ASSIGN_STRDUPA (dir, const_dir);

  src = dir + src_offset;

  dirlen = dir_len (dir);
  dst_dir = alloca (dirlen + 1);
  memcpy (dst_dir, dir, dirlen);
  dst_dir[dirlen] = '\0';

  *attr_list = NULL;

  if (stat (dst_dir, &stats) != 0)
    {
      /* A parent of CONST_DIR does not exist.
         Make all missing intermediate directories. */
      char *slash;

      slash = src;
      while (*slash == '/')
        slash++;
```

```
while ((slash = strchr (slash, '/')))
  {
    struct dir_attr *new IF_LINT ( = NULL);
    bool missing_dir;

    *slash = '\0';
    missing_dir = (stat (dir, &stats) != 0);

    if (missing_dir || x->preserve_ownership || x->preserve_mode
        || x->preserve_timestamps)
      {
        /* Add this directory to the list of directories whose
           modes might need fixing later. */
        struct stat src_st;
        int src_errno = (stat (src, &src_st) != 0
                         ? errno
                         : S_ISDIR (src_st.st_mode)
                         ? 0
                         : ENOTDIR);
        if (src_errno)
          {
            error (0, src_errno, _("failed to get attributes of %s"),
                   quote (src));
            return false;
          }

        new = xmalloc (sizeof *new);
        new->st = src_st;
        new->slash_offset = slash - dir;
        new->restore_mode = false;
        new->next = *attr_list;
        *attr_list = new;
      }

    if (missing_dir)
      {
        mode_t src_mode;
        mode_t omitted_permissions;
        mode_t mkdir_mode;

        /* This component does not exist.  We must set
           *new_dst and new->st.st_mode inside this loop because,
           for example, in the command `cp --parents ../a/../b/c e_dir',
           make_dir_parents_private creates only e_dir/../a if
           ./b already exists. */
        *new_dst = true;
        src_mode = new->st.st_mode;

        /* If the ownership or special mode bits might change,
           omit some permissions at first, so unauthorized users
           cannot nip in before the file is ready.  */
        omitted_permissions = (src_mode
                               & (x->preserve_ownership
                                  ? S_IRWXG | S_IRWXO
                                  : x->preserve_mode
```

```
                        ? S_IWGRP | S_IWOTH
                        : 0));

/* POSIX says mkdir's behavior is implementation-defined when
   (src_mode & ~S_IRWXUGO) != 0.  However, common practice is
   to ask mkdir to copy all the CHMOD_MODE_BITS, letting mkdir
   decide what to do with S_ISUID | S_ISGID | S_ISVTX.  */
mkdir_mode = src_mode & CHMOD_MODE_BITS & ~omitted_permissions;
if (mkdir (dir, mkdir_mode) != 0)
  {
    error (0, errno, _("cannot make directory %s"),
           quote (dir));
    return false;
  }
else
  {
    if (verbose_fmt_string != NULL)
      printf (verbose_fmt_string, src, dir);
  }

/* We need search and write permissions to the new directory
   for writing the directory's contents. Check if these
   permissions are there.  */

if (lstat (dir, &stats))
  {
    error (0, errno, _("failed to get attributes of %s"),
           quote (dir));
    return false;
  }

if (! x->preserve_mode)
  {
    if (omitted_permissions & ~stats.st_mode)
      omitted_permissions &= ~ cached_umask ();
    if (omitted_permissions & ~stats.st_mode
        || (stats.st_mode & S_IRWXU) != S_IRWXU)
      {
        new->st.st_mode = stats.st_mode | omitted_permissions;
        new->restore_mode = true;
      }
  }

if ((stats.st_mode & S_IRWXU) != S_IRWXU)
  {
    /* Make the new directory searchable and writable.
       The original permissions will be restored later.  */

    if (lchmod (dir, stats.st_mode | S_IRWXU) != 0)
      {
        error (0, errno, _("setting permissions for %s"),
               quote (dir));
        return false;
      }
  }
```

```
          }
        else if (!S_ISDIR (stats.st_mode))
          {
            error (0, 0, _("%s exists but is not a directory"),
                   quote (dir));
            return false;
          }
        else
          *new_dst = false;
        *slash++ = '/';

        /* Avoid unnecessary calls to `stat' when given
           file names containing multiple adjacent slashes.  */
        while (*slash == '/')
          slash++;
      }
  }

/* We get here if the parent of DIR already exists.  */

else if (!S_ISDIR (stats.st_mode))
  {
    error (0, 0, _("%s exists but is not a directory"), quote (dst_dir));
    return false;
  }
else
  {
    *new_dst = false;
  }
return true;
}
```

Topics

Snap logic

Kragen Javier Sitaker, 2018-06-17 (3 minutes)

I keep thinking about the explanation in the Art of Electronics† of how reading DRAM works. Each column of the DRAM has a sense amplifier, which is kind of like a differential amplifier — it's actually a latch consisting of two inverters in a loop. Before reading, the loop is shorted to force the inverters into a metastable state; then the short is released and the appropriate row of capacitors is shorted to the sense amplifiers.

This unbalances the previously metastable latches, which then fall into one of their two stable states. In the process, they charge the capacitor all the way to the rail.

This is very similar to Merkle's buckling-spring logic, in which a gate begins in a stable state which is made metastable by the application of a buckling force, which amplifies the balance of forces applied to the spring into a large displacement driven by the buckling force itself. Merkle proposed using these as majority-rule gates with a constant bias input to get AND and OR.

I wonder if such circuits could be a productive way to design digital circuits even with current technology, integrating state and combinational logic rather than keeping them separate. As one simple example, you could build a bidirectional shift register by using three latches per bit, energizing only ⅓ to ⅔ of the latches at any given time, passing the state along in the manner of a CCD or a Dekatron. Or you could build an image memory that directly supports not only shifting the image off the chip but also operations like dilation.

My hunch is that this approach ends up being more or less equivalent to using master-slave flip-flops.

† "The sense amplifiers are latching devices, here drawn notionally as fed-back noninverting amplifiers. (In practice they are implemented as flip-flops that begin the cycle in a balanced state and become unbalanced by the bit-capacitor charge that is switched into them. … In a further dose of reality, things are a bit more complicated: the sense amplifiers are *differential*, and the DRAM array is usually built in a "folded-bit" arrangement so that any given row line activates only the even or odd cells; the inactive (neutral) bit line floats at the precharge level (Vdd/2) and acts as a reference voltage by which the balanced sense amplifier compares the ΔV "bump" up or down from the capacitor's charge in the respective bit cell. …"

Topics

Millikiln

Kragen Javier Sitaker, 2017-01-17 (updated 2017-03-02) (4 minutes)

(See also An electric furnace the size of a sake cup (p. 666) for more thoughts along these lines.)

Suppose I want a ceramics-firing kiln that holds about a liter, about a thousandth the capacity of the big kiln in the studio; call it a "millikiln". Maybe I can make it by packing calcium oxide into a ceramic jar, putting some kind of heating element inside, and cooling the jar on the outside so the ceramic doesn't melt.

I'm going to guess that 30 mm of calcium oxide is adequate insulation. The inside volume of 1 liter could be 100 mm vertically and 112 mm across; the outside volume, including the lid, could be 160 mm vertically and 172 mm across, a total of about 3.7 liters, which means 2.7 liters of calcium oxide. The most practical way to obtain that is to calcine about 2.7 liters of calcium carbonate, whose specific gravity in the calcite form is 2.71, so that's about 7.3 kg of calcium carbonate.

Is that right? Calcium carbonate is $CaCO_3$, while calcium oxide (CaO) has a density of 3.34 g/cc, which is *higher*. And, indeed, WP says, "The reaction of quicklime with water is associated with an increase in volume by a factor of at least 2.5," while "Approximately 1.8 t of limestone is required per 1.0 t of quicklime."

This suggests that calcining the calcium carbonate will reduce its volume by a factor of 2.2 to more than 2.5, which probably will not leave it solid.

Also, what about harmful gas emissions during firing? The 2.7 liters of CaO will weigh 9 kg and be made from a bit over 16 kg of calcium carbonate; 44% of the mass of the calcium carbonate (the other 7 kg) is lost as carbon dioxide; 7 kg of carbon dioxide is about 3.5 cubic meters ((12+16+16)/(16+17) = 1.33 times the density of air, which is 1.2 g/ℓ at STP, so CO_2 is 1.6 g/ℓ, XXX wrong 1.98). CO_2 is safe and beneficial to plants at 1000 ppm, but becomes problematic to humans around 5000 ppm (though submarines commonly have it up to 20000 ppm for long periods of time). 7 kg gives you 5000 ppm by mass when mixed into 1400 kg of air, occupying about 1200 m³, which as a sphere would be a bit over 13 meters in diameter.

This is larger than the pottery studio, which is perhaps 6 m × 6 m × 12 m, a total of 432 m³ or 518 kg of air; so it should be safe to release about 2 kg of CO_2 into it at a time, obtained by calcining some 4.5 kg of calcium carbonate.

Perhaps a different refractory would be more suitable. Maybe ordinary fireclay, which seems to cost AR$10 to AR$20 per kg on Mercadolibre, although I have my doubts about whether the various products sold there as "arcilla refractaria" are in fact any kind of clay at all. Maybe perlite, although supposedly perlite is only good up to 1000°, and vermiculite is hydrophilic (though much less so than quicklime!). Some sources suggest mixing in sawdust and burning it out to increase porosity of the refractory without perlite.

(Vermiculite also melts at 1100°C or below.)

Topics

Mic energy harvesting

Kragen Javier Sitaker, 2016-09-07 (updated 2016-09-08) (5 minutes)

Could you power a computer from your voice via a microphone?

Typical speakers are about 1% to 5% efficient. Suppose a microphone is about the same. Someone shouting at a distance of a meter is supposedly about 85dBa; 0 dBa (0 dB SIL) is defined as 10^{-12} W/m², so this is about 3 mW/m². The shout is maybe spread over an area of a square meter, so it's probably about 3mW in total, and you could probably capture a significant fraction of that if you had the microphone actually in your mouth. 3mW is plenty of power to run certain microcontrollers.

There's another issue of impedance and voltage matching. Piezo and electret microphones are essentially capacitive, making them very high impedance at audio frequencies, while dynamic microphones are essentially inductive, which seems more circuit-friendly. But since we're dealing with an audio-frequency AC waveform, you ought to be able to step it up or down with a steel-core transformer.

Otherwise, though, I think you could probably just feed a low-capacitance capacitor with the rectified dynamic mic, run a high-efficiency boost or buck converter from it to step the voltage up or down, and feed a low-voltage, high-capacitance capacitor with that to run the microcontroller.

For example, the LTC3388 is designed for this kind of application, accepting voltages of 2.7 to 20 V, with 720 nA quiescent current up to 820 nA at 20 V; different configurations manage output voltages down to 1.2 V; the efficiency is about 90% with low input voltages, dropping to about 70% for 20 V input, unless the load is under 100 µA. 100 µA at 1.8 V (typical for low-power microcontrollers) would be 180 µW, which would allow a one-second 3-mW burst of sound to power the microcontroller for 16⅔ s. Various low-power microcontrollers manage a bit under a nanojoule per instruction (like 300 picojoules to 900 picojoules per instruction), so 180 µW is about 360 000 instructions per second, comparable to a Commodore 64.

I have the intuition that for this kind of thing, it's important to be able to draw down the charge stored on the input side fast enough to follow the sound waveform on its way down, to ensure that the available energy there is successfully harvested. The LTC3388 can source up to 50 mA, which would be 90 mW at 1.8 V, which would be 4.5 mA at 20 V or 10 mA at 9 V. If the sound wave had most of its energy in a 200 Hz component, it falls from 9 V to 0 in 0.74 ms; without getting into actually calculating derivatives precisely, that would require that the input capacitor be no larger than about 0.8 µF.

(We shouldn't really try to analyze it linearly into sinusoids, because both the rectifier charging the input capacitor and the buck converter draining it are nonlinear, but that should give us a ballpark.)

I don't know how much input capacitance we need, or if we need any, but it seems like the LTC3388 lets its inductor current ramp up to 150 mA before shutting the gate, so we probably need enough energy stored to get a 22 µH inductor up to 150 mA. $LI^2/2$ is then about a quarter of a microjoule. A capacitor that charges up to a

quarter of a microjoule at, say, 7 volts would be .25 J / (7 V)² = 5000 pF. So probably anything in the middle of the 5000 to 800 000 pF range would work.

My intuition is that if the input capacitor is ever *above* the rectified signal, no current will flow, so no energy is harvested; while if it's far *below* the rectified signal, then most of the voltage has to be dropped by the rectifier itself rather than the capacitor, and that dropped voltage also represents inefficiency. So I think you want to ensure that the input capacitor stays just below the constantly changing supply voltage, at least until the output capacitor is fully charged.

I don't think the LTC3388 will do that, at all, and I don't know what will.

One thing that might help would be an inductor in series with the rectifier. This would keep a significant voltage from ever being dropped across the rectifier, instead charging the inductor until the capacitor is ready to accept its energy. But maybe that will make it hard to ramp up harvesting when there's a rapid rise in voltage?

Topics

- Physics (p. 3632) (119 notes)
- Energy (p. 3438) (63 notes)
- Energy harvesting (p. 3437) (11 notes)

Yeso notes

Kragen Javier Sitaker, 2018-12-25 (updated 2019-01-01) (11 minutes)

It's 2018-12-25, and xshmu is about 1800 lines of code (70 kilobytes), and overdue for its first refactoring. So far it has a terminal emulator, a calculator application, a Tetris game, an audio oscilloscope app, some graphics demos, a Chifir virtual machine, and a couple of sort-of paint programs, with backends for the Linux framebuffer console and X11:

```
 4105 admu.c
 4128 admu_shell.c
 1310 admu_tv_typewriter.c
 3936 chifir.c
 1338 chifir_xshmu.c
 2335 decimal.c
  484 decimal.h
 1562 glyphed.c
 2235 oscope.c
 6909 rpncalc.c
  375 rpncalc.h
 7385 tetris.c
 6778 wercaμ.c
13218 xshmu.c
 1899 xshmucalc.c
 6056 xshmu_fb.c
 4939 xshmu.h
  134 xshmu_hello.c
  525 xshmunch.c
  810 μpaint.c
$ cloc μpaint.c xshmu.c xshmunch.c chifir.c chifir_xshmu.c wercaμ.c \
    admu_tv_typewriter.c admu.c admu_shell.c xshmu_hello.c tetris.c \
    xshmu_fb.c glyphed.c oscope.c \
    xshmucalc.c rpncalc.c decimal.c decimal.h rpncalc.h xshmu.h
      20 text files.
      20 unique files.
       0 files ignored.

http://cloc.sourceforge.net v 1.60  T=0.06 s (350.7 files/s, 44744.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               17            299            308           1786
C/C++ Header                     3             31             74             54
-------------------------------------------------------------------------------
SUM:                            20            330            382           1840
-------------------------------------------------------------------------------
```

There's a lot of half-assed stuff in there, and a lot of duplication, and two separate fonts, one of which has only 16 characters, designed with `glyphed` in about as many minutes, copied and pasted into tetris.c and xshmucalc.c. Glyphed, which is the only thing that draws clickable things so far, would really benefit from some kind of

IMGUI framework.

The two backends have a bunch of duplicated code, namely all the `xshmu_subpic` (buggy clipping!), `xshmu_copy` (a standard-C-compliance bug fixed in xshmu.c), and `xshmu_canvas` stuff, which is kind of a layer that belongs underneath. On top of this, the `fill` function from Tetris (copied into glyphed) and the `show` function from Tetris (copied into xshmucalc.c and extended) should probably be shared in some place, and probably `show` should take a font argument, and also we need some fonts.

The alpha-blending stuff in wercaμ needs to integrate the SSE optimizations from vecalpha.c, and probably also needs to be in some kind of common location so that, e.g., oscope.c can use it. (Or maybe oscope.c should render in a different way, for example, running three box filters over a monochrome 16-bit framebuffer and mapping the resulting intensities through a palette.)

`xshmu_fd` has to go; `xshmu_fb` either needs to have separate input file descriptors for /dev/input/mice and the keyboard, or it needs a subprocess to re-encode those onto a single stream, so it won't have just a single file descriptor. Maybe `yeso_get_fds(void (*f)(void *, int), void*)`. Some of the use of `xshmu_fd` can be taken over by an added timeout parameter to `xshmu_wait`, which would simplify Tetris significantly.

Key repeat is potentially a big problem for Tetris. X11 introduces spurious key-release events into the input stream for key repeat.

(Incidentally, the mouse thing is somewhat documented in /usr/share/doc/linux-doc/input/input.txt.gz, and apparently it's speaking a kernel-emulated PS/2 protocol, which I guess is three bytes per packet; looks like the second and third bytes are delta-X and delta-Y as signed bytes, with Y increasing up, except that there's a 9th sign bit in the 0x20 (y) and 0x10 (x) position in the first byte. The first byte low nibble is 0x08, except with bit 0x01 set for the left button, bit 0x02 set for the right button, bit 0x04 set for the middle button; dev3/psmouse.c has a PS/2 driver. Unfortunately the mouse wheel doesn't register at all! For the mouse wheel I thought you need /dev/input/event5 or whatever, the evdev interface described in input.txt.gz and event-codes.txt.gz and /usr/include/linux/input.h (dev3/evdev.c has a somewhat more limited decoder for that protocol) but apparently you can somehow set the protocol to ImPS/2 for the wheel. My keyboard is on /dev/input/event3, with scancodes (e.g. 30 for a, 31 for s, 32 for d); key repeat shows up, including on things like control keys, but is distinguishable from repeated keypresses; type=1 (`EV_KEY`) code=30 value=1 is a press of 'a', value=2 is a repeat, value=0 is a release; the scan codes are defined in /usr/include/linux/input-event-codes.h and come originally from USB HUT apparently. There are also type=4 (`EV_MSC`) code=4 (`MSC_SCAN`) value=30, but for some keys the value doesn't match. A great benefit of this interface is that it isn't modal; an app that opens this interface doesn't need to reset the keyboard mode to normal before it exits.)

Some kind of windowing/terminal system would be super keen. Also copy and paste with mouse-selection support are needed to make the terminal emulator usable.

A notebook-style shell window manager interface could be super interesting, embedding graphical programs and saving their graphical

output — by default just the last frame, with hotkeys to save screenshots and start full recording — as well as their textual input/output. It could also limit their CPU use, memory use, and filesystem access, and checkpoint their state for revivification. For windowing, you could have a hotkey to undock the graphical program from its window within the shell.

(Also, I should totally have an animated GIF backend.)

The size of the state to checkpoint should be manageable for many apps; the oscilloscope app on the framebuffer has a 135KB heap segment, a 139KB stack segment, its 8-megabyte framebuffer and backing store, and a couple of read-write segments from shared libraries:

```
 8192 00400000-00402000 r-xp 00000000 fc:01 467163                        /ho
me/user/dev3/oscope_fb

 4096 00602000-00603000 r--p 00002000 fc:01 467163                        /ho
me/user/dev3/oscope_fb

 4096 00603000-00604000 rw-p 00003000 fc:01 467163                        /ho
me/user/dev3/oscope_fb

 135168 00889000-008aa000 rw-p 00000000 00:00 0                           [
heap]
 8298496 7f018256c000-7f0182d56000 rw-p 00000000 00:00 0

 8294400 7f0182d56000-7f018353f000 rw-s 00000000 00:06 399                
/dev/fb0

 1835008 7f018353f000-7f01836ff000 r-xp 00000000 fc:01 1179699            
/lib/x86_64-linux-gnu/libc-2.23.so

 2097152 7f01836ff000-7f01838ff000 ---p 001c0000 fc:01 1179699            
/lib/x86_64-linux-gnu/libc-2.23.so

 16384 7f01838ff000-7f0183903000 r--p 001c0000 fc:01 1179699              /l
ib/x86_64-linux-gnu/libc-2.23.so

 8192 7f0183903000-7f0183905000 rw-p 001c4000 fc:01 1179699               /li
b/x86_64-linux-gnu/libc-2.23.so
 16384 7f0183905000-7f0183909000 rw-p 00000000 00:00 0

 155648 7f0183909000-7f018392f000 r-xp 00000000 fc:01 1179696             /
lib/x86_64-linux-gnu/ld-2.23.so
 12288 7f0183ade000-7f0183ae1000 rw-p 00000000 00:00 0

 4096 7f0183b2e000-7f0183b2f000 r--p 00025000 fc:01 1179696               /li
b/x86_64-linux-gnu/ld-2.23.so

 4096 7f0183b2f000-7f0183b30000 rw-p 00026000 fc:01 1179696               /li
b/x86_64-linux-gnu/ld-2.23.so
 4096 7f0183b30000-7f0183b31000 rw-p 00000000 00:00 0

 139264 7ffd0242b000-7ffd0244d000 rw-p 00000000 00:00 0                   [
```

```
ostack]

 8192 7ffd024ba000-7ffd024bc000 r--p 00000000 00:00 0                      [vvo
oar]

 8192 7ffd024bc000-7ffd024be000 r-xp 00000000 00:00 0                      [vdo
oso]

 4096 ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0              [vso
oyscall]
```

That's generated as follows from /proc/$pid/maps:

```
import re
se = re.compile(r'([0-9a-f]+)-([0-9a-f]+)')
for line in maps.splitlines():
  mo = se.search(line)
  if not mo: print(line); continue
  length = int(mo.group(2), 16) - int(mo.group(1), 16)
  print length, line
```

You don't have to checkpoint the framebuffer contents, since they'll be redrawn anyway after restart. And oscope, as it turns out, is already prepared to restart its child process if it dies, because its child process does die, due I think to bugs in the arecord code. So in theory, with a different checkpoint-aware yeso backend, you could checkpoint just its quarter-meg or so of live data. (As it happens, most of that data is garbage too, but that's hard to know.)

ASLR might complicate restarting from checkpoints, but that's in part because we're still programming at the C level; a virtual machine could simplify this enormously.

In other directions: I really want to try writing some LuaJIT code on it, and I want to write Hypothesis tests for rpncalc.

It's somewhat dismaying to have so many applications for xshmu written in C, since, although I would like to keep it pleasant to program in C, I mostly want to use it to bootstrap out of the C ecosystem and into an archival-virtual-machine ecosystem. At this point I have 13 applications written:

- RPN decimal calculator (308 lines of C, not counting the 28 lines in rpncalc_linux.c for tty output)
- PNG viewer (71 lines of C, written after the above count)
- Tetris (244 lines of C, though some of that needs to be factored out)

- glyphEd fatbits bitmap editor (52 lines of C, including some copy-pasted from Tetris)
- admu ADM-3A emulator (339 lines of C, including offline-mode and on a pty)
- Chifir emulator (110 lines of C)
- Audio oscilloscope (36 lines of C)
- Wercaμ graphics hack (166 lines of C)
- Munching squares graphics hack (17 lines of C)
- Hello, world (7 lines of C)
- μpaint (21 lines of C)

• (the nameless graphics demo inside xshmu.c itself)
• The image-slicing font-making program, also written since the above (161 lines of C)

This works out to (+ 308 71 244 52 339 110 36 166 17 7 21 161) = 1532 lines of C for applications which will need to be rewritten in whatever other language I end up using. This contrasts with the 41 lines of xshmu.h, the 312 lines of X11 backend, and the 201 lines of the fbcon backend (somewhat duplicated with the X11 backend).

So I just wrote IMGUI programming language (p. 103) about what kind of programming language I would like. So far it's focusing on very-low-level stuff (reducing code and bugs in the small, not in the large) and it also describes a language with, to me, an intimidatingly difficult implementation. I think I should see if I can simplify the design to a minimum, maybe something at the C level or even a Lisp, and then see what I can add to it.

The simplest possible thing would of course be a Forth, but writing the compiler for it is a pain. Still, a Forth-level "portable assembler" that does simple register allocation would simplify later work by a lot. (At some point I want to extend that into a deterministic vector virtual machine for software archival and deterministic recomputation.) It wouldn't need to support macros, since its intended use is as a backend for higher-level languages. I do want to expose the stack implementation sufficiently to enable static stack-depth bounds for applications where that's desirable.

I had thought, after reading Finkel's book section about CLU iterators, that doing them required allocating the iterator activation record with enough slack space on the stack for whatever functions the yielded-to block might call — that is, that those functions would put their activation records in between the function invoking the iterator and the iterator itself. But that isn't true; the block can push the arguments for those functions on top of the iterator state.

Vaguely related:
https://news.ycombinator.com/item?id=18765868 is a Sixel image listing program for `xterm -ti vt340`.

Topics

• C (p. 3359) (28 notes)
• Python (p. 3671) (27 notes)
• BubbleOS (p. 3352) (17 notes)
• Yeso

Some speculative thoughts on DSP algorithms

Kragen Javier Sitaker, 2014-04-24 (20 minutes)

I was reading a book on digital signal processing (the highly readable if occasionally inaccurate and somewhat outdated *Guide to Digital Signal Processing for Scientists and Engineers* http://www.dspguide.com/), and it seemed to me that the available types of digital filters leave a lot to be desired.

FIR filters are the most popular because you can design them by Fourier-transforming your desired frequency response to get an impulse response, then windowing the impulse response to get a finite-size convolution kernel. Windowing with the popular Blackman or Hamming windows only screws up your frequency response slightly. The trouble is that your convolution kernel may be dozens to hundreds of samples wide, so you end up doing dozens to hundreds of multiply-accumulates if you compute the convolution in the time domain; and to do it in the frequency domain, you need to do the FFT, which also involves typically dozens of multiply-accumulates per sample.

IIR filters are technically a superset of FIR filters, but people usually design them by transforming well-known analog filters (like the classic Butterworth or Bessel filters) into the discrete domain, which is very limiting. Additionally, the typical design technique using the Z-transform requires that the filter itself be linear and stateless, with each output sample being a linear combination of the P previous input samples and the Q previous output samples.

In passing, the moving-average filter was mentioned as being much faster to compute, since it requires only an addition and a subtraction per sample, optimal for suppressing noise while preserving edge sharpness, and producing a good approximation of convolution with a Gaussian when iterated.

So some ideas occurred to me, which might or might not be good ones, but I thought I'd write them down anyway.

Square-wave filters

The discrete Fourier transform transforms your given samples onto a basis space of orthonormal sinusoids, enabling you to, among other things, measure the amplitude of individual frequencies, or multiply them by a desired frequency response curve before transforming them back into the time domain with an inverse discrete Fourier transform (a filter).

But many other sets of orthonormal basis functions are possible: the Hadamard–Walsh functions, unit impulses (which yields the identity transform), an infinite variety of wavelets, windowed sinusoids (the short-time Fourier transform), the Gabor basis function (the Gabor transform happens to be a special case of the STFT and also, I believe, the wavelet transform), and so on.

The Hadamard–Walsh functions are particularly interesting because there's an O(N log N) algorithm to compute the Hadamard or Walsh transform, using only addition and subtraction, because the

basis functions have the range {-1, 1}, with no fractions, and are related to each other in a particular way that enables the fast Hadamard transform to work.

But most of the Hadamard–Walsh functions are not particularly close to being sine waves, so they are of limited usefulness if you want to filter particular frequencies.

On the other hand, if you have a square wave of some frequency f, it's pretty strongly correlated with a sine wave of frequency f with the right phase, and it's perfectly uncorrelated with most other sine waves. However, it does have a largish correlation with the odd harmonics of the original sine wave, with frequencies 3f, 5f, 7f, and so on; its correlations with them are 1/3, 1/5, 1/7, etc., of the original. (So far this is just the standard Fourier analysis of a square wave, seen from the perspective of the square wave.)

This same property means that a comprehensive set of square waves is not an orthogonal basis, since some of its elements are correlated, if imperfectly, and thus not orthogonal. This, in turn, means that you can't simply transform a signal into a weighted sum of square waves by correlating it (i.e. taking the dot product) with each square wave.

But suppose you just want to compute the energy in a given time span at a given frequency f? You could take the correlation with a square wave sq(f, t) of frequency f and subtract off the other square waves until you've approximated a sine to your sampling interval: sq(3f, t)/3 + sq(5f, t)/5 + sq(7f, t)/7, etc. If you can low-pass filter the signal before you do this, even crudely (say, with a moving-average or simple exponential filter), then you can probably quit pretty early.

(You probably want to do this twice, once for sq(f, t) and its "harmonics", and once for sq(f, t + 1/2f) and its "harmonics", so you can catch a wave that's out of phase.)

Why should you care when the DFT is already O(N log N) multiply-accumulates? Because you can do this faster than O(N log N), without multipliers, and without using memory to store the samples, if you don't want too many frequencies.

If you only want a *single* square wave correlation, of course, you can simply add or subtract each sample to the total as it comes in, according to whether sq(f, t) is 1 or -1 at that moment. But doing that for M square waves means doing M additions or subtractions per sample. Instead, use a sum table, also known as a summed-area table or integral image: s[t] = sum(x[0:t]), where 0:t includes 0 but not t, so sum(x[t0:t1]) = s[t1] - s[t0]. So if your square wave is 1 from sample t0 to sample t1, you can add s[t1] - s[t0] to your running sum; and if it's -1 from sample t1 to sample t2, you can add s[t1] - s[t2]. You might as well add 2s[t1] on the negative-going transition at t1 in the first place, and later subtract 2s[t2] at the positive-going transition at t2, and so on. And you don't need to actually do the doubling at the time; you can wait until you're inspecting the final sum before remembering that you need to double it.

This, of course, doesn't require that you actually store the sum table, just that you compute the values in it. This requires one addition per sample, plus one addition or subtraction per square-wave transition.

If you decide to do this calculation for all the N square waves that would correspond to the sinusoids needed for N samples, the waves

will have numbers of transitions ranging linearly from 0 to N, with an average of about N/2, so you'd end up doing N/2 additions or subtractions per sample --- definitely worse than Hadamard–Walsh (as long as N/2 > lg N, which is true for N>4) and possibly worse than the DFT, depending on how much multiplies cost you. But maybe you need less than lg N square waves, or maybe the ones you want are of lower-than-average frequency. Most of the high-frequency square waves here will have substantial phase noise induced by quantization which will degrade their performance anyway.

You might be able to avoid doing some of this work by taking advantage of the fact that it's duplicate work in the case of harmonics. Consider the square wave with period 30, which means it has two transitions every 30 samples. The square wave with period 10 has six transitions every 30 samples, but two of them are the same as the period-30 one; that is, you could compute both the period-10 and period-30 square waves with only 6 transitions, rather than 8. Similarly, the period-6 square wave has 10 transitions in this period, but shares the two of the period-30 wave, so you could compute all three with only 14 transitions, rather than 18. I'm not sure how practical this is; it reminds me of the wheel optimization for probing possible composites in the Sieve of Eratosthenes.

A difficulty is that, of course, the difference between square waves and sine waves has more energy below the Nyquist frequency, or any given filter cutoff frequency, for lower-frequency waves. But perhaps you can do better. The procedure I described above for correlating with the square waves is equivalent to integrating the input signal, once, and correlating it with a train of alternating positive and negative impulses, the derivative of the square wave. The integral of a square wave is a triangle wave, which is much closer to being a sine wave, and the integral of a triangle wave is a wave made of parabolas, which is a piecewise second-order approximation to a sine wave, and actually has almost all of its energy in that frequency; and you can iterate the procedure N times to get an Nth-order approximation of a sine wave.

So perhaps you could integrate the input signal N times, making an Nth-order sum table (materialized or virtual), and correlate that with your square waves, or rather, your impulse trains. One potential difficulty is that this corresponds to a pretty heavy-duty low-pass filter, and so it will be necessary to introduce a corresponding high-pass filter at each stage, if nothing else to prevent any initial DC offset, or any introduced later as a constant of integration, from causing the procedure to diverge entirely. A difference in frequency of one octave will show up as a difference in magnitude of 2 after one stage of integration, of 4 after two stages, of 8 after three stages, and of 16 after four stages. Of course, in a sense, that's exactly what we *want* it to do; but roundoff error could be a problem very quickly.

Magic sinewave filters

Don Lancaster has been writing about a set of functions he calls "magic sinewaves", which are periodic functions with the range {-1, 0, 1} that approximate sinewaves, with the purpose of improving power electronics by replacing simple PWM with lower-distortion waveforms. Of course, they have very substantial power outside the

desired frequency --- the same as a PWM waveform, I think --- but the idea is that if you push that power to a high enough frequency, it's much more practical to filter it, using an analog filter with a slower rolloff and physically smaller components.

You can also use this idea in reverse: take your input signal and correlate it with a magic sinewave, rather than a real sinewave, thus avoiding multiplication entirely. For accurate results, you need to prefilter the signal to eliminate the high harmonics, but you can do this with a simple, easy-to-compute filter, such as a moving-average filter.

Parallelizing IIR filters

Parallelizing an M-sample FIR filter is easy: split your input into windows that overlap by M samples, filter each one independently, and concatenate the results. But how can you parallelize an IIR filter?

Consider the sum table, which is a simple IIR filter; it amounts to convolving a step function with your input. It's simple and efficient to calculate a sum table serially, but how could you parallelize it, since the last output value depends on every value before it?

I think I've written about this before on kragen-tol, but the answer is basically that it's easy, because addition is a monoid; this is the approach taken by lookahead carry in digital logic, too. If you break your input into four equal segments 0:t1, t1:t2, t2:t3, and t3:n, and compute the sum table so[a:b] for each segment independently (in parallel), you can then use the final sum of each segment to adjust the sums of the other segments, again independently and in parallel: s[0:t1] = so[0:t1], while s[t1:t2] = s[t1] + so[t1:t2], s[t2:t3] = s[t2] + so[t2:t3], and s[t3:n] = s[t3] + so[t3:n]. You can apply this division of the process recursively (or with a branching factor of other than 4, although a branching factor of more than √n is unlikely to be an improvement) to get an O(log N) time algorithm.

(This is the well-known "parallel prefix sum" problem.)

General IIR filters are impossible to parallelize. But the IIR filters in common usage are purely linear and dependent on a limited amount of past history, so you can do the same thing: filter each segment independently, then add in the linear contribution from the prefix to its left.

Factoring FIR kernels approximately into sparse FIR kernels

Convolution with a Gaussian is potentially computationally expensive. But N iterations of a moving-average filter give you a piecewise Nth-order approximation of a Gaussian, and the moving-average filter is really cheap to compute.

In general the expense of convolving with a FIR kernel in the discrete time domain is expensive in proportion to its *support*, that is, the number of points where it's nonzero. But what if you could factor a 256-point FIR kernel into a convolution of two 16-point FIR kernels? You could compute it four times faster. That's doable in at least some special cases; the Gaussian mentioned above is an example (although the moving average has a recursive algorithm that's more efficient than just doing it as a FIR convolution in the time domain),

since actually iterating just about any FIR kernel enough times will give you a Gaussian (e.g. [1, 1], whose iterative convolution generates the rows of Pascal's triangle; but I suspect that [1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1] might be a better compute/quality tradeoff for larger Gaussians). But consider these examples:

• Exact factorization of a finite comb: a comb of N equally-spaced impulses can be factored into one P-impulse kernel for each prime factor P of N. For example, a 16-impulse comb can be factored into four two-impulse kernels, with the impulses at distances 1, 2, 4, and 8 times the spacing of the impulses in the desired comb; or a 20-impulse comb can be factored into two two-impulse combs and a five-impulse comb. (In the special case of a composition of two two-impulse combs, it's probably advantageous to preconvolve them into a single four-impulse comb. The optimum number of impulses per kernel, to keep multiply-accumulates to a minimum, is 3.)
• Exact factorization of exponential decay: for N points, compose log(N) two-point FIR kernels whose points are spaced at powers of 2, whose first point is 1 and whose other point is exponentially decayed from 1 according to its spacing. This happens to be exact. This is sort of stupid, though, because you can compute exponential decay even more efficiently with an IIR filter.
• Approximation with a sum of equal-width Gaussians: use one sparse kernel with an impulse indicating the center and amplitude of each Gaussian, then one or more FIR kernels, or the moving-average technique, to convolve the result with the desired Gaussian.
• Exact factorization of a periodic signal: factor into a kernel representing a single period and a comb kernel (itself perhaps factored as above) with one impulse for each period, in order to copy each period into its desired place. If the period is itself a repetition of the same signal twice, but once inverted, you can use an additional kernel with a 1 and a -1 impulse to halve the number of points. Composed with a Gaussian window, this should give you a fairly efficient way to do FIR bandpass filters.
• Approximate Gaussians --- although this is not the right way to do it, convolve(convolve(convolve(convolve([1, 1,1], [1, 1, 1]), [1, 0, 1,0, 1]), [1, 0, 1, 0, 1]), [1, 0, 0, 1]) is quite close to convolve(convolve(convolve([1,1,1,1,1], [1,1,1,1,1]), [1,1,1,1,1]), [1,1,1,1,1]), requiring 14 additions rather than 20. (The recursive algorithm using a fourth-order moving average, though, requires only 8.)
• The well-known "convolution by separability" technique used in image processing, in which you factor a separable kernel into a vertical component and a vertical component, is a special case.

Is there a general technique to find such an approximate factorization into sparse kernels for any desired impulse response? Here's one guess: tabulate the frequency responses for some "basis set" of sparse kernels, then use a greedy algorithm to iteratively pick the ones that best suppress the largest difference between the frequency response of your current set of filters and the frequency response you want.

There's a paper this year from Aimin Jiang and Hon Keung Kwan on the subject that I haven't read, using weighted least squares (WLS); but it sounds from the abstract like they're talking about

approximating one FIR filter kernel with another, sparser one, not factoring one into possibly several.

How small do these factors need to be? The competition is FFT convolution, which, if I understand correctly requires computing a DFT, pointwise complex multiplication, and computing an inverse DFT. Each DFT requires N lg N butterflies, each of which requires one complex multiplication and two complex additions or subtractions. One complex multiplication requires four real multiplications. So we have 2 × 4 × N lg N real multiplications, or 8 lg N real multiplications per point, plus some other overhead work which is mostly proportional. FFT convolution isn't worthwhile unless lg N is at least 6, at which point you're paying 48 real multiplications per point (which is why it isn't useful for smaller N); if lg N is 7, 8, 9, or 10, you're paying 56, 64, 72, or 80 real multiplications per point.

Probably close to the best you can do for a filter kernel with N-point support is 3 $\log_3 N$ multiply-accumulates per point, which is only slightly lower than 2 lg N, although in some cases maybe you can get by without the multiplications. 2 lg N is substantially less than 8 lg N, and furthermore the N you use in the FFT is going to be around two or three times the size of the kernel, so there's some hope that this algorithm could maybe do a reasonable job

IIR: the undiscovered country

Basically we know that IIR filters can always be more computationally efficient than FIR filters, sometimes dramatically; but we don't have a good theory of how to design them. There are probably a lot of tasty morsels hiding in IIR-space.

Topics

- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Convolution (p. 3391) (15 notes)
- Sparse filters (p. 3725) (11 notes)

What's the dumbest register allocator that might give you reasonable performance?

Kragen Javier Sitaker, 2016-10-11 (15 minutes)

Programs compiled with Ur-Scheme are about five times slower than similar programs compiled with GCC. TCC is broadly similar.

It occurred to me that a big part of Ur-Scheme's slowness is probably its complete lack of register allocation. It treats the i386 as a stack machine with the top of stack in %eax. And so it occurred to me that even the dumbest register allocator would probably provide a big speedup, maybe up to a slowdown of less than 2.

What's the dumbest register allocator that might give you reasonable performance? Well, on amd64, as I understand it, you have 16 general-purpose registers, but one of them is the stack pointer. You could statically allocate 8 of these registers to local variables (the first 8 local variables) and 7 of them to an expression evaluation stack which is empty in between statements, assuming you have statements and that control flow is a per-statement thing. Then you can allocate the slots on the expression evaluation stack at compile time and spill to memory if your evaluation gets too deep.

I feel like that would probably work well enough, and it would be very simple to implement.

On i386, you don't have as many registers. You could have 4 levels of expression evaluation and 3 local variables held in registers.

Context: how well do programs get optimized?

Here's a Karplus-Strong string synthesis program I wrote in C, something over a year ago, in golfed form:

```
enum{n=72};s[n]={1<<15},i;main(){void*o=popen("aplay","w");
for(i=0;i<1<<14;i++)fputc((s[i%n]=s[i%n]*.9+s[(i+1)%n]*.1)>>8,o);}
```

Here's a deobfuscated version of the main loop:

```
for (i=0; i < 1<<14; i++) {
    int j = i % n, k = (i+1) % n;
    s[j] = s[j]*.9 + s[k]*.1;
    fputc(s[j] >> 8, output);
}
```

`n` here is 72, and `s` starts out with just an impulse in it. (If you're not familiar with KS synthesis, the weighted averaging of two samples there serves as a gentle low-pass IIR filter, attenuating all frequencies but especially the highest frequencies, while feeding the signal back into the delay line after 72 samples limits the original harmonic content to harmonics of 8000 Hz / 72 = 111 Hz, and the result sounds exactly like the string of a six-string guitar.)

I feel like this is a relatively typical piece of C code in some ways.

It has a for loop comparing against a constant expression that can be folded, some array accesses and a function call inside the loop, some floating point and integer math, some strength-reduction opportunities, and a local variable (or two, although `i` is technically a global).

So how badly does TCC do on this program?

First, what's our gold standard? It's actually pretty terrifying, both in terms of the optimizations achieved and the optimizations missed. On i386, here's a commented disassembly of the 48 instructions gcc 4.8.4 emits for the main loop with -O:

```
 804847d:   89 c6                   mov    %eax,%esi
 # This is the address of i; we are initializing it with 0.
 804847f:   c7 05 64 a1 04 08 00    movl   $0x0,0x804a164
 8048486:   00 00 00
 # And we also have i in %edi at the top of the loop.
 8048489:   bf 00 00 00 00          mov    $0x0,%edi
 # This magic number is the approximate multiplicative inverse of
 # 72.  72 times this number is 0x10,0000,0008.  That means that if
 # you multiply some 32-bit integer X by 0x38e38e39, then the high
 # word of the 64-bit result will contain the number divided by 72,
 # shifted four bits to the left.  So, for example, 0x38e38e39L *
 # 146 >> 36 = 2L.  So we save it in %ebx so we can multiply by it
 # in order to divide.  Multiplying is usually much faster than
 # division.
 804848e:   bb 39 8e e3 38          mov    $0x38e38e39,%ebx
mainloop:
 8048493:   89 f8                   mov    %edi,%eax
 # Now we divide %eax by 72 by multiplying it by the magic number in
 # %ebx.  %eax is the implicit argument of a one-argument imul in
 # i386.  I guess this means i was in %edi and %eax.
 8048495:   f7 eb                   imul   %ebx
 # The high dword of the result is in %edx, so put it in %ecx.
 8048497:   89 d1                   mov    %edx,%ecx
 # Eliminate those 4 extra bits on its low end so %ecx contains %eax/72.
 8048499:   c1 f9 04                sar    $0x4,%ecx
 # Another copy of i goes into %eax.
 804849c:   89 f8                   mov    %edi,%eax
 # Now we shift it 31 bits to the right, which means we are saving
 # off just its sign bit, which is a little bit stupid because it's
 # statically fairly trivial that it's always positive.
 804849e:   c1 f8 1f                sar    $0x1f,%eax
 # Now we do some kind of negative-number correction to our quotient
 # in %ecx with this.  This amounts to a no-op.
 80484a1:   29 c1                   sub    %eax,%ecx
 # Now we magically multiply %ecx (i/72) by 9 and put the result in
 # %eax.
 80484a3:   8d 04 c9                lea    (%ecx,%ecx,8),%eax
 # Then we shift it three more fucking bits, so guess what, now
 # we've multiplied it by fucking 72.  GCC factored 72 into 8*(8+1)
 # in order to multiply it with two obscure instructions instead of
 # just using an imul.  Now %eax is going to contain (i/72)*72,
 # which is to say, i-(i%n).
 80484a6:   c1 e0 03                shl    $0x3,%eax
 # Now we get another copy of i into %ecx.
```

```
80484a9:   89 f9                   mov    %edi,%ecx
# And now at last we compute i%n in %ecx, which took 11
# instructions.  This is the number I called "j" above.
80484ab:   29 c1                   sub    %eax,%ecx
# Now we load the integer at s[j] (s[i%n]) as a floating-point
# number.
80484ad:   db 04 8d 40 a0 04 08    fildl  0x804a040(,%ecx,4)
# This isn't in the listing but I bet the constant 0.9 is stored
# there.
80484b4:   dc 0d d8 85 04 08       fmull  0x80485d8
# Now we increment i, which makes a certain amount of sense because
# now we have to compute s[k], and we aren't going to use i again.
80484ba:   83 c7 01                add    $0x1,%edi
# Oh boy, here we go again.  Compute k as (new i) % n in another 11
# instructions, except now it's in %edi.
80484bd:   89 f8                   mov    %edi,%eax
80484bf:   f7 eb                   imul   %ebx
80484c1:   c1 fa 04                sar    $0x4,%edx
80484c4:   89 f8                   mov    %edi,%eax
80484c6:   c1 f8 1f                sar    $0x1f,%eax
80484c9:   29 c2                   sub    %eax,%edx
80484cb:   8d 04 d2                lea    (%edx,%edx,8),%eax
80484ce:   c1 e0 03                shl    $0x3,%eax
80484d1:   29 c7                   sub    %eax,%edi
# Compute the weighted sum, multiplying by a different constant in
# memory, presumably 0.1.
80484d3:   db 04 bd 40 a0 04 08    fildl  0x804a040(,%edi,4)
80484da:   dc 0d e0 85 04 08       fmull  0x80485e0
80484e0:   de c1                   faddp  %st,%st(1)
# Truncate the result back to an int by way of storing it in the
# stack frame and loading it into %eax.
80484e2:   d9 7c 24 0e             fnstcw 0xe(%esp)
80484e6:   0f b7 44 24 0e          movzwl 0xe(%esp),%eax
# Okay, I have no idea what is going on here, except that it
# results in storing the new value of s[j] at 0x8(%esp) and also in
# %eax.  Somehow this involves overwriting the high byte of
# some 16-bit number with 0xc!?
80484eb:   b4 0c                   mov    $0xc,%ah
80484ed:   66 89 44 24 0c          mov    %ax,0xc(%esp)
80484f2:   d9 6c 24 0c             fldcw  0xc(%esp)
80484f6:   db 5c 24 08             fistpl 0x8(%esp)
80484fa:   d9 6c 24 0e             fldcw  0xe(%esp)
80484fe:   8b 44 24 08             mov    0x8(%esp),%eax
# Now we store the result into s[j]; %ecx still has j in it, and
# 0x804a040 is the base address of s, as indicated by its use above
# in the fildls.
8048502:   89 04 8d 40 a0 04 08    mov    %eax,0x804a040(,%ecx,4)
8048509:   89 74 24 04             mov    %esi,0x4(%esp)
# Here we shift the s[j] result in %eax right by 8 bits and "push
# it" in order to call fputc with it.
804850d:   c1 f8 08                sar    $0x8,%eax
8048510:   89 04 24                mov    %eax,(%esp)
8048513:   e8 38 fe ff ff          call   8048350 <fputc@plt>
# Now we ignore fputc's return value and load the value of i again
# from memory and increment it, again.  fputc could have changed
```

```
 # it, after all, since it's a global variable.
 8048518:   a1 64 a1 04 08          mov    0x804a164,%eax
 804851d:   8d 78 01                lea    0x1(%eax),%edi
 # Save the incremented value to memory.  You never know, fputc
 # could be reading a global variable.
 8048520:   89 3d 64 a1 04 08       mov    %edi,0x804a164
 # Check to see if the loop is done.  It's interesting that the <
 # test got turned into a <= test.
 8048526:   81 ff ff 3f 00 00       cmp    $0x3fff,%edi
 804852c:   0f 8e 61 ff ff ff       jle    8048493 <main+0x36> (mainloop)
```

All right, so we can see GCC missed a *lot* of optimization opportunities there, although some of them were missed by the perverse choice to make i a global variable. But it did manage to compute i%72 without using any division instructions. It also manages to keep j and k in registers and avoid recomputing j --- common subexpression elimination. And the control flow is nice and clean, just a single conditional jump instruction at the end, because it can statically check that the loop will execute at least once.

How does TCC do? It's a bit messier but somewhat more straightforward, as you'd expect. tcc 0.9.25 emits 67 instructions:

```
# Initialize i to 0.  Note the unnecessary indirection through a
# register that isn't used afterwards.
     242:   b8 00 00 00 00          mov    $0x0,%eax
     247:   89 05 00 9f 04 08       mov    %eax,0x8049f00
looptest:
# Load i from memory and check it.  This is at the top of the loop
# so that it can run zero times if necessary.
     24d:   8b 05 00 9f 04 08       mov    0x8049f00,%eax
     253:   81 f8 00 40 00 00       cmp    $0x4000,%eax
     259:   0f 8d c8 00 00 00       jge    0x327 (exitloop)
     25f:   e9 11 00 00 00          jmp    0x275 (loopbody)
mainloop:
# This is "i++".  Neither %eax nor %ecx is used afterwards.  I have
# no idea why it makes a copy in %ecx.
     264:   8b 05 00 9f 04 08       mov    0x8049f00,%eax
     26a:   89 c1                   mov    %eax,%ecx
     26c:   40                      inc    %eax
     26d:   89 05 00 9f 04 08       mov    %eax,0x8049f00
     273:   eb d8                   jmp    0x24d (looptest)
loopbody:
# Load i from memory.
     275:   8b 05 00 9f 04 08       mov    0x8049f00,%eax
     27b:   b9 48 00 00 00          mov    $0x48,%ecx   # 0x48 == 72 == n
     280:   99                      cltd                # clears %edx?
# implicitly divides %edx:%eax by its operand; remainder is in %edx,
# quotient in %eax.
     281:   f7 f9                   idiv   %ecx
# Shift the quotient left two bits to use it to index dwords.
     283:   c1 e2 02                shl    $0x2,%edx
# Load base address of s into %eax and index it.
     286:   b8 44 9d 04 08          mov    $0x8049d44,%eax
     28b:   01 d0                   add    %edx,%eax
# Save the pointer into a temporary in the stack frame.  This is
```

```
# where we are going to store the result of the weighted sum.
    28d:   89 45 f8              mov    %eax,-0x8(%ebp)
# Load i from memory again.  This is the common subexpression s[i%n]
# not being eliminated.
    290:   8b 05 00 9f 04 08     mov    0x8049f00,%eax
    296:   b9 48 00 00 00        mov    $0x48,%ecx
    29b:   99                    cltd
    29c:   f7 f9                 idiv   %ecx
    29e:   c1 e2 02              shl    $0x2,%edx
    2a1:   b8 44 9d 04 08        mov    $0x8049d44,%eax
    2a6:   01 d0                 add    %edx,%eax
# Load from s[i%n] into %ecx, then into a floating-point register by
# way of the stack, since that's apparently how we do things.
    2a8:   8b 08                 mov    (%eax),%ecx
    2aa:   51                    push   %ecx
    2ab:   db 04 24              fildl  (%esp)
# Pop it back off.
    2ae:   83 c4 04              add    $0x4,%esp
# Store the floating-point version of s[j] in an on-stack temporary?
    2b1:   dd 55 f0              fstl   -0x10(%ebp)
# Not sure what's going on here.
    2b4:   dd d8                 fstp   %st(0)
    2b6:   dd 05 6c 9e 04 08     fldl   0x8049e6c
    2bc:   dc 4d f0              fmull  -0x10(%ebp)
# Okay, now we load i from memory again, increment it, and divide it
# to compute (i+1)%n.
    2bf:   8b 05 00 9f 04 08     mov    0x8049f00,%eax
    2c5:   40                    inc    %eax
    2c6:   b9 48 00 00 00        mov    $0x48,%ecx
    2cb:   99                    cltd
    2cc:   f7 f9                 idiv   %ecx
# And then again we index off s.
    2ce:   c1 e2 02              shl    $0x2,%edx
    2d1:   b8 44 9d 04 08        mov    $0x8049d44,%eax
    2d6:   01 d0                 add    %edx,%eax
# Wait, what are we doing with the FPU?
    2d8:   dd 55 e8              fstl   -0x18(%ebp)
    2db:   dd d8                 fstp   %st(0)
# Okay, load s[k]
    2dd:   8b 08                 mov    (%eax),%ecx
    2df:   51                    push   %ecx
    2e0:   db 04 24              fildl  (%esp)
    2e3:   83 c4 04              add    $0x4,%esp
# I don't know what's going on here either.
    2e6:   dd 55 e0              fstl   -0x20(%ebp)  # ???
    2e9:   dd d8                 fstp   %st(0)       # ???
    2eb:   dd 05 74 9e 04 08     fldl   0x8049e74    # Maybe 0.1?
 # But this looks like the weighted sum.
    2f1:   dc 4d e0              fmull  -0x20(%ebp)
    2f4:   dc 45 e8              faddl  -0x18(%ebp)
    2f7:   d9 2d 7c 9e 04 08     fldcw  0x8049e7c  # ???
    2fd:   81 ec 04 00 00 00     sub    $0x4,%esp
 # Store the weighted-sum result at the stack pointer.
    303:   db 1c 24              fistpl (%esp)
    306:   d9 2d 7e 9e 04 08     fldcw  0x8049e7e  # ???
```

```
 # And now load it into %eax.
     30c:   58                      pop    %eax
# Load the pointer where we are going to save the result (stored at
# 0x28d above) into %ecx.
     30d:   8b 4d f8                mov    -0x8(%ebp),%ecx
# Save the result.
     310:   89 01                   mov    %eax,(%ecx)
# Shift the result before emitting it with fputc.  Note that we
# don't load it from memory again.
     312:   c1 f8 08                sar    $0x8,%eax
# Load o into %ecx to push it.
     315:   8b 4d fc                mov    -0x4(%ebp),%ecx
     318:   51                      push   %ecx
     319:   50                      push   %eax
     31a:   e8 f1 09 00 00          call   0xd10     # fputc
     31f:   83 c4 08                add    $0x8,%esp # pop fputc args
     322:   e9 3d ff ff ff          jmp    0x264 (mainloop)
exitloop:
     327:   c9                      leave
     328:   c3                      ret
```

The only real optimization TCC managed here was to keep a value in a register, once, and constant-fold the loop bound. It didn't attempt common subexpression elimination, and its FPU code is relatively horrible.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Small is beautiful (p. 3714) (40 notes)
- Audio (p. 3331) (40 notes)
- Assembly language (p. 3328) (25 notes)
- Compilers (p. 3383) (16 notes)
- Tcc

Low-power microcontrollers for a low-power computer

Kragen Javier Sitaker, 2016-09-06 (updated 2018-10-28) (18 minutes)

(See also Keyboard-powered computers (p. 2220).)

E-ink displays are capable of supporting continuous, comfortable reading at average power levels under 3 milliwatts. For this to be useful, we need a microcontroller to drive them at comparable power. But 5–20 watts might be available sometimes, and we'd like to be able to take advantage of it.

To build a computer that provides rapid UI response on very low average power (<10mW, >100 seconds per joule) I think you need, among other things, multiple processors: a low-speed, low-power "peripheral processor", probably designed as an embedded microcontroller, for normal real-time interaction, plus a higher-speed "central processor" that powers up occasionally, or when lots of power is available, to handle heavier-duty computation.

Computational power needed

Simply slinging pixel data around is a minimum; at 1 bit per pixel and 400 pixels per character (due to high-res screens), you need to send 50 bytes of data to the framebuffer to draw a single letter or space. That's probably about 100 instructions, so 100 instructions/glyph × 350 wpm × 6 glyphs / word = 3500 instructions/second. But that's like a theoretical minimum.

WordStar on a 2MHz (≈0.5MIPS) 8-bit CPU would sometimes fall behind your typing a bit. This suggests that one (8-bit) MIPS is adequate to provide real-time word processing.

A nanojoule (1000 pJ) per instruction is a microwatt per MIPS, which would be about 0.56 µA/MIPS (the unit microcontroller vendors like to use) at 1.8V. That's also 1000 giga-ops-per-second per watt. Low-power microcontrollers are usually 2–5 times better than this.

Atmel picoPower

Atmel ARMs can do 250 pJ per 32-bit instruction and 2µW in standby. However, they take forever to wake up, like tens of microseconds (and a microsecond is 12 to 48 instructions!). A full reboot from OFF takes 2.2 milliseconds.

At this energy cost, 10W would pay for 40 billion instructions per second, but that would require nearly a thousand microcontrollers.

These are Cortex-M0+s with single-cycle hardware (32×32!) multipliers. And also a tiny four-cell FPGA called "CCL", "configurable custom logic".

http://www.atmel.com/technologies/lowpower/picopower.aspx makes it hard to tell how low power can go. They say 1.62V.

http://www.atmel.com/Images/doc8349.pdf gives an example of how to lower power consumption by lowering clock frequency from 8MHz to 1.8432MHz, transmitting data at 115,200 baud instead of 19,200 baud, lowering the brownout voltage from the usual 2.7V to

1.8V, and going to sleep, increasing run-time off a capacitor from 6 seconds to 217 seconds. However, they never specify the capacitance!

http://www.atmel.com/products/microcontrollers/arm/sam-l.aspx#o
osaml21 says "power consumption down to 35 µA/MHz in active mode and 200nA in Sleep mode".
http://www.atmel.com/Images/Atmel-42385-SAM-L21_Datasheet.o
opdf says 48MHz (the Performance Level 2 model) or 12MHz (the PL0 model), 256kB flash, 40kB SRAM, 1.62 to 3.63 V, and typically 25 to 100 µA/MHz with the regulator in buck mode, lower for PL0 than PL2. 75 µA/MHz at 3.3V in the PL0 model seems like the worst-case performance. Normalizing that, we get 250pJ/inst. Also gives numbers in the 1 to 10 µA for STANDBY and BACKUP states, depending on temperature, and a bit lower for OFF.

More easily available is the https://www.digikey.com/product-detail/en/ATSAMD10D13A-MO
OUT/ATSAMD10D13A-MUTCT-ND/5226477 Atmel ATSAMD10D13A-MUT (US$2.24); datasheet at http://www.atmel.com/Images/Atmel-42242-SAM-D10_Datasheeto
o.pdf. 8 kB flash, 4 kB RAM, 22 GPIOs. Worst case 75 µA/MHz (+284µA), which normalizes to the same 250pJ/inst. Wakeup time is 4–20µs.

STM32

See also Notes on the STM32 microcontroller family (p. 3176), but the STM32L low-power ARM microcontroller series is supposedly 144 pJ/insn, and can reasonably do duty cycles down to 0.02%, while using 2 mA at 1.8 V at 16 MHz, and the STM32F0 range is 250 pJ/insn.

ATMega328

The familiar ATMega328 used in the Arduino isn't the highest-tech or lowest-power microcontroller out there, but it's very well known, and it's simple to get running. It runs at up to 20MHz, with most instructions in a single cycle, but only 8-bit ALU operations. In power-save mode, with a real-time clock enabled, it uses 0.75µA (at 1.8V, so 1.4 µW); at 8MHz and 5V, it uses up to 12mA, so 60mW, working out to 7500 pJ/instruction. This is about 30 times worse than the AT-SAM family and MSP430 family, even before you take into account that these are 8-bit operations.

Consequently, at 20MHz, this processor will gobble 150mW.

MSP430

The von Neumann MSP430 family is in the range of 300–500 pJ per instruction, 10–25 16-bit MIPS, and 220–4000nW idle, which means instruction consumption exceeds idle consumption above about three instructions per second or one wakeup per minute. At a power budget of 5mW, an MSP430 could average 12 16-bit MIPS.

These processors are generally the processors of choice for very-low-duty-cycle applications.

http://www.ti.com/lit/wp/slay015/slay015.pdf TI in 2012 explaining why they think MSP430 is better at low-power than Microchip XLP.

http://www.greenarraychips.com/home/documents/greg/WP003-100617-msp430.pdf says that in 2010 the F18 consumed 7 pJ per instruction or 450 pJ per 17×17-bit multiply; an MSP430 (at 8MHz) consumed 330 pJ per instruction or 2310 pJ per 16×16 multiply. It also gives a sleep/wakeup time of 4 ns for the F18, with an idle power consumption of 100 nW, and a sleep/wakeup time of 5µs for the MSP430, with an idle (LPM4 sleep state) power consumption of 3600 nW.

http://johann-glaser.blogspot.com.ar/2012/10/msp430-launchpad-with-debian.html says that an MSP430G2231 has 2kB flash, 128 bytes RAM, and runs at 16MHz (performing 16-bit operations). (Also, it gives a nice howto for getting a TI Launchpad device working with Debian.)

http://www.ti.com/tool/msp430-gcc-opensource is TI's deprecated but actively maintained port of GCC.

https://en.wikipedia.org/wiki/TI_MSP430A says the FRAM series of MSP430s has 320 nA RAM retention and 82 µA/MIPS, which is half the power consumption of other MSP430s, with up to 2 kB of RAM. Other MSP430s have up to 512kB of ROM or 10kB of RAM. Normalizing these numbers using 3.3V, which ought to be about right, we get 271 pJ/insn and 1100 nW. The MSP430G2xx series has sub-microsecond wakeup, 0.1µA RAM retention, and 220µA/MIPS at 2.2V, or 480 pJ/insn and 220nW idle.

Experimental subthreshold processor

http://www.eit.lth.se/fileadmin/eit/courses/eit095f/Hanson_Variability_J_2008.pfd describes an 8-bit processor that works at 350 mV and uses 3.5 pJ/instruction, two orders of magnitude less than the MSP430.
https://web.eecs.umich.edu/~taustin/papers/VLSI-subliminal.pdf describes it again more briefly.
https://web.eecs.umich.edu/~taustin/papers/VLSI06-sublim.pdf describes another one that reaches 2.6 pJ/inst at 833kHz. None of these processors are commercially available.

CoolRisc 81

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.1097&rep=rep1&type=pdf says the CoolRisc 81 uses 22 pJ/inst. In 2001!

Brains

http://icri-ci.technion.ac.il/files/2013/07/Karlheinz-Meier_red2.pdf Neurons use about 10 fJ (0.01 pJ) per neural spike (action potential) per synapse. It is not yet feasible to use them to build a portable computer.

Energy storage

A solar-powered computer could still be somewhat useful at night if it can store some energy. But standard batteries are flaky and break

after a few years, so fuck them. And we're talking about a super-low-power device, one that might use under 10mW when in active use, so even storing 1 joule of energy is potentially useful (100 seconds), 10 joules could last you 20 minutes, and 100 joules could last you three hours.

But how can you store 100 joules without batteries? I haven't found a good way.

If you can reduce the power consumption below 1mW, then a smaller amount of energy storage would be feasible.

Solid-state cell energy storage

http://www.mouser.com/pdfDocs/Cymbet-WP-EnerChips-vs-Supercaps.pdf claims that supercapacitors discharge at 10%–20% per day, while EnerChips (theoretically on the market but not in stock at Digi-Key) discharge at 1%–2% per *year*.

These solid-state batteries use a thin-film solid with ionic conduction as their electrolyte, so they are fully solid-state, despite the mobility of ions within them.

http://www.nature.com/nmat/journal/v14/n10/full/nmat4369.html discusses some new research on solid-state electrolytes for lithium batteries in more detail.

http://news.mit.edu/2015/solid-state-rechargeable-batteries-safer-longer-lasting-0817 is a press release about the paper.

Capacitor power storage

Solid-state capacitors are the physically stablest medium for storing a few joules of energy for short periods of time. They are low in capacity and expensive. To store 1J, a capacitor needs:

E <	C >
3V	0.2 F
10V	20000 µF
50V	800 µF
100V	200 µF
300V	22 µF
1000V	2 µF

Ceramics

Ceramic capacitors generally do not have the necessary energy capacity.

https://www.digikey.com/product-detail/en/C3216X5R0J476M160AC/445-1428-2-ND/569054 TDK Corporation C3216X5R0J476M160AC is 47µF, 6.3V, 3.20mm x 1.60mm, US$0.12. $CV^2/2$ = 1 mJ.

https://www.digikey.com/product-detail/en/AMK432BJ477MM-T/587-4368-1-ND/5405565 Taiyo Yuden AMK432BJ477MM-T is 470µF, 4V, 4.50mm x 3.20mm, US$5.34. $CV^2/2$ = 4 mJ.

https://www.digikey.com/product-detail/en/JMK325ABJ227MM-T/587-3980-1-ND/4950534 Taiyo Yuden JMK325ABJ227MM-T is

220μF, 6.3V, 3.20mm x 2.50mm, US$2.56. $CV^2/2$ = 4 mJ.

https://www.digikey.com/product-detail/en/B58033I5206M001/495-6706-5-ND/5039882 EPCOS (TDK) B58033I5206M001 is 20μF, 500V, 33.00mm x 22.00mm, 31g, US$89.90. $CV^2/2$ = 2.5J. Designed for industrial power inverters.

Films

Some film capacitors are designed for high voltages and consequently have substantial energy capacity. You could hook up the output side of an ordinary switchmode power supply (the part after the rectifier) designed for 240VAC line current to many of these, because in normal use a 240VAC-input SMPS's capacitors will be charged up to 340V.

http://www.righto.com/2014/05/a-look-inside-ipad-chargers-price-y.html gives an overview of these power supplies, and points out that you can get a crappy one with 5V output for US$3.

http://www.onsemi.com/pub_link/Collateral/SMPSRM-D.PDF explains a bit about how you could charge them from a low-voltage source such as solar panels.

https://www.digikey.com/product-detail/en/DS371506-CA/P9669-ND/821966 Panasonic Electronic Components DS371506-CA is 50μF, 370V, 50.50mm diameter × 112.00mm, US$7.87. $CV^2/2$ = 3.4 J. This capacitor is designed for shunting a motor.

https://media.digikey.com/pdf/Data%20Sheets/Panasonic%20Capacitors%20PDFs/AC%20Film%20Caps.pdf is the datasheet; specifies insulation resistance >1000 MΩ (giving $\tau > 14$h), explains it's an "oil wound metallized polypropylene film" capacitor.

https://www.digikey.com/product-detail/en/B32778G1276K/495-3933-ND/1884954 EPCOS (TDK) B32778G1276K is 27μF, 1300V, 57.50mm × 35.00mm × 50.00mm, US$14.48. $CV^2/2$ = 23 J.

http://en.tdk.eu/inf/20/20/db/fc_2009/MKP_B32774_778.pdf is the datasheet; gives lifetime expectancy of 100k hours (11 years) to 1M hours (110 years) at room temperature, but insulation resistance such that τ is only 10k seconds (3 hours).

https://www.digikey.com/product-detail/en/C44AFGP6200ZE0J/399-5955-ND/2704609 Kemet C44AFGP6200ZE0J is 200μF, 400V, 76.00mm diameter × 140.00mm, US$55.44. $CV^2/2$ = 16 J.

https://www.digikey.com/product-detail/en/B25620B1317K322/B25620B1317K322-ND/3489148 EPCOS (TDK) B25620B1317K322 is 310μF, 1320V, 116.00mm diameter × 103.00mm, US$82.83 (in quantity 16!). $CV^2/2$ = 270 J.

Tantalums

A few tantalum capacitors have reasonable energy capacity.

https://www.digikey.com/product-detail/en/T491B476K010AT/399-9728-2-ND/3724805 Kemet T491B476K010AT is 47 μF, 10V, 3.50mm x 2.80mm, US$0.12. $CV^2/2$ = 2 mJ.

https://www.digikey.com/product-detail/en/T491X476K035AT/39

o9-3821-2-ND/818681 Kemet T491X476K035AT is 47 µF, 35V, 7.30mm x 4.30mm, US$0.94 (in quantity 500!). CV²/2 = 30mJ.

https://www.digikey.com/product-detail/en/109D107X9060F2/7180-1226-ND/1559882 Vishay Sprague 109D107X9060F2 is 100µF, 60V, 7.92mm x 20.22mm, US$41.91. CV²/2 = 180mJ.

https://www.digikey.com/product-detail/en/TWDE503M006CB0Z0700/478-9404-ND/4990384 AVX Corporation TWDE503M006CB0Z0700 is supposedly 50 millifarads, but actually mislabeled; it's 50 microfarads. 6.3V, 9.52mm x 26.97mm, US$69.20. CV²/2 = 1 mJ. Also "wet tantalum".

https://www.digikey.com/product-detail/en/592D228X06R3X8T20H/592D228X06R3X8T20H-ND/2802652 Vishay Sprague 592D228X06R3X8T20H is 2200µF, 6.3V, 14.50mm x 7.37mm, US$3.69 (in quantity 500!). CV²/2 = 40mJ.

https://www.digikey.com/product-detail/en/TWAE228K025CBSZ0000/478-4966-ND/1879539 AVX Corporation TWAE228K025CBSZ0000 is 2200 µF, 25V, 9.52mm x 26.97mm, US$102.73, wet tantalum. CV²/2 = 700mJ.

https://www.digikey.com/product-detail/en/M39006%2F22-0660/1012-1025-MIL/2773731 Vishay Sprague [MIL] M39006/22-0660 is 56 µF, 125V, 10.31mm x 28.60mm, US$76.61. CV²/2 = 400mJ.

https://www.digikey.com/product-detail/en/TWAE157K125SBDZ0000/478-7301-ND/3451926 AVX Corporation TWAE157K125SBDZ0000 is 150 µF, 125V, 10.31mm x 26.97mm, US$87.20. CV²/2 = 1 J.

Supercapacitors

These are newish (like, the last 30 years). I'm not sure they can be trusted to last a long time; the numbers I'm seeing on Digi-Key's page are like "1000 hours at 70°".

http://www.murata.com/~/media/webrenewal/products/capacitor/edlc/techguide/electrical/c2m1cxs-053.ashx talks about the failure modes; it says they degrade little by little over time, although even after 5 years their 4.2V 470 mF supercap holds 2 J; they claim that their flat packages do a better job of keeping moisture out and electrolyte in than cylindrical packages, showing a degradation to 80% of capacity after 3000h (4 months). It also mentions that leakage current is around a microamp. On p.20 it shows a dryup lifetime graph, showing that even their DMT products will dry up in only 20 years at 70°, while their DMF products will dry up in only 5 years at 40°; on p.23 it explains that at 5.5V, their DMF will break after only 500 hours at 70° or 6400 hours (9 months) at 40°, while the DMT will last dramatically longer, like 40,000 hours at 70°.

https://www.digikey.com/product-search/en?pv13=1538&FV=fff40002%2Cfff8000c&mnonly=0&newproducts=0&ColumnSort=0&page=1&quantity=0&ptm=0&fid=0&pageSize=500 Nichicon JJD0E608MSEH is 6000 farads, 2.5V, 76.20mm × 168.00mm,

US$326. $CV^2/2$ = 18750 J. This sounds like science fiction.

https://www.digikey.com/product-detail/en/JUWT1105MCD/4930-4330-ND/2538684 Nichicon JUWT1105MCD is 1F, 2.7V, 6.30mm diameter × 10.50mm, US$0.86. $CV^2/2$ = 3.6 J.

https://www.digikey.com/product-detail/en/DSK-3R3H224U-HL/604-1020-1-ND/970232 Elna America DSK-3R3H224U-HL is 0.22F, 3.3V, 6.80mm × 3.00mm, US$2.16. $CV^2/2$ = 1 J.

Batteries

https://www.sparkfun.com/products/10319 is a 24.5 mm × 3.0 mm 3.6V LIR2450 rechargeable coin cell, rated for 110 mAh, which is 1430 J. It costs US$3.
https://electronics.stackexchange.com/questions/218655/running-3-03v-mcu-from-lir2032-lithium-ion-button-cell cites a LIR2032 (20 mm × 3.0 mm) cell with 40 mAh at 3–4.2 V, 520 J.

Horowitz & Hill (the 1989 edition with the micropower chapter) says that commonplace batteries (9-volt and AA batteries) "give nearly their full shelf life at drain currents less than 20μA." At 2 V and 200 pJ/insn (reasonable ballpark figures, though the STM32L does better) this works out to an average of 200 000 instructions per second. At 1400mAh for an ordinary 1.5 AA cell, of which you would need two (or a boost converter), this is 6 years, which is indeed comparable to the battery's shelf life. At 1 MIPS, the two-cell battery only lasts a year.

I bought a couple of rechargeable NiMH AA batteries the other day for some absurd price, like US$10 or something. They claim 2000 mAh, 1.2V, which is 8.6 kJ (per cell). At the full 3.5 mW power of a 16 MHz STM32L at 1.8V (see Notes on the STM32 microcontroller family (p. 3176)), that would run for 28 days.

Zamboni piles

A cell of the https://en.wikipedia.org/wiki/Zamboni_pile yields 0.8 volts, but only a few nA, and the Clarendon Dry Pile has been ringing its bell since 1840. The cell is perhaps 20mm in diameter, but <1mm thick; you could easily stack three of them to get 2.4 volts, perfectly adequate to run a microcontroller on without regulation. In particular, the Clarendon Dry Pile produces about 1 nA at 2 kV, consuming about 1 μg from each of its cells over the last 144 years, according to Croft's 1984 paper, "The Oxford electric bell." He doesn't give dimensions but the dry pile seems to be about 30mm in diameter, i.e. 700mm².

However, at least STM32s have a minimal current consumption of a few hundred nA (180 nA in standby mode with no RTC, 410 nA in standby mode with RTC). This would require, say, 500 times the cross-sectional area of the Clarendon battery: perhaps a 3-meter roll of 120mm-wide battery. If the total thickness were 1mm, the battery would be 350 mℓ. This seems like a small price to pay for a battery life measured in centuries, as long as the device doesn't have to be portable.

Memory

ISSI's IS62C256AL-45ULI-TR is 256 kilobits of parallel 5V 45ns SRAM for US$1.43 down to US$1.11 in quantity 500. It uses 150 μW

, or up to 15 mW in operation.

Microchip's 23K256-I/SN is 256 kilobits of SRAM in a little bitty 3.3V 8-pin 20MHz SPI SOIC for US$1.09 down to US$1.05. It uses 4 µA max for standby, which works out to 13.2 µW.

ISSI's IS62/65WVS2568FALL / IS62/65WVS2568FBLL is similar, 256 kiloBYTES of SRAM in a 3.3V SPI/SDI/SQI SOIC for US$2.20, but only available in quantity, running on 4 µA for standby.

The Adesto AT25SF041-SSHD-T is four megabits of 104MHz 2.5–3.6V Flash in the same 8-pin SOIC format, and it's much cheaper, US$0.36. Being Flash, it uses no power for "standby" but a lot of power to erase (10 mA for 500 ms to erase 32 kB ≈ 500 nJ/byte), but "only" 4 mA to read. It has quad outputs, so 104 MHz is pretty decent speed actually.

ISSI's IS62C1024AL-35QLI-TR is a megabit of parallel 5V 35ns SRAM for US$2.10 down to US$1.62 in quantity 500. It uses 20 µW, or up to 100 mW in operation.

In PSRAM, ISSI's IS66WV51216EBLL-55TLI is 8 megabits of parallel 55ns 2.5–3.6V DRAM, with all the refresh nonsense hidden inside the chip, for US$3.20 down to US$2.31 in quantity 1000. Because of the refresh nonsense, it uses more power, but not as much as you'd think — in CMOS standby mode (where presumably it retains the data) it uses 100 µA, which works out to 250–360 µW. That *is* about 80 times as much as the Microchip part, but it also has about 32 times as much memory, so the penalty isn't that bad.

Topics

Sparse filter optimization

Kragen Javier Sitaker, 2019-11-01 (5 minutes)

I've written a few items about "sparse filtering" previously, by which I mean factoring a DSP filter into some network of filters which requires very few total operations per sample. Generally I'm thinking in terms of single-rate LTI filtering, not multi-rate, nonlinear, or non-shift-invariant systems, and generally the nodes in the processing graph are basically comb filters of one kind or another.

This is a difficult optimization problem, being essentially combinatorial in nature: even for small numbers of operations per sample such as 10, the signal flow graph can have any of a large number of topologies, and each processing node in the graph can be configured in a large variety of different ways.

I just checked, and it seems that I haven't written down what now seems to me like the most straightforward way to handle this design problem, unless I'm not finding it.

Optimizing a fixed processing graph

Let's consider a processing graph made out of continuous-time delay, sum, and amplification nodes, and the problem to approximate a desired frequency and perhaps phase response with a fixed processing graph. The only parameters available are the delays and gains, and the frequency and phase response are continuous differentiable functions of them, so we can almost certainly use gradient descent and related methods (see, e.g., Using the method of secants for general optimization (p. 1773) and Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138)) to find a reasonable local optimum. In fact, for many problems, the dimensionality is small enough that I think we can use interval arithmetic to find the *global* optimum.

This is a continuous-time relaxation of the discrete-time problem we want to solve. Once a good solution to the relaxation has been found, imposing a penalty for fractional-sample delays is likely to be sufficient to find a good solution of the discrete-time problem; branch-and-bound is a surer approach, though exponential-time.

Deriving the processing graph

How can we derive the processing graph? A variety of approaches suggest themselves. We could start with a too-complex processing graph and use optimization penalties to drive most of the amplifier gains to 0, then remove the now-useless connections. We could start with a too-simple processing graph and alternate continuous-optimization phases with graph-mutation phases, which for example might replace a subgraph with two copies of that subgraph, each feeding to an attenuator, summed to form its original input, or insert a delay-0 element. Alternatively, we could just randomly generate a lot of graphs and try to optimize each of them separately.

A nontraditional choice that might improve the mutation properties of the graph is to use comb-filter nodes, which contain one delay, two gains, and a bit distinguishing whether they're feedforward

or feedback, rather than separate delay and gain nodes.

Databases and meet-in-the-middle

Of course, exploration of any optimization space can be facilitated by database and meet-in-the-middle approaches. In this case, we could begin any given search using a database of already-generated designs, starting the search from designs that already come close to meeting the desired specification; and, by dividing our desired frequency and phase response by that of every item in the database, we can get a set of frequency responses to search the database for.

For example, suppose we want a zero-phase gain of between 0.9 and 1.1 at 100 Hz and between 0.09 and 0.11 at 1000 Hz, and there's a signature in the database with a zero-phase gain of 0.5 at 100 Hz and 2.0 at 1000 Hz. So, if we found a system that had a zero-phase gain of between 1.8 and 2.2 at 100 Hz and between 0.045 and 0.055 at 1000 Hz, we could cascade the two together to get the desired response.

In practice, though, what we'd want to do is to find a database system or pair of database systems that get close to our objective (as measured by a response curve covering hundreds to millions of frequencies, or perhaps an affine-arithmetic approximation), then use local search as described earlier to converge precisely on that objective.

Even on modern hardware, the database can reasonably weigh terabytes, containing millions to billions of known inexpensive candidate systems. On future hardware we could conceivably tabulate trillions.

Cost functions

Aside from the penalty terms mentioned above, plus the penalty function for not meeting the desired response, there is some computational cost we're trying to minimize; this cost depends on the implementation technology. For digital computation, components of the cost function might include memory buffers, required bit width of those buffers, multiplications, and even additions. Analog signal processing is crucially limited by component precision and noise immunity, so you'd want to take the derivative of the system frequency response with respect to the vector of component values and injected noise and try to minimize that derivative; also, analog delay lines are not only bulky and expensive but also lossy. All of these situations can be handled by tweaking the cost function.

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Mathematical optimization (p. 3611) (29 notes)
- Convolution (p. 3391) (15 notes)
- Sparse filters (p. 3725) (11 notes)

Lenticular deflector

Kragen Javier Sitaker, 2019-09-08 (updated 2019-09-09) (9 minutes)

I think I might have a solution to the problem of deflecting macroscopic light beams through large angles under electronic control at submicrosecond time scales, using a plano-convex lens and a plano-concave lens, which translate relative to each other — or, rather, an array of each, like the surface of a lenticular 3-D image. Then you can get large light deflections out of submicron movements.

The problem of electronically rapidly deflecting light

One of the problems I've frequently confronted — in, for example, Bokeh pointcasting (p. 92), Photodiode camera (p. 2265), and CCD oscilloscope (p. 1861) — is how to rapidly change the direction of light beams, such as laser beams, under the control of some kind of control system, such as an electronic circuit. This is a standard problem, and Michelson and Morley's 19th-century solution to it — a prismatic mirror spinning on an air bearing — led to new measurements of the speed of light and is used today in, among other things, supermarket scanners and laser printers. (I think it also led to the modern dentist's drill.). However, the spinning mirror has a lot of momentum, so while the movement is very predictable, it is not very controllable.

Modern laser light shows confront this problem with "laser galvos", descendants of the old instrument for measuring small currents — a galvanometer connected to a mirror rather than a needle. These are used in pairs, one for X deflection and one for Y deflection. Galvos for laser shows are rated in "kpps", thousands of points per second, and a few tens of kpps is typical.

Contrast this with the electron gun in a CRT. High-end mass-market CRT computer monitors in the early 2000s supported 1200 scan lines at a 72 Hz refresh rate, meaning that even with magnetic deflection, they could scan the electron beam across the screen in a sawtooth-wave pattern at 86 kHz. Analog oscilloscopes, using electrostatic deflection, routinely managed 20 MHz vertical deflections with 3 dB attenuation; high-end ones could reach 100 MHz. To look at it a certain way, that's 200 000 kpps.

The limitation on galvos is that mirrors have mass, which makes it harder to move them. If you try to make them smaller so that they have less mass, you suffer from divergence problems, where focusing the laser beam onto a smaller mirror narrows the beam waist and induces beam spreading through diffraction.

This is a serious problem: light can change direction in femtoseconds, and it's easy to switch electricity in nanoseconds, but changing the direction of light with electricity takes microseconds. Isn't there any way to get that number down to hundreds of nanoseconds or better, without suffering milliradians of divergence?

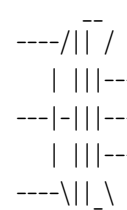
Well, there are lots of things that might work, like Kerr cells and dynamic LCD holography and ultrasonic gratings and whatnot, but I think I've found a good one.

Neutral meniscus lenses

A meniscus lens is concave on one side and convex on the other; it can be designed concave, convex, or neutral, so that collimated on-axis light entering the convex side comes out still collimated on the concave side. The exiting light is brighter and does not cover all of the lens; this effect gets stronger as the lens gets thicker, reaching a singularity when the lens thickness is the focal length of the convex side and, in the geometric-optics approximation, the light would come out in an infinitely thin pencil.

The sliced neutral meniscus lens

Take a neutral meniscus lens that is thick enough to be sliced along a flat plane parallel to its curved surfaces without interrupting either surface, and slice it in this way, dividing it into a plano-convex lens and a plano-concave lens. Separate the two halves by just enough air or oil to equal the delay of the glass removed. Aside from the stray light reflections at the new surfaces, this compound lens still behaves precisely as before, leaving collimated light collimated, but making it brighter.

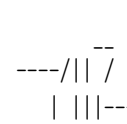


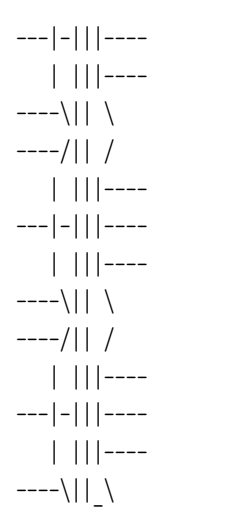
However, if we slide the two halves relative to each other, retaining flat-surface parallelism and distance, the collimated light exiting will change direction. If we slide the plano-concave half up, for example, the light passing through the center of the plano-convex lens, previously not deflected at all, now finds itself exiting just below the center of the plano-concave half, and so is refracted slightly down. The light a little above it enters the plano-convex lens and is refracted down, but exits through the center of the plano-concave lens, continuing its trajectory parallel to the other beam.

The beam remaining collimated depends on the deflection being linear with off-axis distance, and this linearity is not perfect. I think it's pretty good over ±30°, though, especially if the lens system is thick enough.

Microlens arrays

Consider the case where instead of one lens we have many, like two layers of that lenticular plastic covering that covers those 3-D parallax Jesus cards, except that the lenslets on one of them are concave:





This exhibits the same direction-changing behavior over the same angle with the same displacement as the single lens, but can handle a great deal more light.

You could fabricate this with little hexagonal lenslets in two dimensions, so that you can slide the two sheets relative to each other in two dimensions and deflect light on two axes, or you could fabricate it as two prismatic solids with complicated profiles amounting to many concatenated "cylindrical" lenses, like the typical 3-D lenticular parallax Jesus cards, and get only one dimension of displacement.

How fast can you move a thing? Tens of microseconds if it's 100 mm

The speed of sound in glass is commonly about 4500 m/s though it varies from 4000–6000 m/s for different glasses. Suppose you have a 100-mm-wide sheet of these lens things and you give it a shove on one edge, parallel to its surface, say with a piezoelectric actuator. That shove produces a sound wave that bounces back and forth through the glass about ten times before the whole thing has settled, in about 220 µs, using the above estimate of its speed of sound.

This sounds dismayingly slow, but I don't think the picture is that bleak. It's true that you have *latency* in the tens of microseconds, but it's very consistent, predictable latency. You can inverse-filter for the phase delay. XXX what about variation across the sheet while it's strained? Just turn the laser off except at key moments?

You might also be able to improve the situation with some acoustic impedance matching: some kind of matched resistance on the other side of the glass sheet that absorbs the shock and keeps it from bouncing back. That doesn't save you the initial 22-µs latency, but it means you don't have to contend with reflection.

This doesn't depend on the thickness of the sheet, as long as it's stiff enough not to buckle.

How small can you make the lenslets?

If you make the lenslets smaller than the wavelength of the light in question, they won't work at all. I think the variation in phase delay needs to be about half a wavelength from the center of the lenslet to its edge. That makes me think you should be able to make them on the order of 100 µm wide for visible light, even for fairly shallow

curvatures.

That means that the relative movements between the lenslet sheets to get light deflections can be on the order of 1 μm.

Uncertainty

Is it possible to deal with the acoustic delay somehow?

Will the light coming out of each lenslet have a different phase delay from the light coming out of the other lenslets? Does this mean that each individual beamlet will diffract on its own, giving a uselessly large divergence?

Topics

A survey of small TCP/IP implementations

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

Implementations

There are a few small TCP/IP implementations out there. Most recently, the Viewpoints Research Institute "STEPS toward the reinvention of programming" project has just done a "TinyTCP", including IP and TCP. [0] Although they claim it is "well under 200 lines of code" and provide some details about implementation techniques, it looks larger to me; see the section "TinyTCP" for details.

There have been a number of small TCP/IP implementations before. Adam Dunkels's 2001 "Miniweb", which is proprietary [1], supposedly implements a more or less working TCP, IP, and web server in about 400 lines of C.

Later, Dunkels wrote "uIP" and "lwIP", for "microIP" and "lightweight IP", which are complete and supposedly correct implementations of ARP, IP, UDP, and TCP. [2]

TinyTCP

I checked out TinyTCP r400 from Subversion:

```
svn co http://piumarta.com/svn2/idst/trunk/function/examples/tcp/
```

In the resulting directory, I counted the number of unique source lines:

```
cat *.k *.st | sort -u | wc -l
```

It counted 1270 lines of code, not "well under 200" as claimed. However, this includes the following (the listing is hand-annotated):

```
$ for x in *.k *.st; do printf "%30s " "$x"; sort -u "$x" | wc -l ; done
                  boot.k 152 construct the default environment (C iface)
               net-icmp.k 25  ICMP implementation
                 net-if.k 19  network pseudo interface
                 net-ip.k 62  the IP implementation
                net-tcp.k 81  TCP packet structure
             quasiquote.k 53  quasiquotation as userland syntax
              structure.k 87  the packet structure ASCII art parser
                   tcp2.k 78  TCP state machine, daytime, http
                    tcp.k 47  smaller version of the above
        Match-printing.st 74  COLA Smalltalk PEG parsing
                Match.st 221 more COLA Smalltalk PEG parsing
 NetworkPseudoInterface.st 108 TUN/TAP network interface in C and Smalltalk
               parse.st 275 more PEG parsing
          ParseStream.st 120 more PEG parsing
```

So the part that specifically pertains to TCP/IP, and not a particular network interface, or the system as a whole, and that isn't

duplicative, is much smaller:

```
$ sort -u net-icmp.k net-ip.k net-tcp.k structure.k tcp2.k | wc -l
311
```

A bit over 311 lines of code.

TCP Wrinkles

TCP itself is defined in RFC 793, STD 7 [3], which dates from 1981. But a number of problems have been discovered in TCP since then and worked around.

There's a full list of the specification documents in RFC 4614 [4]

RFC 896: where the term "congestion collapse" comes from.

http://64.233.169.104/search?q=cache:LGoJzUQCXH8J:www.welzl.oo at/research/publications/q2s-ntnu-2006-tcp.ppt+slow+start+nagle+oo syn+cookies&hl=en&ct=clnk&cd=10

"Requirements for Internet Hosts -- Communication Layers"

http://tools.ietf.org/html/rfc1122

"Known TCP Implementation Problems"

http://www.faqs.org/rfcs/rfc2525.html http://tools.ietf.org/html/rfc2525

Slow start.

Congestion avoidance. Increase congestion window by at most one segment per RTT.

http://tools.ietf.org/html/rfc2581 (TCP Congestion Control) (explains fast retransmit and fast recovery)

RTT estimation: Jacobson's algorithm.

http://tools.ietf.org/html/rfc2988 (Computing TCP's Retransmission Timer, 2000)

Jacobson, V. and M. Karels, "Congestion Avoidance and Control",

ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

Karn's algorithm.

http://en.wikipedia.org/wiki/Karn's_Algorithm

SYN cookies.

http://cr.yp.to/syncookies.html http://cr.yp.to/syncookies/archive

Fast retransmit.

Fast recovery.

Nagle.

Hard-to-predict initial sequence numbers.

http://www.faqs.org/rfcs/rfc1948.html

Stuff To Leave Out

Miniweb optionally does slow start, but not ...

SACK.

PAWS.

WSCALE.

NAK.

Path MTU discovery.

ECN.

Header prediction.

Silly Window Avoidance.

Delayed ACK.

References

[0] Viewpoints Research Institute Technical Report TR-2007-008, "STEPS Toward The Reinvention of Programming, First Year Progress Report," Dec 2007; the TinyTCP work is documented in Appendix E, "Extended Example: A Tiny TCP/IP Done As A Parser (by Ian Piumarta)", p.44, and the section "A Tiny TCP/IP Using Non-Deterministic Parsing, by Ian Piumarta", p.17.

http://www.viewpointsresearch.org/pdf/steps_TR-2007-008.pdf

[1] Adam Dunkels's Miniweb

http://www.sics.se/~adam/miniweb/

[2]

[3] Internet Society RFC 793, currently STD 7, "Transmission Control Protocol", by Jon Postel.

http://www.faqs.org/rfcs/rfc793.html

[4] Internet Society RFC 4614, by M. Duke, R. Braden, W. Eddy, E. Blanton, 2006-09, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents".

http://www.faqs.org/rfcs/rfc4614.html

[5] Internet Society RFC 2581, "TCP Congestion Control", by Van Jacobson (?), 1999-04

http://www.faqs.org/rfcs/rfc2581.html

Topics

- Small is beautiful (p. 3714) (40 notes)
- Networking (p. 3594) (7 notes)
- OMeta (p. 3605) (3 notes)
- TCP/IP (2 notes)

Modeling trees with slices containing metaballs

Kragen Javier Sitaker, 2014-06-29 (updated 2014-07-02) (6 minutes)

So I'm doing this tree-growth simulation thing in slices, where theoretically we want the total cross-sectional area of the tree to stay constant through branch points. That is, the branch before a fork should have the same total cross-sectional area as the two branches after it, and also that every intermediate point in the forking process should have the same cross-sectional area.

The underlying structure is a set of branch center points that move around in the XY plane as a function of Z, but my current problem is how to convert that into a three-dimensional shape that has the right thicknesses, transitions smoothly from one layer to the next, and is approximately circular around the branch center points.

I've been trying to fake this by putting a circle around each branch center point and inflating it in such a way that two overlapping circles with the same center will have twice the area as if there were just one of them, and doing a radius linear interpolation thing as their radii get further apart, until they have no overlap at all. This is not working super well for a couple of reasons: the circles intersect one another and don't have a way to find corresponding points to join with a mesh, so they're only a step toward the two-dimensional shell I want, not the final result; and when more than two of them overlap because two branch events happened close together in time or because circles were forced to collide, the resulting circles are too big, and sometimes discontinuously so.

So it occurred to me that you could solve most of this problem with some variety of gradient descent. You have a bunch of boundary points linked into a bunch of loops. You're trying to optimize the following:

- The area of each loop is close to the sum of the weights of the branch center points contained inside it.
- The difference between the boundary points in this generation and in the last generation is small.
- The total length of each loop is as short as possible.
- The loops do not intersect or self-intersect.
- The loops have the right number of points: no more than necessary, but enough to keep their form well-defined.

Most of these can be optimized by continuously varying the boundary points' coordinates, but sometimes you might instead split or join loops or reorder the points of a loop, which might be done heuristically. If you can really differentiate the merit function with regard to the coordinates, you can do real gradient descent as such; otherwise, you're stuck with blind hill-climbing search or genetic algorithms.

Implicit functions

Unfortunately most of those things require a lot of geometric algorithms, and Sedgewick has impressed upon me how tricky those

are. As an alternative, maybe I could use an implicit function, a function whose zero is a circle of the correct area when it's fed with only a single branch center, which is computed from the sum of the functions of the different branch centers, and where the contribution of a branch center falls off rapidly to zero as you get far away from it.

As an example, you could use a sum of Gaussians centered on each center point, scaled to match the branch's radius, and subtract 1 from the sum to get a function with useful zeroes.

However, you'd also want it to be the case that the area of overlapping branches is the area that would be the sum of the individual branches, at least in the common case I'm looking at here where I divide a branch by making two new branches in the same place whose eventual cross-sectional area, once they separate, will sum to the cross-sectional area of the original branch.

A necessary but not sufficient condition for this is that the gaussian-like function of the distance from the center have the property that doubling the function makes the distance from the center at which the sum falls to 1 increase by a factor of $\sqrt{2}$. It happens that Gaussians do in fact have that property, at least if you double them! You can show it by algebra, but you can also do the experiment:

```
import math
def gaussian(r): return math.exp(-r**2)

def search(f, v, min, max, tolerance):
    fmin, fmax = f(min), f(max)
    assert (fmin - v) * (fmax - v) <= 0
    if tolerance >= abs(fmin - v): return min
    if tolerance >= abs(fmax - v): return max
    mid = (min+max)/2.0
    fmid = f(mid)
    if (fmid - v) * (fmin - v) <= 0:
        return search(f, v, min, mid, tolerance)
    else:
        return search(f, v, mid, max, tolerance)
```

This code shows that gaussian(.832554) ≈ 0.5, and 2*gaussian(1.17741) ≈ 0.5. The ratio between those two is indeed $\sqrt{2}$. However, it is not true in general that gaussian(x)/2 = gaussian(x$\sqrt{2}$), although that shouldn't stop us from using it here.

It seems like if there's an analytic function that has these properties (minimally, that in general f(x)/2 = f(x$\sqrt{2}$), that it's always positive and finite, that f(x) = 1 at exactly one positive x, and that its limit at infinity is zero) it should be unique.

I've already written an interval-arithmetic package in JS that could be used to efficiently divide the canvas into an implicit k-d tree to search for zero pixels. It may need to be enhanced somewhat to look harder for the zeroes, down into deep subpixel, in order to avoid false positives.

Topics

Regenerative fuel air cutting

Kragen Javier Sitaker, 2016-09-06 (4 minutes)

An oxy-acetylene flame can reach 3500°, which can easily heat steel past its kindling point of about 870°. Other oxy-fuel flames, like propane at 2800°, can also flame-cut steel. But single-tank torches that mix their fuel with air generally cannot, because they have to heat up a lot of inert nitrogen along with the active components, and even though the resulting temperature is higher than steel's kindling point, it's not hot enough to heat the steel fast enough; air-acetylene tops out at 2500°, propane at 1980°, gasoline at 2140°, hydrogen at 2250°, wood at 1980°.

You might be tempted to heat the air by running it through *two* flames, but that doesn't work, because the oxygen in the air has been used up; you'd have to mix oxygen back in, which lowers the temperature. What you need is a way to separate the heat from the gas and transfer it to fresh air, which you then use to burn a second flame.

You can do this with a regenerator. Taking propane as an example, you can heat the regenerator to 1980° with a first propane flame, and then heat fresh air to 1980° with the regenerator before using it to fuel a second propane flame. Propane has a specific heat capacity of 73.6 J/K/mol (and a molar mass of 44.1 g/mol), while air's is about 29.2 J/K/mol (and a molar mass of basically 30 g/mol), the stoichiometric mixture is $C_3H_8 + 10O_2 \rightarrow 3CO_2 + 4H_2O$, and air is only about 21% oxygen, so each mole of propane needs 10 moles of oxygen and 47.6 moles of air to burn. So the stoichiometric mixture is 97.9% air by volume, 97.0% air by mass, and 95.0% air by thermal mass, so if the air is at 1980° and the propane is at 20°, then the mixture would be at 1882° if it didn't immediately burn.

But it does immediately burn, which should raise its temperature by another 1960 K to about 3840°, quite a bit hotter than the oxy-acetylene flame, and plenty hot enough to cut steel.

For such a high temperature, both the regenerator and the combustion chamber would need to be of a material that can withstand the full 1980° produced by the propane-air flame. This is fairly demanding, and only a relatively small number of such refractory materials are available, including zirconia (melts at 2715°), urania (melts at 2865°), thoria (melts at 3300°), graphite or carbon (subliming at 6000°, but will burn at 700° if oxygen is available), and lime (melts at 2615°.) But if we only want to achieve the 2800° achieved by an oxy-propane flame, the regenerator only needs to withstand 840°, which is achievable by any number of everyday materials, including iron, steel, copper, and most kinds of dirt and rock.

It isn't necessary to use modern fuels like propane; syngas from a wood gasifier would work just as well.

(Other low-tech ways of flame-cutting steel are possible; Theodore Gray has demonstrated flame-cutting steel using thermic lances made of prosciutto and breadsticks and a cucumber, for example.)

There are several ways you could arrange the regenerator cycles;

for example, you could have two or three regenerators and alternate between them using valves, or you could mount the regenerators in a wheel like the desiccant wheel of a desiccant-type dehumidifier.

This approach should make it possible to flame-cut steel without using any exotic materials.

Optimizing for temperature

If we use the regenerator technique to optimize for high temperature rather than for easily-available materials, using a thoria regenerator to preheat oxygen for an oxy-acetylene flame should enable much higher temperatures, though not the 6800° you'd naïvely think, because the acetylene is a much larger fraction of the thermal mass entering such a flame, and the acetylene is by necessity not preheated.

This approach should make it possible to flame-cut stainless steels and aluminum, maybe glass and stone and other materials that cannot normally be flame-cut except with a plasma torch. (Plasma torches can reach over 20,000°.)

Topics

- Materials (p. 3560) (112 notes)
- Independence (p. 3520) (63 notes)
- Manufacturing (p. 3558) (50 notes)
- Thermodynamics (p. 3747) (49 notes)
- Heat exchangers (p. 3497) (5 notes)
- Regenerators (p. 3679) (4 notes)
- Fire

Optical lever thermometer

Kragen Javier Sitaker, 2015-09-03 (1 minute)

Was just watching Dan Gelbart's video on building large structures with adhesives (http://youtu.be/EeEhS3zmnDg), and he talks about the danger of mounting mirrors on silicone that changes size with temperature: if the adhesive layer is not of even thickness, its thermal expansion and contraction (an order of magnitude greater than that of metal or glass) will rotate the mirror, and the optical lever effect then can give you a substantial displacement! Maybe you can use this to get a thermometer, although liquids like mercury should have 3× greater coefficient in effect, since you're interested in the volumetric rather than linear coefficient of expansion there.

Some page about cleanrooms says that typically silicones have 30 to 300 ppm per kelvin expansion. So if you have a 1cm-wide mirror on a 5mm-thick wedge-shaped cushion of silicone, it should expand by half a micron per kelvin, or about 10 arc seconds per kelvin. That's half a millimeter of displacement at a distance of 10 meters.

This effect seems too small to make into a very useful thermometer.

Topics

• Optics (p. 3609) (34 notes)
• Metrology (p. 3579) (18 notes)
• Gelbart (p. 3470) (2 notes)

Intro to algorithms

Kragen Javier Sitaker, 2016-09-06 (4 minutes)

If I want my paper-oriented algorithm notation to be useful, one avenue is to express algorithms that people are interested in in it. Perhaps introduction-to-algorithms classes would be a good place to find them. So here is a summary of some intro-to-algorithms classes, covering mostly only the algorithms, although in a few cases I've also included the problems because it wasn't clear which algorithm was being taught. I've probably missed a lot.

http://ocw.mit.edu/courses/electrical-engineering-and-computer-sci
ence/6-006-introduction-to-algorithms-spring-2008/calendar/ is an intro-to-algorithms syllabus. Algorithms it covers:

- document distance
- mergesort
- airplane scheduling
- binary search trees
- balanced binary search trees
- avl trees
- hashing
- hash table doubling
- karp-rabin hashing
- rolling hashes
- open addressing
- heaps
- heapsort
- stable sorting
- radix sort
- counting sort
- bucket sort
- graph search
- breadth-first search
- depth-first search
- topological sort
- shortest paths
- bellman-ford
- dijkstra's algorithm
- dynamic programming
- memoized fibonacci
- longest common subsequnce
- text justification
- knapsack
- maximum-sum subarray
- vertex cover
- dominating set
- integer multiplication
- matrix multiplication
- strassen's algorithm

http://www.cs.rpi.edu/~goldsd/spring2013-csci2300.php is another, in some sense much less comprehensive. It covers:

• graph coloring
• depth-first search
• breadth-first search
• binary search
• topological sort
• strongly-connected components
• shortest paths
• bellman-ford
• dijkstra's algorithm
• euler tour
• heaps
• minimum spanning tree
• kruskal
• prim
• dynamic programming
• memoized fibonacci
• longest increasing subsequence
• pascal's triangle
• sierpinski's triangle
• longest path
• hamiltonian cycle
• traveling salesman
• genetic algorithms
• satisfiability

http://www.cs.rit.edu/~anh/alg_resources.html is another. Unfortunately it doesn't give any significant detail on what is covered.

http://www.fas.harvard.edu/~libcs124/E124/syllabus.html is another, with rather nice lecture notes. It covers:

• debiasing coin
• mediation and duplation
• fibonacci by repeatedly squaring matrices
• mergesort
• depth-first search
• breadth-first search
• dijkstra's algorithm
• minimum spanning tree
• prim
• kruskal
• union-find with path compression
• satisfiability
• set cover
• huffman coding
• longest common subsequnce
• shortest paths
• all-pairs shortest paths
• traveling salesman
• matrix multiplication
• strassen's algorithm
• dynamic programming
• edit distance
• hashing

• bloom filters
• primality testing
• linear programming
• simplex algorithm
• vertex cover
• max cut
• factoring
• suffix trees
• one-time pad
• euclid's algorithm
• extended euclid's algorithm
• rsa
• least common ancestor
• range minimum query
• network flow

http://www.cse.msstate.edu/~swan/teaching/2008-3_CSE-4833-6830_Algs/schedule.htm is another, far less comprehensive:

• insertion sort
• mergesort
• heapsort
• quicksort
• dynamic programming
• minimum spanning tree
• shortest paths

http://cs.smith.edu/~streinu/Teaching/Courses/252.html is another:

• depth-first search
• breadth-first search
• connectivity
• minimum spanning tree
• shortest paths
• traveling salesman
• longest path
• hamiltonian cycle
• satisfiability
• clique
• vertex cover
• integer multiplication

Finally, http://courses.cs.washington.edu/courses/cse421/14su/ covers:

• fft
• matrix multiplication
• strassen's algorithm
• knapsack
• edit distance
• depth-first search
• breadth-first search
• strongly-connected components
• shortest paths
• minimum spanning tree

• satisfiability
• vertex cover
• graph coloring
• traveling salesman
• huffman coding
• connectivity
• topological sort
• bipartiteness
• fewest coins
• interval scheduling
• dijkstra's algorithm
• mergesort
• inversion counting
• closest pair of points
• integer multiplication
• karatsuba
• rsa
• dynamic programming
• memoized fibonacci
• dynamic programming fibonacci
• dynamic programming string alignment
• dynamic programming fewest coins
• weighted interval scheduling
• rna structure
• network flow

Here are the items that appear in more than 2 of those lists:

```
6 shortest paths
5 minimum spanning tree
5 dynamic programming
5 depth-first search
5 breadth-first search
4 vertex cover
4 traveling salesman
4 satisfiability
4 mergesort
4 dijkstra's algorithm
3 topological sort
3 strassen's algorithm
3 memoized fibonacci
3 matrix multiplication
3 integer multiplication
```

That's 15 algorithms or problems, which seems like a reasonably short list of things to cover; some of them are even redundant, like Strassen's algorithm to solve matrix multiplication.

It seems sort of bizarre to me that graph algorithms are given such great prominence in these introductory courses, since they seem, to me at least, to have relatively little importance in practical day-to-day programming. Certainly we work with graphs all the time, and we even occasionally topologically sort them. But shortest-path algorithms and minimum spanning trees almost never occur in my experience (except in Ethernet switches). Even dynamic programming is relatively uncommon, and search needs heuristics

most of the time to be useful.

Topics

A homoiconic language with a finite-map-based data model rather than lists?

Kragen Javier Sitaker, 2019-09-25 (updated 2019-09-28)
(46 minutes)

I wrote a mock Lisp the other night, which was a surprisingly pleasant experience. Thanks to LuaJIT, it took me only a couple of hours to get from nothing to generating reasonably fast native code — wasting only 70% of the CPU's performance rather than 80% as with Ur-Scheme or 95% as with CPython. The mock Lisp isn't powerful enough for a metacircular interpreter, since it lacks data structures, but it's powerful enough to write a recursive Fibonacci number function.

This led me to wonder whether an imperative homoiconic programming language based on maps rather than lists could be a better alternative to Lisp. I think it'll necessarily have more redundancy than Lisp, since a map with N keys has $N!$ equivalent permutations (so the parsing process discards lg (N!) bits), but that may not be a bad thing; after all, we can diminish redundancy further by dropping to Forth, PostScript, or APL.

After examining some clumsy alternatives, I think I have a reasonable alternative based on a forgotten SourceForge project for text munging.

The magic of READ and PRINT

Although LuaJIT is amazing, the experience of debugging things at the LuaJIT REPL made me wish for Python, JS, OCaml, or Lisp — languages where your data structures can be automatically serialized in a parseable form, a very handy feature not only for interactive testing but also for network communication, ad-hoc filesystem persistence, manual fixup of broken systems, primitive user interfaces, and shared-nothing message-passing parallel and concurrent processing, as with fork(). Here's a rehearsed interaction with the OCaml interpreter:

```
$ ocaml
        OCaml version 4.02.3

# type rope = Leaf of int * string | Cat of int * rope * rope ;;
type rope = Leaf of int * string | Cat of int * rope * rope
# let rope_length = function Leaf(a, _) -> a | Cat(a, _, _) -> a ;;
val rope_length : rope -> int = <fun>
#   let leaf s = Leaf(String.length s, s)
    let cat a b = Cat(rope_length a + rope_length b, a, b)
    let cat2 a b = match (a, b) with
    | (Leaf(n1, s1), Leaf(n2, s2)) when n1+n2 < 128 ->
        leaf(s1 ^ s2)
    | (Cat(_, x, Leaf(n1, s1)), Leaf(n2, s2)) when n1+n2 < 128 ->
        cat x (leaf(s1 ^ s2))
```

```
    | (Leaf(n1, s1), Cat(_, Leaf(n2, s2), x)) when n1+n2 < 128 ->
        cat (leaf(s1 ^ s2)) x
    | (_, _) ->
        cat a b
;;

val leaf : string -> rope = <fun>
val cat : rope -> rope -> rope = <fun>
val cat2 : rope -> rope -> rope = <fun>
# cat2 (cat2 (leaf "x") (cat2 (cat2 (leaf "x")
                                 (leaf (String.make 128 'h')))
                       (leaf "x")))
     (leaf "x")
;;
- : rope =
Cat (132,
 Cat (131, Leaf (1, "x"),
  Cat (130,
   Cat (129, Leaf (1, "x"),
    Leaf (128,

      "hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh")),
   Leaf (1, "x"))),
 Leaf (1, "x"))
#
```

(This experimentally confirms that the leafnode-coalescence strategy used by cat2, proposed in B-Tree ropes (p. 2762), does indeed experience pathological fragmentation for some workloads.)

Here's JS, though you can't see the pleasant syntax highlighting Node applied to its output:

```
$ node
> x = {a: [3, 4], b: 5}
{ a: [ 3, 4 ], b: 5 }
> x.b += 4
9
> x.c = ['okay']
[ 'okay' ]
> x.c.push('now')
2
> x
{ a: [ 3, 4 ], b: 9, c: [ 'okay', 'now' ] }
>
```

Here's the same interaction in Python, which is quite a bit fussier than JS, but still comes through:

```
>>> x = {"a": [3, 4], "b": 5}
>>> x["b"] += 4
>>> x["c"] = ['okay']
>>> x["c"].append('now')
>>> x
{'c': ['okay', 'now'], 'a': [3, 4], 'b': 9}
```

Lua has READ but not PRINT

Now contrast Lua, which is eminently capable of handling the same kind of flexible data structures, but doesn't come with any way to print them:

```
$ luajit
LuaJIT 2.0.4 -- Copyright (C) 2005-2015 Mike Pall. http://luajit.org/
JIT: ON CMOV SSE2 SSE3 SSE4.1 fold cse dce fwd dse narrow loop abc sink fuse
> x = {a = {3, 4}, b = 5}
> x.b = x.b + 4
> x.c = {'okay'}
> table.insert(x.c, 'now')
> =x
table: 0x40542160
> =x.a
table: 0x405485a8
> =x.a[1]
3
> =x.a[2]
4
> =x.b
9
> =#x.a
2
> =#x.c
2
> =x.c[1]
okay
> =x.c[2]
now
> =table.unpack(x)
stdin:1: attempt to call field 'unpack' (a nil value)
stack traceback:
        stdin:1: in main chunk
        [C]: at 0x004044a0
```

I guess `table.unpack` wasn't added until Lua 5.2, and LuaJIT is a Lua 5.1. Doesn't matter, because it wouldn't have helped anyway — `table.unpack` is only for lists, not for dictionaries. There's a `pairs` function in core Lua for iterating over dictionaries, but just printing it isn't useful; it returns an iteration state, not an unpacked sequence:

```
> =pairs(x)
function: builtin#4     table: 0x4163d160       nil
```

You actually have to write code to iterate over the pairs:

```
> for k, v in pairs(x) do print(k, v) end
b       9
a       table: 0x405485a8
c       table: 0x40548920
```

And of course it doesn't recurse; you have to do that yourself:

```
> for k, v in pairs(x.a) do print(k, v) end
1       3
2       4
> for k, v in pairs(x.c) do print(k, v) end
1       okay
2       now
>
```

S-expressions and their discontents

Lisp S-expressions are probably a sort of minimum-complexity way to give you fully general data structures that are readable and printable, with a syntax you can write in a single-rule BNF grammar:

```
sexp ::= [ \n]* "(" sexp* [ \n]* ")" | [ \n]* [-A-Za-z0-9*]+
```

And once you implement that, which is flexible enough to use for any kind of tree data structure, it's straightforward to use them for your source code as well as your data structures, although many people complain about the clarity of the resulting code. Here's the recursive-descent parser I hacked together in half an hour (plus a couple of cleanups), using Lua's built-in list structure and `string.match`, although I probably should have used LPEG:

```
function read_sexp(c, getc)
   while c:match("[%s]") do c = getc() end
   if c == '(' then return read_list(getc(), getc) end
   return read_atom(c, getc)
end

function read_list(c, getc)
   while c ~= nil and c:match("[%s]") do c = getc() end
   if c == ')' then return nil end

   local car, c2 = read_sexp(c, getc)
   if c2 == nil then c2 = getc() end

   return {car=car, cdr=read_list(c2, getc)}
end

function read_atom(c, getc)
   local name = {}

   while c ~= nil and c:match("[^%s()]") do
      table.insert(name, c)
      c = getc()
   end

   name = table.concat(name)
   if not name:match("[^%d.]") then return tonumber(name), c end
   return name, c
end
```

With this, the compiler can successfully parse and compile programs such as the following:

```
(letrec (fib (lambda (n)
               (if (< n 2)
                   1
                   (+ (fib (- n 1)) (fib (- n 2))))))
  (fib 40))
```

S-expressions have some real merits. They have great simplicity of implementation, and they're relatively light on delimiters, which makes them easy t type; compare JS's `{ a: [ 3, 4 ], b: 9, c: [ 'okay', 'now' ] }` or Python's `{'c': ['okay', 'now'], 'a': [3, 4], 'b': 9}` to `(a (3 4) b 9 c (okay now))` or even `((a (3 4)) (b 9) (c (okay now)))`.

The discontents

However, S-expressions also have some real drawbacks.

As you can see in the string `))))))` in the above mock Lisp program, they expose the deeply nested nature of the data structure rather crudely; this is often unhelpful to the humans. Tim Peters's line in the Zen of Python, "Flat is better than nested", is a response to this. Although the humans are capable of recursive thought, it is a lot of effort for them, so they do much better when they can stick to finite state machines and Markov chains.

Code that stores data as S-expressions can be somewhat inscrutable and therefore bug-prone; consider this Elisp snippet from files.el:

```
    (if (and mode
             (consp mode)
             (cadr mode))
        (setq mode (car mode)
          name (substring name 0 (match-beginning 0)))
      (setq name nil))
```

What are `(car mode)` and `(cadr mode)` (that is, the second item of the list `mode`)? They're some kind of fields of a data structure, but it's hard to tell what they intend. Fairly often Elisp will unpack such lists at the entry to a function or the top of a loop, which inflates the code a bit (this from Ken Manheimer's allout.el outliner mode):

```
  (while pairs
    (let* ((pair (pop pairs))
           (name (car pair))
           (value (cadr pair))
           (qualifier (if (> (length pair) 2)
                          (caddr pair)))
           prior-value)
      ...
```

A perhaps more subtle problem of S-expressions is their extensibility. The last example above handles tuples of the structure `(name value)`, which may be extended with an optional `qualifier` to become `(name value qualifier)`. Perhaps at some future point a `scope` or `mode` will be added.

As with Protocol Buffers, it's safe to add new items at the end of such tuple-shaped lists — but only if nobody else is doing so concurrently somewhere else, or you'll end up misinterpreting each other's data. That is, if I add a `scope` item as the fourth item, and you

add a `mode`, and for whatever reason I end up running my code on your data (from your .emacs.d/init.el, perhaps), something will go wrong.

(An example of this is how Racket doesn't really parse your program into cons nodes; instead it parses it into things that are similar to cons nodes but also contain file, line number, and column number information, so that it can report runtime errors in context. Adding such a feature using normal cons nodes in a way that wouldn't break existing users would be infeasible.)

Lists being used as lists are much worse, though — there's no way to add any extra data to them (other than per list item) without breaking backward compatibility.

Finally, S-expressions use a buttload of memory, especially on 64-bit machines: at least two pointers per list item, plus potentially extra space for type tags, garbage collection tags, locks, and so on.

These aren't really drawbacks of Lisp, except for))))))); Common Lisp, Scheme, Racket, Clojure, and even Elisp have a variety of other data structures and aren't limited to cons chains. It's a drawback of just storing data in cons chains.

Problems like these are why awk, Python, JS, and Lua privilege finite maps (also known as dictionaries, tables, associative arrays, or hashes) over lists or arrays, and OCaml is instead based on discriminated unions (although it has lists and tuples). Finite maps, sets of name-value pairs, have a sterling record of backward-compatibility in things like email headers, HTML element attributes, and library APIs.

However, none of these non-Lisp languages has attempted to use the same syntax for code and for data. I think it might be a fun thing to try.

Nested dicts

The simplest approach to a homoiconic language syntax consisting of finite maps is to just use S-expressions with an even number of items, and interpret them as finite maps. Clojure does something like this:

```
user=> {:a [3 4] :b 5}
{:b 5, :a [3 4]}
```

(The leading : in Clojure marks a keyword, which is autoquoted in much the same way as in Common Lisp or Ruby.)

The grammar for such a dictionary-expression approach is very nearly as simple as that for ordinary S-expressions:

```
dexp ::= [ \n]* "{" (dexp dexp)* [ \n]* "}" | [ \n]* [-A-Za-z0-9*]+
```

It remains to be seen how to encode programs in this form in a usable fashion. First, though, some exploration of a subtle point that can produce a lot of confusion.

Reading, evaluation, and auto-quoting

There's a subtle distinction which is glossed over above. Python reads "dictionary displays" like `{3: 4}` differently from the way Lisp parses S-expressions. Python, like Lua, JS, Ruby, and OCaml, is *evaluating* these expressions, with potentially Turing-complete

consequences. Lisp's READ, by contrast, just *parses* them, though if you type them in at the REPL prompt it will evaluate them, and they may evaluate to themselves. Here's an edited interaction with the SBCL implementation of Common Lisp:

```
* (read)
(x y z)
(X Y Z)
*
```

That is, I typed `(read)` at the prompt, and then it waited for another S-expression of input; I typed `(x y z)`, and it responded by parsing that and returning a list of those three symbols, which unfortunately it prefers to print in uppercase, as if it were still 1962. By contrast, if I type `(x y z)` at the prompt, first it parses ("reads") that into the same list as before, and then attempts to evaluate it, which fails because I haven't defined the variable `y`:

```
* (x y z)
; in: X Y
;     (X Y Z)
```

...(many lines omitted)...

```
debugger invoked on a UNBOUND-VARIABLE in thread
#<THREAD "main thread" RUNNING {100399C553}>:
  The variable Y is unbound.
```

If you actually want that list of three symbols, you can use `quote`, optionally abbreviated as `'`, both for input and for output:

```
* (quote (x y z))
(X Y Z)
* (list (quote x) (quote y) (quote z))
(X Y Z)
* '(x y z)
(X Y Z)
* (list 'x 'y 'z)
(X Y Z)
* '(quote (x y z))
'(X Y Z)
```

It happens that, in Common Lisp and in Elisp, evaluation of lists does things such as call functions and apply macros, but evaluation of other data such as integers or strings just returns that data (it is "auto-quoting"). In fact, this rule even extends to things like "vectors", which is to say, arrays:

```
* (defvar x 99)
X
* #(3 x 9)
#(3 X 9)
```

We got the symbol `X` rather than the value 99. What happened is that the whole vector `#(3 X 9)` was read, and vectors are auto-quoting,

so evaluating it simply returned the vector. If we want to build up a vector from values computed at run-time, we have to call the function `vector`:

```
* (vector 3 x 9)
#(3 99 9)
```

Smalltalk originally had the same problem, although Squeak has a fix. Clojure, on the other hand, follows Common Lisp by separating reading from evaluation, but follows Python by evaluating items inside of aggregate data structures unless they are explicitly quoted; like Elisp, JS, and Python, but unlike Common Lisp and Lua, it uses [] to delimit vectors/arrays:

```
$ clojure
Clojure 1.6.0
user=> (def x 99)
#'user/x
user=> [3 x 9]
[3 99 9]
user=> '[3 x 9]
[3 x 9]
user=> {x 3 (+ x 1) 4}
{99 3, 100 4}
user=> '{x 3 (+ x 1) 4}
{x 3, (+ x 1) 4}
user=> (type [3 x 9])
clojure.lang.PersistentVector
user=> (type '[3 x 9])
clojure.lang.PersistentVector
user=> ({x 3 (+ x 1) 4} x)
3
user=> ({x 3 (+ x 1) 4} 'x)
nil
user=> ('{x 3 (+ x 1) 4} 'x)
3
user=> (read)
[3 x 9]
[3 x 9]
```

In Clojure, only primitive atomic data types like numbers, strings, and keywords are auto-quoting. But it happens that, if the only things in your map or vector are auto-quoting, it evaluates to itself:

```
user=> {:x 3 :y 4}
{:y 4, :x 3}
```

(The comma is optional.)

This can be confusing because it obscures the fact that an evaluation is happening, as in Python, unlike in Common Lisp. Python is fussy enough to remind you of this if you forget, requiring a lot of extra line-noise punctuation in your literal "dictionary displays":

```
>>> x = {a: [3, 4], b: 5}
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> x = {"a": [3, 4], "b": 5}
>>> x
{'a': [3, 4], 'b': 5}
```

JS and Lua, on the other hand, have special cases in their syntax so that you can forget most of the time:

```
> x = {a: [3, 4], b: 5}
{ a: [ 3, 4 ], b: 5 }
```

Lua:

```
> x = {a = {3, 4}, b = 5}
```

If you really want to force the evaluation of an expression for a key consisting of just a variable, there is special syntax for this in Lua:

```
> a = 5
> x = {[a] = 8}
> =x[5]
8
> =x.a
nil
```

And in JS:

```
> a = 8
8
> x = {a: 4}
{ a: 4 }
> x = {[a]: 4}
{ '8': 4 }
```

Encoding fundamental imperative-language constructs

An imperative programming language contains at least sequence, assignment, and looping; a normal programming language also contains identifiers, subroutine definitions, subroutine invocations, conditionals, and primitive operations. How can you encode these usably in a tree of maps?

I mean, obviously you can write an abstract syntax tree of the form

```
{nodetype binop operator +
 left {nodetype identifier name x}
 right {nodetype constant value 1}}
```

but this results in a very poor signal to noise ratio. Even if you don't use maps to represent the leafnodes, it's pretty bad:

```
{nodetype binop operator + left x right 1}
```

But can we do better?

Consider the example mock Lisp program I mentioned earlier:

```
(letrec (fib (lambda (n)
               (if (< n 2)
                   1
                   (+ (fib (- n 1)) (fib (- n 2))))))
  (fib 40))
```

Built-in operation invocations

A Smalltalk-like approach, treating things like addition like method calls, could reduce the noise factor somewhat:

```
{: x + 1}
```

We could treat function invocation in a similar way, providing named arguments. With this approach the Lisp expression

```
(+ (fib (- n 1)) (fib (- n 2)))
```

becomes

```
{: {: fib n {: n - 1}} + {: fib n {: n - 2}}}
```

which seems, if not ideal, at least potentially tolerable.

Because the sequence of name-value pairs is undefined, though, this could just as well be rendered as follows:

```
{+ {n {- 2 : n} : fib} : {n {- 1 : n} : fib}}
```

which seems pretty confusing; the interpreter is going to be looking for the : to see if the node is such a method-invocation node, and not having it first makes it harder to determine what is going to happen.

Conditionals

Lisp-style `cond` is not easy to obtain, but an if-then-else node is easy:

```
{if {: n < 2}
 then 1
 else {: {: fib n {: n - 1}} + {: fib n {: n - 2}}}}
```

Assignments

Lisp-like `let` or `letrec` fits very nicely into this sort of scheme; consider this Elisp (also from allout.el):

```
(let ((start (point))
      (ol-start (overlay-start ol))
      (ol-end (overlay-end ol))
      first)
  body)
```

This translates to

```
{let {start {: point}
      ol-start {: overlay-start overlay ol}
      ol-end {: overlay-end overlay ol}
```

```
     first nil}
 in body}
```

Imperative side-effecting assignments like this one (also from allout.el) demand a different approach:

```
(when (not first)
  (setq first (point)))
```

I mean there are lots of ways you could spell that:

```
{set first to {: point}}
{let first = {: point}}
{my first is {: point}}
{make first be {: point}}
```

However, I favor these, because they comfortably support parallel assignments (like the many-argument `setq` in one of the examples above) and have less noise words:

```
{= {first {: point}}}
{set! {first {: point}}}
```

Subroutine definitions

Darius Bacon has been working on a new dialect of Scheme called "Cant", previously "Squeam", in which the fundamental procedure-call mechanism uses a pattern-matching mechanism on the argument list to select a method to invoke on the receiver object. That is, you don't have procedures as such, just receivers. This provides a very nice unification of ML-style pattern matching and Smalltalk-style object orientation.

You could do something similar here, defining functions as sets of argument-list/body pairs. To guarantee determinism the compiler would have to verify that the argument lists were mutually exclusive.

This allows us to translate the Lisp above:

```
(letrec (fib (lambda (n)
               (if (< n 2)
                   1
                   (+ (fib (- n 1)) (fib (- n 2))))))
  (fib 40))
```

as

```
{let {fib {lambda {{n {}}
                   {if {: n < 2}
                    then 1
                    else {: {: fib n {: n - 1}}
                          + {: fib n {: n - 2}}}}}}}
 in {: fib n 40}}
```

which in this case contains only a single argument list, containing `n`. It's not clear what kind of values to associate with the arguments; maybe `{n {default 2}}` would be a valid specification.

Looping

The Common Lisp LOOP macro provides a good example of how you can usefully specify a loop as a set of name-value pairs: `{while foo do bar}` and `{for i = 1 to 10 do bar}` are simple examples, but it is reasonable to support also `{for x in mylist collect {: f n x} when {: x > 2}}` and the like.

Sequencing

I've saved the worst for last. In the language as described above the only obvious way to sequence is by nesting; things that aren't nested have no sequence. You could potentially use line numbers:

```
{prog {10 {print "HOWIE IS AWESOME"}
      20 {goto 10}}}
```

but failing that you are stuck with constructs of the form `{do x then y}`, which you must nest to get sequences of an arbitrary number of steps.

Flat dict syntax

So, what if we use infix syntax to build our dicts instead of circumfix syntax? I think I've written a bit about this before; the idea is that you use parentheses merely for grouping, and you have two operators `:` and `,` which you can use to build up an arbitrary dict. `x: y` is a single-entry dict in which the key `x` has the value `y`, and `a, b`, with lower operator precedence, is the union of the dicts `a` and `b`, with some kind of rule about key conflicts (either it's an error or there's a defined winner).

We need to change the name of the invocation-target tag `:` to something else, ideally something inoffensive; `.` is a reasonable candidate.

So our translation above

```
{let {fib {lambda {{n {}}
                  {if {: n < 2}
                   then 1
                   else {: {: fib n {: n - 1}}
                         + {: fib n {: n - 2}}}}}}}
 in {: fib n 40}}
```

could be spelled

```
let: (fib: (lambda: ((n: ()):
                     (if: (.: n, <: 2),
                      then: 1,
                      else: (.: (.: fib, n: (.: n, -: 1)),
                             +: (.: fib, n: (.: n, -: 2)))))))),
in: (.: fib, n: 40)
```

I think I've made it even worse, if that's possible! If we decree that `:` associates to the right, so `x: y: z` means `x: (y: z)`, we can remove all of the parentheses that neither contain a comma nor precede a colon:

```
let: fib: lambda: ((n: ()):
```

```
                  (if: (.: n, <: 2),
                   then: 1,
                   else: (.: (.: fib, n: (.: n, -: 1)),
                          +: (.: fib, n: (.: n, -: 2))))),
in: (.: fib, n: 40)
```

So far, so abysmal. But consider the above example:

```
{set! {first {: point}}}
```

Now we can write this example as

```
set!: first: .: point
```

This gives us a potential way to write semantically nested execution sequences as syntactically flat ones, but at the cost of putting our statements in the "name" field of name-value pairs, which means that there's no way to attach other data to these nodes in an unambiguous way. (This is something that is already true of, for example, argument lists.)

Consider this snippet of code from this bit of C++ 3-D ASCII-art animation:

```
  // Rotate by theta.
  nx = c*xs[i] + s*zs[i];
  zs[i] = -s * xs[i] + c*zs[i];
  xs[i] = nx;
```

It's necessary to either do the three statements in the order in which they're written or to do parallel assignment. For a moment let's disregard the much better option of parallel assignment and pretend it's a case where we have to write a sequence. Which of the following horrors is the less bad translation?

```
{do {= {nx {. {. c * {. xs at i}} + {. s * {. zs at i}}}}}
 then {do {. zs at i put {. {. {- s} * {. xs at i}}
                          + {. c * {. zs at i}}}}
       then {. xs at i put nx}}}
```

or

```
do: (=: nx: (.: (.: c, *: (.: xs, at: i)), +: (.: s, *: (.: zs, at: i)))):
    (.: zs, at: i, put: (.: (.: -: s, *: (.: xs, at: i)),
                        +: (.: c, *: (.: zs, at: i)))):
    (.: xs, at: i, put: nx)
```

Well, they're both pretty fucking bad compared to the FORTRAN-style code above, but I think the second one is worse.

Edge-labeled graphs: now is the winter of our discontent made glorious summer

Years ago I saw a project on SourceForge that treated text as an edge-labeled graph (similar to Suciu's UnQL unstructured query language) delimited by whitespace and structured by indentation, and

provided tools to query it and to reformat a number of Unix commands to make them more amenable to processing with it. (Unfortunately, I forget the name, and I haven't been able to find it again.) So, for example, given this input:

```
time    real    0m1.694s
        user    0m1.524s
        sys     0m0.168s
```

it would decide that from the start node there was an arc labeled "time", and from the node it led to three more arcs labeled "real", "user", and "sys", each of which led to a node with one further arc labeled with a string such as "0m1.524s". This of course means that you can easily query `time.user` and get that response.

(In a tree, it's immaterial whether the labels are on the arcs or on the nodes they lead to, but in a more general graph it can matter. It might make more sense to think of all of the following as having tags on nodes rather than edges.)

Note how this generalizes a subset of the finite-map-tree model: the names are just strings, as in JS and Perl, rather than general objects, as in Python, Lua, and Clojure, but there's no distinction between keys and values — the values are just the labels after the level where your query stopped traversing edges. Also, the keys need not be unique. (They may or may not be sequenced; those are different variants of the data model.)

Suppose we try to use this approach for our homoiconic language, although using parentheses rather than indentation to indicate side branches — the above graph comes out as `time (real 0m1.694s) (user 0m1.524s) sys 0m0.168s`.

The grammar here is something like

```
tree ::= [ \n]* ("(" tree [ \n]* ")" | [-A-Za-z0-9*]+) tree | ε
```

Here the three alternatives amount to three different ways to grow a tree: by branching it, by extending a branch by a segment, and by terminating a branch. This apparently has one more alternative than the original `sexp` grammar given above, but this is an illusion; the `sexp` grammar contained `sexp*`, which hides an alternative in its Kleene closure.

A homoiconic language using edge-labeled graphs?

In a sense, this is similar to the Lisp cons rotated 90°: nesting (car) is the default and parentheses turn it off! But let's say that the sequence of branches is not important, so the above example is equivalent to say `time (user 0m1.524s) (sys 0m0.168s) real 0m1.694s`.

Sequencing is now easy, but what do expressions look like?

With this approach, we could use sequencing not only for imperative statement sequences but also for argument lists. A sequence of statements might look like `(some action) ; (some other action) ; a third action`, with `;` edges connecting the sequence of statement nodes.

And the problem we previously had with difficulty determining what operation to invoke is gone — the label leading us into an expression node can be the variant tag that tells the interpreter

unambiguously how to handle that expression — or, as in Common Lisp, either an identifier of a special form or the name of a function. This suggests that, as in Forth, variables (and perhaps constants) are just zero-argument functions, but in this case — unlike in Forth — they can look to see if they're being invoked with arguments, such as maybe `=`. (And you can of course have a FUNCALL function as in Common Lisp or a `value` message as in Smalltalk, so if you store a function pointer in a variable, you can still invoke it.)

The fundamental benefit of property-list-like systems is that you can always attach new "properties" to some well-defined set of nodes without bothering the things that are already using those nodes, because they only look at the properties they care about. In this system this is somewhat vitiated by the fact that since property values are just arc labels, just like property names, there are inevitably a lot of nodes where this benefit does not obtain — any new property you attach at those nodes might be mistaken for the property value!

Function calls are kind of hairy

Unary operations and commutative, associative operations like `+` might conceivably just attach their arguments directly to their node: `+ (x) (y) * (a) b`, for example, for $x + y + a \times b$. But more general function calls might require named arguments `fib (n 10)` or an argument sequence analogous to statement sequences `cat2 (leaf "x") , leaf "y"`. Also, if duplicate edges are not allowed, `* (x) x` would be a problem.

What if math operators are messages sent to numbers?

As before, a possible alternative to applying (typically global and constant) functions to arguments is to send messages to objects, which would seem to allow syntax like `x + 1` or `xs (at i) put nx`. However, though the actors and closures models are formally equivalent, this poses a real problem for chains of operators of the kind we commonly see in mathematical expressions — it would seem to require the equivalent of Lisp's FUNCALL function, GlyphicScript's `;` operator, or Haskell's `$` function to separate the two operators. For example, $x + y - a \times b$ could be written, for example, as `(o (f x + y) - a * b)` or as `(x + y) $ - a * b`.

The problem with the obvious way of writing it `(x + y) - a * b` is, I think, that the root expression node has the `-` edge coming directly out of it, and we're considering here a universe where root expression nodes instead have edges coming out of them that denote message receivers, not operators, and it isn't clear who is supposed to be receiving the `-` message. Maybe it could be made to work, though, even for things like `(x + y) - (a * b) - 3` and `x + (y) - (a * b) - 3`, by making operators like `+` and `-` link together a sequence of expressions in the same way that `;` and `,` are suggested to do above.

This will inevitably lead to somewhat of an impedance mismatch with conventional mathematical precedence, as did the systematic rules of APL and Smalltalk, which may lead to bugs in programs. In this case, it might be possible to refuse to parse most expressions that have such problems, but not, for example, `a * b + c`.

How about currying?

If the difficulty only pertains to functions that must distinguish between their arguments, such as `<` or `÷`, can we solve it by currying?

In the function paradigm, this seems to require a function analogous to FUNCALL or APPLY:

```
funcall (< 2) x
```

Maybe in the message-receiver paradigm it works better?

```
x . (2 <)
```

This (equivalent to `x . 2 <`) doesn't seem promising.

The other programming constructs are simple enough

By contrast with primitive operations and function calls, there are relatively few difficulties with looping, conditionals, assignment, and function declarations.

A simple while loop poses no difficulty:

```
while (condition)
   do body
```

Perhaps we can use : as a tag for such bodies:

```
for (x in mylist): some body expression
```

More generally, looping can be written in a way quite analogous to Common Lisp, introduced with a `loop` tag and containing an unsequenced set of keyword-driven clauses:

```
loop (for i = 1 to 10)
     (for x in mylist)
     (if ((> x) (< 2)) then collect i)
```

Conditionals can be written either in a variant of the if-then-else style described earlier, with the `if` pulled out into an introductory tag, as

```
if ((< x) (> 2)) (then 1) else recursive expression
```

or in a cond sequence, since now we have a reasonable way of writing sequences:

```
cond ((condition a) -> consequent a)
   | ((condition b) -> consequent b)
   | else other consequent
```

Or

```
if ((condition a) then consequent a)
elseif ((condition b) then consequent b)
else other consequent
```

And lambda-expressions can be written with a delimiter to distinguish the body from the argument list:

```
fn (x y z) => some expression of x y z
λ (x y z) . some expression of x y z
```

Assignment could be written in a conventional way:

```
x = 3
x ← 3
x <- 3
x := 3
```

Or in a parallel-assignment way, like `letrec` or `setq`:

```
fn (a b) => (while (a): setq (b a) (a (b % a))); b
```

What if we represent branching with infix operators rather than parens?

It's rather jarring in the above that, for example, these two expressions are equivalent, even though the second looks like a typographical error:

```
while (a): setq (b a) (a (b % a))
while (: setq (b a) (a (b % a))) a
```

The fact that the associative and commutative operation of attaching two branches x and y to the same node is represented using the asymmetric syntax `(x) y` is, I think, the root of this difficulty. In a one-dimensional media it is unavoidable that we put them in some order, but we could imagine using a more visually symmetrical operator to separate the two symmetrical branches.

For example, ,, as explored briefly in the ill-fated "Flat dict syntax" section above. But we don't want to write

```
while a, do setq ...
```

because as long as comma binds more loosely than juxtaposition (as it should), that attaches the `setq` and `while` edges to the same root. Instead we get

```
while (a, do setq (b a, a b % a))
```

which seems potentially reasonable, though perhaps it gives us back the extreme nesting we were hoping to escape. It echoes Python's named-parameter syntax, but more placidly; instead of `f(g=h, i=j)` we have `f(g h, i j)`.

Maybe we should use indentation rather than parentheses to indicate side branches

A purely indentation-based version of this syntax is doable, replacing the line-noise punctuation and recursive nesting parentheses with preattentively-comprehensible horizontal juxtaposition for path concatenation and vertical juxtaposition for branching. Maybe using parentheses rather than indentation to indicate side branches wasn't such a hot idea after all!

```
while a
      do setq b a
             a b % a
```

Alternatively, with more vertical syntax:

```
while
  a
  do
    setq
      b a
      a b % a
```

These variants at last seem like they might actually be an ergonomic improvement over Lisp syntax rather than a regrettable compromise, at least if there's a solution to the problem with arithmetic expressions.

It unfortunately gets us back to the problem of representing sequences of statements with progressively increasing nesting:

```
do setq nx sum product c
              *      xs at i
          +   product s
              *      zs at i
   then do zs at i
             put sum product - s
                    *      xs at i
                +   product c
                    *      zs at i
          then xs at i
                 put nx
```

Or maybe

```
let nx sum product c
          *      xs at i
       +  product s
          *      zs at i
    in do zs at i
            put sum product - s
                   *      xs at i
               +   product c
                   *      zs at i
         then xs at i
                put nx
```

Or maybe, using , as an argument-sequencing graph label this time instead of a syntactic branching operator:

```
let nx + * c
          , xs at i
        , * s
            , zs at i
    in do zs at i
            put + * - s
                   , xs at i
                 , * c
                     , zs at i
```

```
        then xs at i
               put nx
```

This seems pretty bug-prone because it took me a couple of tries to get the `, zs at i` lines to the right indentation level.

This is less nauseatingly bloated than the earlier versions but it still compares poorly to the C++ version:

```
nx = c*xs[i] + s*zs[i];
zs[i] = -s * xs[i] + c*zs[i];
xs[i] = nx;
```

You could argue that maybe syntactic sugar can compensate, but the right syntactic sugar is precisely what I'm looking for here.

I may be asking too much, since even in Common Lisp it's still uglier and more bug-prone than the C++:

```
(let ((nx (+ (* c (aref xs i)) (* s (aref zs i)))))
  (setf (aref zs i) (+ (* (- s) (aref xs i)) (* c (aref zs i))))
  (setf (aref xs i) nx))
```

But that's not in the same league of noise bloat as most of the examples above.

We could imagine an infix-formula-evaluating macro like the ones in sh and Tcl, called, say, `[` (since `FORTRAN` would be a tasteless name, Σ is hard to type and too specific, and `eval` probably means something else):

```
let nx [ c
        * xs at i
        + [ s
            * zs at i ] ]
    in do zs at i
            put [ - s
                  * xs at i
                  + [ c
                      * zs at i ] ]
          then xs at i
                 put nx
```

However, that won't work as written; `[` has no way to tell whether you wrote `x * y + z` or `x + z * y`. If you want it to have a whole *sequence* of labels to compile, you have to put them on one line, which also means you can't inline-evaluate arbitrary bits of code like `xs at i` without some kind of magic tag. If you do it that way you could say

```
let nx [ c * [ xs at i ] + s * [ zs at i ] ]
    in do zs at i
            put [ - s * [ xs at i ] + c * [ zs at i ] ]
          then xs at i
                 put nx
```

In practice this is maybe not the best example since you would probably want array indexing in your numeric expression evaluator and also because a better way to do the whole calculation is

```
let xi xs at i
    zi zs at i
    in do xs at i
            put [ c * xi + s * zi ]
          then zs at i
                 put [ - s * xi + c * zi ]
```

or maybe even some kind of parallel assignment. I just picked an example that's too easy, I guess.

A potential problem with the proposed embedding syntax: what happens if you have branching inside the embedded expression? I mean, you could imagine something like

```
[ 2 * [ gcd a x ] + 3 ]
          b 2
```

where we invoke `gcd` with named arguments `a` and `b`, which upon some thought it can be seen can be made to work just fine — the `[` parser just needs to look down all the branches to find the terminating `]` of the embedded expression, not just one. This gets uglier if you have two such things; this will not work:

```
[ 2 * [ gcd a x ] + 3 * [ lcm a x ] ]
          b 2             b 2
```

But it will work if expressed this way:

```
[ 2 * [ gcd a x ] + 3 * [ lcm a x ] ]
                          b 2
          b 2
```

This is complex enough to be confusing.

How is that different from SRFI-49 I-expressions, Wisp, or LISPIN?

In the above proposal, as long as arcs out of a node remain unordered, these two expressions are equivalent:

```
a b c
   d
  e
```

and

```
a e
  b d
    c
```

as well as two more variations. Moreover, as mentioned toward the beginning, code written for the following structure will also work on the above structure without change:

```
a b c
```

An unpolished parser in Python

I just wrote the following simple parser in Python which seems to

handle the syntax outlined above properly and translates it into graphviz files you can view with, for example, `dot -Tx11`. It's maybe 50% longer than the S-expression parser above in Lua. It contains some duplication to factor out, would need to be extended to handle quoted strings, can break its graphviz output if you put special characters in the input, wastes memory, doesn't handle tabs, and (as always with Python) breaks on Unicode input in environmentally dependent ways, but hopefully it represents some kind of clarifying sketch.

```
from __future__ import print_function
import re
import sys

def parse(lines):
    stack = []
    node_counter = 1
    edges = []

    for line in lines:
        col = len(re.match(r'\s*', line).group(0))
        while stack and stack[-1][0] >= col:
            stack.pop()
        word, start, empty = [], col, ()

        while col < len(line):
            c = line[col]
            if word and re.match(r'\s', c):
                nw = ''.join(word)
                word[:] = empty
                if stack:
                    edges.append((stack[-1][2], nw, node_counter))
                else:
                    edges.append((0, nw, node_counter))
                assert start is not None
                stack.append((start, nw, node_counter))
                node_counter += 1
                start = None

            elif re.match(r'\S', c):
                if not word:
                    start = col
                word.append(c)

            col += 1

        if word:
            nw = ''.join(word)
            if stack:
                edges.append((stack[-1][2], nw, node_counter))
            else:
                edges.append((0, nw, node_counter))

            assert start is not None
            stack.append((start, nw, node_counter))
```

```
            node_counter += 1
            start = None

    return edges

def graphviz(edges, name='cosas'):
    yield 'digraph '; yield name; yield ' {\n'
    yield '    rankdir=LR;\n'
    yield '    node [label="", shape=circle];\n'
    for start, label, end in edges:
        yield '    '; yield str(start); yield ' -> '; yield str(end)
        yield ' [label="'; yield label; yield '"];\n'
    yield '}\n'

if __name__ == '__main__':
    sys.stdout.writelines(graphviz(parse(sys.stdin)))
```

So for example it renders the first example above as follows:

```
digraph cosas {
    rankdir=LR;
    node [label="", shape=circle];
    0 -> 1 [label="while"];
    1 -> 2 [label="a"];
    1 -> 3 [label="do"];
    3 -> 4 [label="setq"];
    4 -> 5 [label="b"];
    5 -> 6 [label="a"];
    4 -> 7 [label="a"];
    7 -> 8 [label="b"];
    8 -> 9 [label="%"];
    9 -> 10 [label="a"];
}
```

Since it took me about 40 minutes to write, test, and (mostly) debug that, including the graphviz output, this syntax is probably not too complex for a language whose first-draft compiler you want to write in an afternoon.

Functions for manipulating edge-labeled graphs

What is our equivalent of the ur-Lisp's CAR CDR CONS NULL ATOM QUOTE EQ? The fundamental traversal operation should presumably be `go(node, tag)`, which returns the node (if any) obtained by traversing the edge labeled `tag` from `node`, equivalent to `node[tag]` in JS or Lua syntax; if duplicate edges are allowed, probably both its `node` argument and its return value should be *sets* rather than individual nodes, and possibly `tag` should also be a set. Invoking the tag as a function `tag(node)` is an alternative possibility.

(If they are sets of nodes, we need to be able to iterate over them; we need to be able to test whether they are empty, but iterating over them with side effects may be an adequate interface for that.)

In the case where they are indeed individual nodes, there is the possibility of returning nil, in which case we need a way to detect nil — in a functional paradigm, `isnil(node)`, but alternatives include treating nil as false in conditionals (making the function implicit); a

pattern-matching `ifnil(node, consequent, alternate)` function which takes two functions to invoke, one with the node if it is not nil, and the other if it is; and the λ-calculus-like `node(consequent, alternate)`, which does the same without a separate function.

It's also necessary to iterate over the edges out of a node, since arbitrary values such as 57 are also stored as tags in the above model. (This wouldn't have to be the case — 57 could be a node, as it is in Lisp — but without this ability to iterate over edges you also can't write PRINT or the macro transformer for LETREC.) You can do this with a function `kids(node)` or `pairs(node)` which returns a list in the usual car–cdr or first–rest form; perhaps each node in the resulting list contains both a key — the edge label — and a value — the child node that it leads to. (That's superfluous, though, since `go(parent, tag)` will give you the child node.)

If we consider tags to attach to nodes rather than to the edges leading to them, we might be able to conflate nodes with tags, so that the `go()` function above takes two arbitrary node arguments. (The alternative is to have special "tag nodes".) But we still need to be able to compare tags for equality, thus `eq(tag1, tag2)` or `sametag(node1, node2)` is needed.

With `go`, `isnil`, `kids`, and `eq` (or the other alternatives discussed above), we can traverse the graph as we please, just as with CAR, CDR, NULL, ATOM, and EQ. However, constructing new graph structure — as with CONS — requires another operation: `add(node, tag, kid)`, which returns a new node identical to `node` except that it now has an edge with tag `tag` to node `kid`. However, this is not quite enough — it doesn't allow us to produce new nodes with no children. So we need `new()` to produce such a fresh node.

And of course we can provide QUOTE (and, more interestingly, quasiquote) just as Lisp does. That's what homoiconicity means!

Topics

- Programming (p. 3658) (286 notes)
- Python (p. 3671) (27 notes)
- Compilers (p. 3383) (16 notes)
- JS (p. 3533) (12 notes)
- Bootstrapping (p. 3348) (12 notes)
- Lisp (p. 3552) (9 notes)
- Scheme (p. 3694) (8 notes)
- OCaml (p. 3602) (8 notes)
- Lua (p. 3556) (5 notes)
- LuaJIT
- Clojure
- Cant

Regenerator gas kiln

Kragen Javier Sitaker, 2016-09-05 (updated 2017-04-10) (9 minutes)

I want to start firing ceramics at home, but the mini-kilns I'm finding on MercadoLibre suck shit. They're electric and top out at low temperatures like a bit over 1200°. (Maybe this is a consequence of their heating elements?) Although earthenware can fire at as low as 1000°, temperatures in the 1350° range are needed for many other ceramics.

For example, in July, http://articulo.mercadolibre.com.ar/MLA-618696798-horno-para-vidrio-ceramica-gres-box-1-_JM was a 1200-watt kiln with a 160 mm × 160 mm × 115 mm inside space (2.8 ℓ) that cost US$500. And it only reached 1200°, but weighed 6 kg.

I was thinking that maybe a gas-fired kiln would work better, maybe fueled by a portable LP gas cylinder and with regenerators to keep noxious exhaust to a minimum.

I want to beat that shitty electric kiln handily on all axes except weight: energy usage, volume, temperature, and cost. I'm okay if it weighs a lot more, though. Let's shoot for a 320 mm cube inside capacity, 1400° max, cost under US$200, weight under 100 kg, and use under 600 W.

(See files Millikiln (p. 2581) and An electric furnace the size of a sake cup (p. 666) for versions of this with further reduced scope.)

Energy and power usage and efficiency

The gas heater we use to heat the living room is 8000 BTU/hour, which is 2300 watts. The little electric space heater I'm using to heat my office is 2000 watts. Suppose 2000 watts is a reasonable power envelope and we need to be able to heat at 300° per hour (0.083 K / second) once the ceramic is dry. Then the maximum thermal mass in the kiln is 24 kJ/K. The specific heat of water is 4.2 kJ/K per kilogram, so this is about 6 kilograms of water; other substances like quartz have lower specific heats like 0.83 kJ/K/kg, so it's in the neighborhood of 30 kg. This is pretty close to the amount of stuff you'd like to have on the inside of a kiln's insulation.

(Engineeringtoolbox lists specific heats of .70, .73, and .83 kJ/K/kg for different forms of quartz.)

This implies that you need pretty high efficiency; we can't afford to just spew out the vast majority of your heat in an exhaust and expect to get the kind of temperatures we need.

How much energy do you need at a minimum? The minimum would be when you heat your kiln up inadvisably fast. At 300° per hour (83 mK/s) you can reach 1200° in four hours, which would be 2000 W · 4 hours = 29 MJ. LP gas is 46.4 MJ/kg or 26 MJ/ℓ, so this is 630 g of gas, occupying 1.12 ℓ. A regular 10 kg gas cylinder would power 15 such firings before refilling, a cost of about US$6 per firing.

This doesn't take into account the heat lost while the kiln is hot, which depends on insulation and regenerator losses, or the energy lost in boiling off the water, which is about 2.3 kJ/kg and thus relatively small in this context.

Using regenerators in this context involves feeding the flame with

air preheated from the regenerators. LP gas burns at 1970° in cool air, which is hot enough, and presumably this temperature increases as the inlet temperature increases. This means, though, that the chamber where the flame is happening will need to be able to withstand such temperatures. Some amount of exhaust gas recirculation may reduce the load on the regenerators and lower the flame temperature.

You probably don't need to run the flame the whole time; the autoignition temperature of LP gas is around 400° or 500°, so you should be able to turn the gas back on and ignite it just from the walls of the burner chamber.

You need two regenerators because you need to pull the intake air through one while running the exhaust air through the other, periodically alternating direction. The use of a recuperator instead of regenerators would avoid this necessity, conventional (non-fractal) recuperator designs need to be fabricated from high-thermal-conductivity materials such as copper or aluminum, which can't withstand such high temperatures. (Copper melts at 1085°.)

What regenerator material can withstand 1350°, though? Lots. Alumina melts at 2072° and even quartz doesn't melt until 1670°, even though quartz-based felsic lavas and well-fluxed quartz-based glasses melt at much lower temperatures. Regular fireclay is good to 1515°. So a pebble bed of ordinary fireclay should work fine. Limestone would calcine to quicklime, which would also work, and would be a useful product in its own right, though it might be an unnecessary hazard. Quicklime could also perhaps eliminate some acid contaminants from the exhaust gases, such as HCl from salt firing.

Olivines are not suitable for the regenerator itself; although you might expect them to be much more refractory due to their role in raising the melting temperatures of mafic rocks, not to mention their foundry use as refractory sands, not only do some of them melt as low as 1200° (fayalites, though forsterites melt at higher temperatures up to 1900°), but they oxidize exothermically to quartz and magnesite when exposed to hot CO_2. This might be useful to get a zero-carbon-emission gas-fired kiln if the reaction is fast enough; the exhaust air could contain less CO_2 than the input air.

If the alternation time on the regenerators is 1 minute and the advection reaches 2000 watts (that must be retained), each regenerator must hold 120 kJ. If the temperature swing in the regenerator averages 600° (more for the innermost chunks, less for the outermost), then it would be sufficient to have 250 g of quartz per regenerator at 0.83 kJ/K/kg. This is very promising!

Let's suppose that we want the kiln to stay at 1350° for 12 hours without losing more than, say, 10% of the heat energy through its insulation, so that we run the burner at about 10% power to keep it warm. Earlier I said the inside should be a 320-mm cube, which gives it 0.61 m² surface area. R-values are m²·K/W, and in this case we want to lose under 200 W through 0.61 m² and a 1330-K difference, which means our R-value needs to be at least 4.1. Fiberglass is about 22 m·K/W, so that's about 190 mm of fiberglass. Loose-fill perlite is about 19 m·K/W, loose-fill vermiculite can be as low as 15 m·K/W, so you might need as much as 275 mm of vermiculite. This is eminently feasible, although it doesn't leave a lot of extra room.

NIST's data

http://ws680.nist.gov/bees/ProductListFiles/Generic%20Fiberglass.pdf says R-13 fiberglass batts (i.e. 13 m²·K/W, which is 3.2 times what we need) are 89 mm thick and weigh 12.1 kg/m³ (i.e. 12.1 g/ℓ). This works out to be 146 m·K/W, which is 6.6× as high as the value I calculated above and twice as high as the values I find online for silica aerogel, so it cannot possibly be correct.

Ah, this random CertainTeed™ brochure http://www.certainteed.com/resources/IGProductKnowledge20060008.pdf explains that "R-13" is actually 2.3 m²·K/W, giving a thermal resistivity for fiberglass of 25.8 m·K/W, which is correct. So we can probably take the density as correct.

Using such batts (minus their facing layer, I suppose) to achieve the insulation level needed would require two layers of them, providing 4.6 m²·K/W of insulation and increasing the dimensions of the kiln by 178 mm on each side, for a total of (178+320+178) mm = 676 mm. The total mass of the fiberglass then would be 3.3 kg, entirely reasonable.

But you can't actually use just those batts, because the innermost layers of insulation need to withstand 1400°, and glass won't. Glass will only handle up to 500° or so, so only the outer reaches can be made of glass. You need inner layers of kiln bricks or vermiculite or something, and that's going to drive up the weight considerably. (Or you could use ceramic fiber batts, which look like glass fiber batts but are made of alumina to withstand high temperatures.)

To observe the inside of the kiln without losing a ridiculous amount of heat, although peepholes are the traditional solution, you might want some kind of optics. I have no idea how this could work; wouldn't a mica window melt? Even if it doesn't melt, could you see through it while it's glowing?

Topics

Zombie contingency plan

Kragen Javier Sitaker, 2017-07-19 (9 minutes)

Suppose there were a zombie apocalypse (or similar event, such as a war or deadly epidemic). What could I do?

I live in a Buenos Aires apartment with two other people, about 29 meters from the street. If we could, it would be better not to leave the apartment if possible, preferring to conduct day-to-day life behind closed doors as long as possible.

Let's suppose for the time being that we don't need to provide our own air the way nuclear submarines and space stations do. What about the other necessities — food, water, medicine, warmth, coolth?

We have about 9m² of roof terrace and about 50m² of indoor space with 4m ceilings, a total of 200m³ of indoor space.

Food

A food reserve probably needs 2500 kcal per person per day for each of the two adults, plus 1500 kcal for the seven-year-old, a total of 6500 kcal/day. If we want it to last a year, that's 2.4 million kcal; if 40% of the calories are carbohydrate, 30% fat, and 30% protein — the desirable balance used in e.g. Clif bars — then that's 950 thousand kcal of carbohydrate and 700 thousand kcal each of fat and protein. At 5 kcal/g, that's 190 kg of carbohydrate and 142 kg of protein; at 9 kcal/g, that's 79 kg of fat. The total is 411 kg of dry mass, or probably 500 kg of dry food, probably occupying about 500 ℓ, or about 250 two-liter plastic coke bottles, if it's stored in that form.

We currently cook the food using the municipal natural gas supply; off-grid alternatives in case of catastrophe would include burners driven by LP gas cylinders, like the one that recently exploded the house of a friend of mine; alcohol burners; and solar cookers.

Lentils, as a representative cheap food, cost AR$5.20/kg in 10-kg bags. So 500kg of food would cost about AR$2600, which is currently about US$173. Of course, you wouldn't buy 500 kg of lentils; you would buy a mix of different cheap foods. But this would be super cheap.

Climate control

Right now I have a portable air conditioner set up in the bathroom drying my roommate's sheets and towels. Dry laundry is essential to maintaining body temperature when it gets cold and damp. This machine uses 6 amps of 220-volt single-phase AC and can evaporate and recondense between 1ℓ and 10ℓ per hour (I haven't measured carefully). (It also provides heat and cool when needed, although right now we're heating the apartment rather expensively with a gas heater run off the municipal natural gas supply.)

Devices like this one cost about AR$5000 = US$333. It would be good to have a spare.

However, running it requires electricity. Maintaining electricity in the event of a zombie apocalypse would require being able to generate it locally.

Electricity

We could drive the air conditioner off some car batteries and solar panels with an inverter, ideally on a pallet in a plastic tray under an awning on the roof terrace, so that if shit goes wrong it doesn't blow up the house. But how much would we need?

220V × 6A = 1320W; 1320W ÷ 12V = 110 A. So we'd need batteries capable of 110 amps, which is easy enough as far as it goes, but we probably also need to be able to run it for a couple of hours at a time, which works out to 9.5 MJ or 2600 watt-hours. That suggests we need something like three deep-cycle batteries like the Trojan 27TMX, which holds 3.2 MJ by my calculations and can supply 530 amps. This gives the electrical system a "burst" capacity of 19 kilowatts.

These batteries cost AR$5100 each, so this is AR$15300 of batteries. Probably you'd actually want four so that you can put them in series strings of two batteries to get 24V, for a total of AR$20400.

There are 2000W inverters on MercadoLibre for AR$7500; they require 24V input.

Such an electrical system could also be used to run a microwave for cooking, lights, even electric stove burners.

Somehow you'd also need to charge the batteries. A 250-watt (peak) solar panel is 1.62 m² and costs AR$7000. Since this is 5.3 times less power than the air conditioner, a single one of these would require 5.3 hours of midday sunlight to recharge from a single hour of air conditioner use; it would be better to have two of them, for 500 watts peak and AR$14000. The roof has space for about six of them at most.

You also need a charge controller, which I think is about AR$1000.

So the total is AR$20400 + AR$7500 + AR14000 + AR$1000 = AR$42900 = US$2860.

Water

Buenos Aires gets 1200 mm of rain per year, which is distributed fairly evenly through the year. That means our ≈9m² of roof terrace gets about 10.8 m³ of rain per year, or 10,800 liters; on average, that's almost 30 ℓ per day, or 10 ℓ per person per day. Even at Burning Man, you only need about 6 ℓ per day per person, and you lose less water when you're not in the hot sun in the desert.

So even a cistern system would work adequately, given adequate filtration and other measures against contamination. Transparent awnings covering 60% of the small terrace area would be adequate. Probably a month's worth of water is adequate; that would be 6 ℓ × 3 × 30 = 540 ℓ. 500-liter drinking water tanks are readily available and cost a bit under AR$1000 = US$67.

Additionally, though, the air conditioner I mentioned earlier condenses water from the air, somewhere between 1ℓ and 10ℓ per hour. (If we figure that all of the 3000 kcal/h of cooling provided by a machine like this are provided by the condensation of water, whose heat of vaporization is 2257 kJ/kg, it would be 5.6 ℓ/h.) That means we can get water even without the cistern. It seems to be a bit jelled, though, and tastes funny; I suspect it may have some kind of microbial slime in it.

This calls for a 400× microscope with slides (AR$1000 = US$67)

and a water tank.

Access control

Of course none of this is useful if the zombies can get in and bite us. The walls are concrete and therefore fairly zombie-proof, but it would be good to replace the apartment's front door with a metal door, and add another metal door at the bottom of the stairwell, since an inhabitant of any of the other dozen or so apartments in our passage could get bitten and become a zombie. The stairwell is shared between only three apartments and provides us access to our roof terrace.

"Armored" doors from vendors like Pentagono cost on the order of AR$10000, and we would need three of them. This costs AR$30000 = US$2000.

(We have wood floors, but I think there may be a concrete subfloor.)

Other safety equipment

We need fire extinguishers, probably three of them. These cost about AR$700 each, for a total of AR$2100 = US$140.

Carbon monoxide detectors and flammable-gas sniffer alarms would be a good idea.

Communication and interchange

A catapult launching guided gliders from the roof would make it possible to get a look at what's going on in the area and transmit radio signals to the rest of the city; the glider could return to the terrace to land, using a parachute to effect a vertical landing. If there are other survivors of the zombie apocalypse, we could send them small items.

Weapons

What, are you kidding? You can't kill zombies. Weapons are pointless. Just hide.

Total cost

```
US$ 200 food
US$ 666 two portable air conditioners (I already have one)
US$2860 solar electric system
US$  67 water tank
US$  67 microscope
US$2000 armored doors
US$ 140 fire extinguishers
-------------------------
US$6000 total
```

Topics

• Pricing (p. 3646) (89 notes)
• Independence (p. 3520) (63 notes)
• Energy (p. 3438) (63 notes)

Why the Cartesian product of fields isn't a field

Kragen Javier Sitaker, 2019-05-02 (2 minutes)

If you have some underlying field like GF(2) you might think you could derive an infinite variety of other fields from it by way of taking arrays of some finite size which operate elementwise.

For example, GF(2) is just bits, using AND as multiplication and XOR as addition; considering the field axioms, these are associative, commutative, have identity, distribute (a & (b ^ c) = (a & b) ^ (a & c)) and have inverses (each element is its own inverse, trivially so in the case of AND: 1 & 1 = 1), so that's a finite field. You might think that you could extend this elementwise to "bitvectors" (not in the sense of a vector space, just in the sense of arrays of bits) but this fails when we get to the inverse: AND with a "bitvector" containing zeroes is information-destroying, so there can be no inverse. In the single-bit case, we get to escape by pleading division by zero, but not in the multibit case.

So "bitvectors" of some size form a commutative ring with unity, but not a field.

There *is* a field of size 2^8, though. It's GF(256), which is not just $\mathbb{Z}/256\mathbb{Z}$, arithmetic modulo 256, as you might think — that's, again, just a commutative ring with unity, since there are plenty of pairs of numbers that multiply to zero, like 2·128, so not every member has a multiplicative inverse. No, in some sense it *is* a vector of 8 bits, but I don't understand the construction of the operations; it's some kind of construction with monic irreducible polynomials.

Topics

• Math (p. 3564) (78 notes)

Real-time bokeh algorithms, and other convolution tricks

Kragen Javier Sitaker, 2018-12-18 (updated 2019-08-15) (23 minutes)

It would be super neat if I could blur a video stream in real time with a realistic camera-like bokeh, like a flat circle or hexagon or octagon or something, with a diameter in the neighborhood of like 32 pixels or more.

By "flat" I mean that, within the area of the bokeh, the output transfer function is almost constant. (And, outside, it's zero.) This is very different from the Gaussian blur that is such a popular effect on computers, which has the advantage that it's isotropic (the PSF/OTF is circularly symmetric) and very fast to compute (as a cascade of box filters and/or separable X and Y Gaussian blurs). It's the *only* circularly symmetric separable convolution kernel.

On my laptop, the time budget for fullscreen full-resolution 60-fps video is 8 ns per pixel. So I think this is probably feasible on a GPU with enough work, but not on a CPU.

However, I have found some algorithms that are probably only *slightly* too slow, and are much faster than the state of the art.

Candidate approaches

Downsampled bokeh computation

Some of the algorithms described below may produce a bokeh whose edge is unrealistically sharp, or sharp in a way that is undesirable under some circumstances, or has visible aliasing artifacts. One way to ameliorate these problems is to downsample the image (using an algorithm more sophisticated than nearest-neighbor, but even decimating a first-order box filter ought to be sufficient), compute the bokeh, and then upsample the computed bokeh (with at least a second-order box filter). This of course also reduces the computational load of computing the bokeh itself.

Box filters

A box filter is pretty fast (using a prefix sum†), and it has the desirable camera-like flatness, but I've never seen a camera with a rectangular bokeh. You could use a set of 32 one-dimensional box filters for the bokeh on 32 scan lines, adding together the results, and that would work, but it might be pretty slow. (However, see the section below about "chord decomposition" for ways to improve this approach.)

And of course you can affinely texture-map the image to rotate it, box-filter the rotated version, and then texture-map it back to unrotate it. And that can get you a bokeh of a rotated rectangle or indeed an arbitrary parallelogram.

But is there some efficient way to get even, say, a paraxial trapezoid? With, like, less than a separate box filter for every 2–4 scan lines? (Yes, see the section below about efficient convolution with horizontal trapezoids.)

If you have successfully achieved two bokeh filters, one of which is

entirely contained in the other, you can subtract one from the other to get a bokeh with a hole in it.

You can get a spatially varying somewhat trapezoidal bokeh by a variation of the rotation method: do the texture mapping with perspective rather than just an affine transformation. This is a useful approach to get a bokeh of spatially varying size in general: spatially transform the image, apply the bokeh, and transform it back.

(See also the McIntosh et al. paper below for a hack that gives incorrect, but visually plausible, results for many interesting bokehs.)

Rounded corners by composing with hollow-circle convolution

A possibly useful approach to round the corners of the bokeh without blurring its edges much: add a stage to the bokeh pipeline whose PSF is a small hollow circle, e.g., this 28-pixel 11×11 circle, using . for 0:

```
. . . . 1 1 1 . . . .
. . 1 1 . . . 1 1 . .
. 1 . . . . . . . 1 .
. 1 . . . . . . . 1 .
1 . . . . . . . . . 1
1 . . . . . . . . . 1
1 . . . . . . . . . 1
. 1 . . . . . . . 1 .
. 1 . . . . . . . 1 .
. . 1 1 . . . 1 1 . .
. . . . 1 1 1 . . . .
```

This can be achieved with the difference of two one-dimensional box filters (using prefix sums) per distinct scan line. Note that in this case there are only four distinct scan lines, although there are 11 scan lines in total.

However, there's a more efficient way to compute this. We can divide the circle into slices of similar scan lines, then compute these slices with the composition of some sparse kernels without using prefix sums. For example, consider this slice of the above circle:

```
. 1 . . . . . . . 1 .
. 1 . . . . . . . 1 .
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
. 1 . . . . . . . 1 .
. 1 . . . . . . . 1 .
```

This convolution can be computed efficiently by the following composition of convolution kernels with three additions per pixel, plus an appropriate change of coordinates:

```
1 ∘ 1 . . . . . . . 1 ∘ 1
1                       .
                        .
                        .
                        .
```

The composition of the transposes of these kernels compute the transposed slice:

```
. . 1 1 . . . 1 1 . .           1
. . . . . . . . . . .           .
. . . . . . . . . . .           .
. . . . . . . . . . .           .
. . . . . . . . . . .  = 1 1 ∘  .  ∘ 1 . . . 1
. . . . . . . . . . .           .
. . . . . . . . . . .           .
. . . . . . . . . . .           .
. . 1 1 . . . 1 1 . .           1
```

The central slices also need three additions per pixel:

```
1 . . . . . . . . 1     1
1 . . . . . . . . 1  =  1  ∘  1 . . . . . . . . 1
1 . . . . . . . . 1     1
```

This might be useful in combination with some of the jaggy approaches, and an alternative that might work well for that in the case of an octagonal bokeh might be a 45° rotated hollow square, achieved through a difference of rotated box filters.

This works out to 3 and 3 additions for the horizontal and vertical central slices and 3 and 3 additions for the other two slices, plus 3 more additions to sum them all together, a total of 15 pixel additions.

Brute-force: Fourier convolution!

Finally, maybe doing the bokeh convolution in frequency space is the best available option. However, on my laptop's CPU, an FFT of a 640×480 grayscale image followed by an inverse FFT takes 96 ms, which is 312 ns per pixel. Much smaller FFTs should be significantly faster, but slightly larger FFTs (like a screenful) should be slightly slower (though for some reason 1024×1024 is actually often much slower on my laptop, possibly because of other CPU-intensive things I'm running).

Chord decomposition and per-scanline box filters

Consider a filled version of the 11-pixel hollow circle above:

```
. . . . 1 1 1 . . . .
. . 1 1 1 1 1 1 1 . .
. 1 1 1 1 1 1 1 1 1 .
. 1 1 1 1 1 1 1 1 1 .
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
. 1 1 1 1 1 1 1 1 1 .
. 1 1 1 1 1 1 1 1 1 .
. . 1 1 1 1 1 1 1 . .
. . . . 1 1 1 . . . .
```

If we wanted to convolve with that filled circle, we could do it by

convolving the following 12×11 sparse kernel (see Sparse filters (p. 834)) with a horizontal prefix sum of the pixels, thus applying a one-pixel-tall box filter to each scan line:

```
 . . . .-1 . . 1 . . . .
 . .-1 . . . . . . 1 . .
 .-1 . . . . . . . . 1 .
 .-1 . . . . . . . . 1 .
-1 . . . . . . . . . . 1
-1 . . . . . . . . . . 1
-1 . . . . . . . . . . 1
 .-1 . . . . . . . . 1 .
 .-1 . . . . . . . . 1 .
 . .-1 . . . . . . 1 . .
 . . . .-1 . . 1 . . . .
```

In this form this would require 22 additions and subtractions per pixel, but this, in turn, can be decomposed into pipelines of sparse kernels in a very similar way to that applied to the hollow circle above:

```
 .-1 . . . . . . . . 1 .
 .-1 . . . . . . . . 1 .                                  1
 . . . . . . . . . . . .                                  .
 . . . . . . . . . . . . = -1 . . . . . . . . 1 ∘ 1 ∘ .
 . . . . . . . . . . . .                        1    .
 .-1 . . . . . . . . 1 .                                  1
 .-1 . . . . . . . . 1 .
```

I think that, decomposed in this way, this circle decomposes into four kinds of scan lines; from the top, these require 2, 2, 3, and 3 per-pixel additions and subtractions each, plus the addition per pixel needed to compute the prefix sum. This works out to 11 operations per pixel, which is pretty good for a 121-pixel convolution kernel.

This is closely analogous to the Urbach–Wilkinson algorithm for erosion and dilation described in Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216), with a few differences: Urbach and Wilkinson need to use a cascade of exponentially growing kernels rather than a simple prefix sum because of the inverseless nature of the max and min operations; and, because summation is not idempotent the way max and min are, overlap is significant to convolution in a way that it is not to the morphological operations. For example, 1 1 1 1 ∘ 1 0 0 1 = 1 1 1 1 1 1 1 when it comes to erosion or dilation, but 1 1 1 2 1 1 1 for convolution.

This approach generalizes to arbitrarily-shaped flat bokehs using, at most, two subtractions and additions per "chord" (Urbach and Wilkinson's term) of the convolution kernel. This should scale adequately to things like pentagonal bokeh.

In some cases, DAGs, rather than strict pipelines, of sparse kernels may be advantageous. For example, it wouldn't be unusual to have two blocks of lines that were both 3 pixels tall; a single vertical 1 1 1 kernel applied to the horizontal prefix sum can be shared among both of them. And consider the kernel 1 0 0 0 1; this can be implemented as `p[x] += p[x+4]`, and in this form it represents a convolution which could, for example, convert a convolution with 1 0 0 1 into a convolution with 1 0 0 1 1 0 0 1. But if instead we do `p[x] += q[x+4]`, we can get the pixels we're adding in from some other image — if

previously p is the convolution with 1 0 0 1, as before, and q is the original image, then this will convert p into the convolution with 1 0 0 1 1. Furthermore, I suspect it may be the most efficient way to compute that convolution.

Above I suggested computing bokeh on a downsampled version of the image to get blurry edges. An alternative is to compute the bokeh at full resolution, but using a jaggy approximation to a circle made out of a stack of rectangles, say 3 pixels tall, using 2-D box filters on a 2-D prefix sum; then a final blur pass might hide the worst of the resulting jaggies.

If the bokeh kernel has reflection symmetry around a vertical axis rather than a horizontal axis — for example, a pentagon with a point on top or on the bottom — it may be worthwhile to do all these computations using a vertical prefix sum in order to take advantage of that symmetry.

Efficient convolution with (some) horizontal trapezoids using prefix sums

Consider this convolution kernel:

```
-1 . . . . . . . 1
 .-1 . . . . . 1 .
 . .-1 . . . 1 . .
 . . .-1 . . 1 . . .
```

Composed with a horizontal prefix sum, this kernel computes the convolution with a 4-scanline-high trapezoid whose top and bottom are horizontal. But the tricks used above to divide circles into slices will avail us nothing here. However, barring overflow, the convolution with that kernel is clearly equivalent to the sum of convolutions with these two kernels:

```
-1 . . .      . . . . . . . . 1
 .-1 . .      . . . . . . . 1 .
 . .-1 .  +   . . . . . . 1 . .
 . . .-1      . . . . . 1 . . .
```

And this can be computed more efficiently as follows, with five additions and subtractions per pixel, plus some changes of coordinates:

```
                . . 1           1 . .
. . . . . . . 1  ∘  . . .  -  1 .  ∘  . . .
. . . . . . 1 .    1 . .      . 1     . . 1
```

This trick will work for trapezoid boundary lines of some angles, but maybe others are more problematic. Consider this decomposition to five per-pixel additions, similar to some of those discussed in Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216):

```
1 . . .
1 . . .                   1 . .
1 . . .       . .         . . .
. 1 . .       . .   1     . . .
```

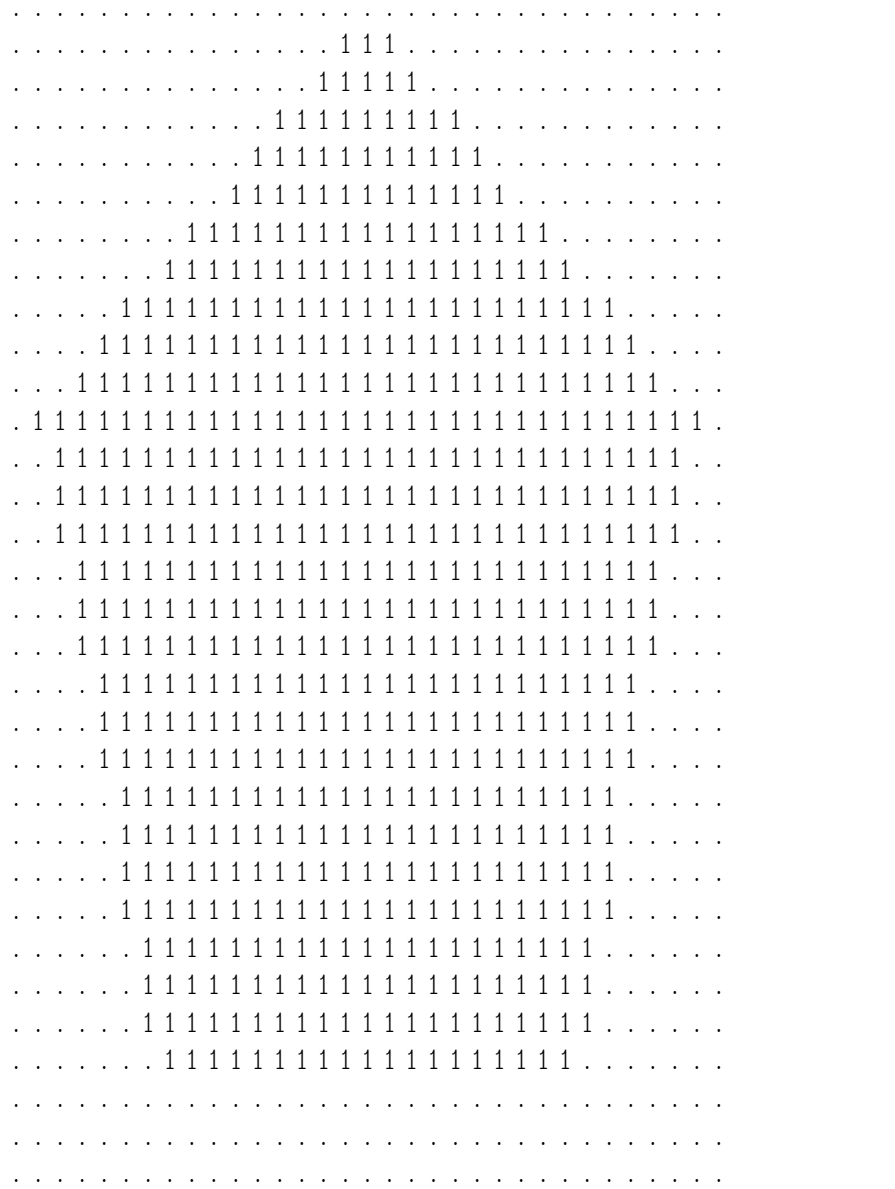
```
. 1 . .  =  (  . . + 1  ) ∘ . . .
. . 1 .        . 1   1      . . .
. . 1 .        . 1          . . 1
. . 1 .
. . . 1
. . . 1
```

Very nice, right? At first I thought you were fucked if the last scan line of the kernel is missing, but you can handle that just by subtracting it off. I don't have a fully general algorithm for finding efficient decompositions of this kind.

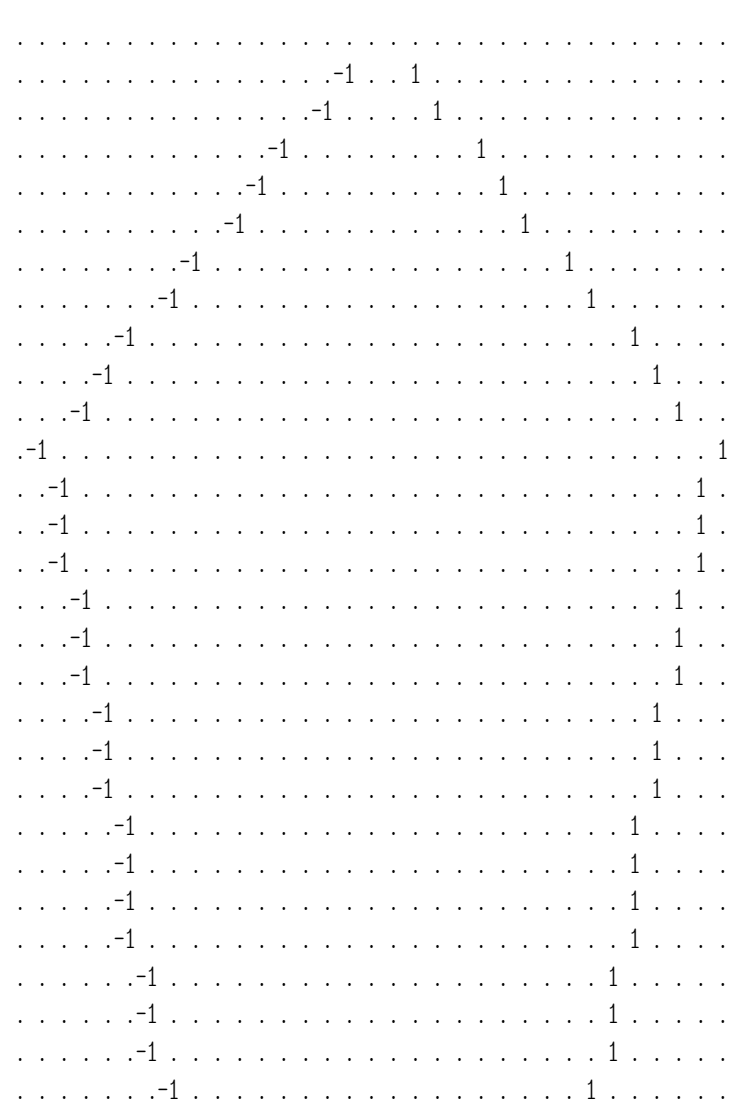
Since you can decompose any polygon into horizontal trapezoids in this fashion, using at most one trapezoid per polygon vertex, this potentially provides an efficient way to convolve with arbitrary flat polygonal bokeh. In particular, a pentagon with a point pointed up or down can be decomposed into two trapezoids, one degenerate in the sense that it is a triangle.

Convolving with a 32×32 pentagon with 600 pixels in 26 additions and subtractions

Here's a regular pentagon computed to fit into a 32×32 square:

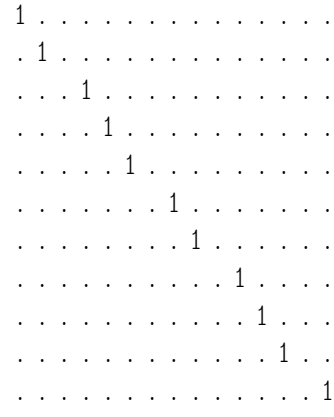


And here is the sparse delta kernel you would need to compute the convolution of the above with an image from a horizontal prefix sum of that image:

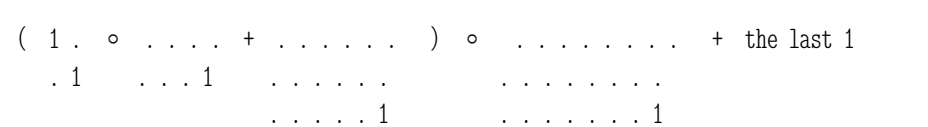


Since the left and right edges are the same in mirror image, I will just deal with the right edge.

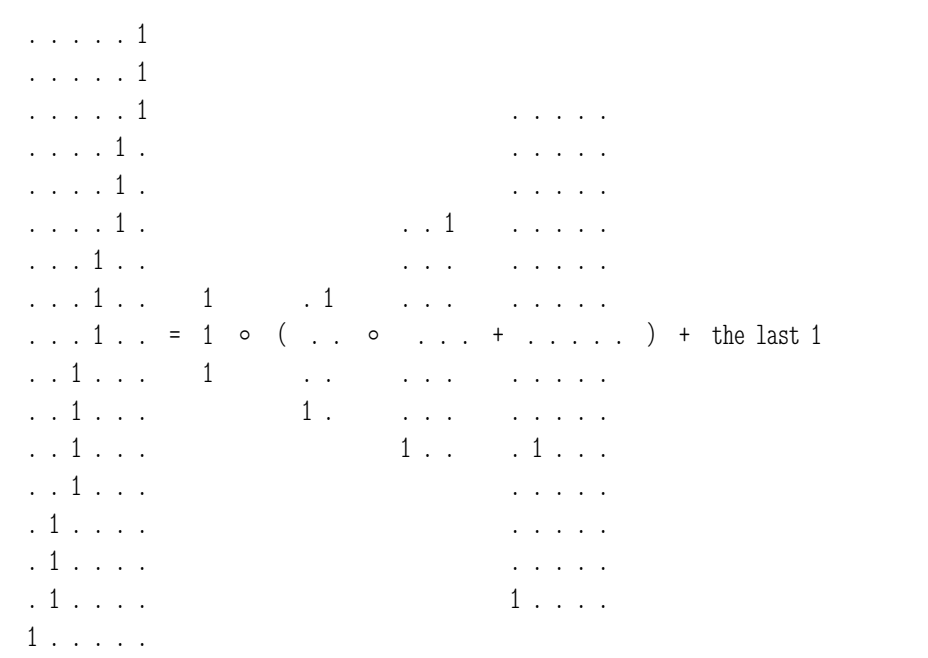
The upper right edge consists of these 11 lines:



I broke this down to the following, using a greedy algorithm:



That's five additions per pixel. The lower right edge is similar:



This works out to seven additions per pixel, so the whole right edge can be done in 12 additions per pixel; the left edge can then be done in 12 more, and the two subtracted, for a total of 26 operations per pixel, including the original prefix-sum calculation.

Not bad for a 600-tap bokeh kernel calculated to fit into 1024 pixels!

Plausibly, these numbers might scale linearly with the number of sides in the polygon, and logarithmically with the size of the kernel.

McGraw's singular-value-decomposition algorithm (see below) could probably produce a reasonable approximation of the same bokeh with a rank-3 reduced matrix; this amounts to summing three separable 32×32 filters, each requiring 64 multiply-accumulates, for a total of 192 multiply-accumulates. It has the advantage of not being limited to flat bokeh, but the additional disadvantage that its results are only approximate.

Pipelining

As described in Evaluating DSP operations in minimal buffer space by pipelining (p. 321), in cases where your dataflow topology *is* a strict pipeline of filters, and your filters only reach a limited distance into the past, you can pipeline them all into a single pass over a shared buffer, preserving just enough of the results of each stage in the pipeline to allow the next stage to run. This is especially important for low-computational-intensity algorithms like those explored here, because (at least on the CPU) it's likely that the algorithms' bottleneck will be memory bandwidth rather than the amount of time to add pixels together or whatever. Tiling is another related strategy for improving locality.

Floored Gaussian subtraction

Lenses with spherical aberration commonly produce bokeh that is

not completely flat; for example, it might be brighter or dimmer in the center. The above algorithms mostly cannot reproduce this, although the hollow-circle convolution will produce a brighter edge.

Maybe, to get a brighter or dimmer center, you could apply a Gaussian blur to the image (say, a third-order box filter) and add or subtract the attenuated blurred image from the bokeh image, using saturating subtraction when subtracting to prevent the creation of negative pixels where the support of the Gaussian extends beyond that of the bokeh. This is pretty imperfect since the darkness still leaks out to infect neighboring pixels, but might still look okay.

Previous work

The chord-decomposition and trapezoid algorithms above don't seem to have been published before, and in significant cases they seem to be dramatically more efficient than published algorithms.

Efficiently Simulating the Bokeh of Polygonal Apertures (McIntosh, Riecke, and DiPaola 2012) uses min between rotated and skewed box filters to get fairly realistic hexagonal and octagonal bokeh, also varying it with the depth of field and using max to get star-shaped polygons. This does produce visible artifacts, but they are fairly minimal. It cannot produce polygons with odd numbers of sides.

Fast Bokeh Effects Using Low-Rank Linear Filters (McGraw, 2014) used "low-rank linear approximations" to get linear approximations of arbitrary bokeh shapes. The best summary seems to be on p. 4:

> Our low rank filter approach for bokeh effects is to approximate an arbitrary filter kernel as a sum of separable kernels.

It criticizes the above paper as follows:

> McIntosh et al. [17] present a novel approach that is capable of generating polygonal bokeh, but their method has several restrictions that ours does not: the kernel is of uniform intensity throughout, the bokeh shape must be a union or intersection of parallelograms, and large overlapping bokeh do not blend properly.

In a way, it's similar to the various decomposition-based approaches I described above. He treats the rows and columns of the PSF as the rows and columns of a matrix, and then uses singular value decomposition to find a "best" rank-*N* approximation of that matrix; the vectors corresponding to the largest singular values then provide separable filters whose sum approximates the original filter kernel.

This is a very interesting approach to the problem of convolution in general, and it turns out there's a fair bit of research on it, although you wouldn't know that from reading McGraw's paper; see The miraculous low-rank SVD approximate convolution algorithm (p. 747) for details.

† Prefix sums are also known as scan or addition scan (+\ in APL), sum tables, integral images, cumulative sums, and summed area tables (SATs), the last especially when the prefix sum is taken along both the X and Y dimensions.

Topics

An IMGUI-style drawing API isn't necessarily just immediate-mode graphics

Kragen Javier Sitaker, 2015-09-03 (3 minutes)

There's this thing called "immediate-mode GUIs" floating around which is kind of related to immediate- versus retained-mode drawing systems. But it's not really. In an immediate-mode GUI, you supply the library with a function which draws your UI, and it invokes it when necessary, influencing its execution as needed.

A real drawing equivalent of immediate-mode GUIs might be how you would draw if you only had a single scan line of framebuffer. You could have a function that represented the image you wanted to draw, invoked with a drawing context object of type `dctx`, once every scan line:

```
void draw_stuff(dctx *c)
{
    draw_rect(c, 10, 10, 100, 100, DC_GREY);
    draw_circle(c, 10, 10, 100, 100, DC_BLACK);
    int baseline = 55;
    if (bbox(c, baseline - font_height, 0, baseline, 100)) {
        char *s = "Welcome";
        int text_width = text_escapement(s);  // for centering
        show_text(c, 55 - text_width/2, baseline, s, DC_RED, DC_BLACK);
    }
}
```

Note that this is almost entirely abstract over the question of whether you're drawing a scanline at a time or what.

The `bbox` function determines whether or not part of the currently-being-drawn area impinges on that bounding box. If not, the user-provided function above skips the potentially expensive calculations within. This is the same approach IMGUI libraries use to handle buttons being clicked, menus being open, and so on.

Serious IMGUI libraries do stuff to help you divide your GUI-drawing function into separately memoized parts, but I'm not going to touch on that here. Instead I'm going to focus on the idea of using this to make it usable to draw graphics scanline by scanline.

Let's consider what `show_text` has to do. It's being invoked with a `dctx`, a start point, some text, a foreground, and a background:

```
// in the dctx library:
void show_text(dctx *c, int x, int y, char *s, color fg, color bg)
{
    <<show_text implementation>>
}
```

Let's suppose for the moment that we only have one font, and it's a proportional bitmap font; and that this function in particular only has to draw a single line.

First, we ought to check to see if our bounding-box is being drawn, which means we have to compute it.

```
// in show_text implementation:
int width = text_escapement(s), max_x = x + width, min_y = y - font_height;
if (!bbox(c, x, min_y, max_x, y)) return;
```

Now, though, we need to copy the appropriate chunks of pixels into the scan line, in the appropriate color. This part inevitably depends on the fact that we're drawing a single scanline, rather than half a scanline or four scanlines.

```
int font_line = c->y - min_y;
char k;
for (char *p = s; (k = *p) != '\0'; p++) {
    short pixels = get_font_pixel_slice(k, font_line);
    char width = get_font_pixel_width(k);
    for (int i = 0; i < width; i++) {
        c->buf[x] = pixels & 1 ? fg : bg;
        x++;
        pixels >>= 1;
    }
}
```

And that's it, aside from some relatively mundane details about `get_font_pixel_width` and `get_font_pixel_slice`. That gives you the software equivalent of an old-style character generator, permitting color framebufferless rendering at arbitrary pixel offsets.

Topics

- Programming (p. 3658) (286 notes)
- Graphics (p. 3483) (91 notes)
- C (p. 3359) (28 notes)
- Graphical user interfaces (p. 3489) (23 notes)
- Immediate-mode GUIs (p. 3515) (8 notes)

Personal notes from 2013-06-06

Kragen Javier Sitaker, 2013-06-06 (updated 2014-04-24) (11 minutes)

Well, the movie with Caro didn't work out, due to technical difficulties: she'd bought the tickets for the wrong day, so we ended up starting to see some kind of 3-D animated hero film for kids before going down to the ticket counter to buy new tickets. Her intense frustration at this state of affairs reminded me that she's one of the most organized and together people in my current inner circle. Maybe I can improve my conscientiousness by hanging out with her more.

Dinner with Naudy, however, was awesome; but afterwards I was sufficiently tired (and stuffed --- Los Sabios food tempts one to overindulge) that I'm not going to the hospital tonight.

I did my first experiments with frequency-domain convolution and sparse factorings of time-domain FIR kernels, but I don't know

I've stuffed the last of the moldy clothes in the wash bucket so I'll wash them tomorrow. And I've gotten some leads on roommates on the web, which I'll check out more tomorrow.

So for the morning here's my plan:

• Wake up 10:00.
• Bootup and fsck (1 hour):

• Put away bed.
• Flaxseed omelet with salsa golf, or something.
• Wash dishes and another bottle.
• Fold and put away dry laundry.
• Shower and laundry.
• Sweep floors. (Sisyphean: I swept them two days ago, and they're already dirty again. This is still an improvement over Augean, I suppose.)
• Optimization (1 hour):

• Bottle soybeans.
• Put away stray objects from living room floor.
• Reticulate splines:

• Read dspguide chapter on frequency-domain convolution.
• Networking:

• Go to Santi's.
• Call Telefónica to get the internet connection here fixed.
• Investigate housing possibilities: Craigslist, and followup on leads.
• Work project:

• Check status of server; do the needful.
• Talk with client.
• Look for a cellphone on Mercadolibre, Craigslist, OLX.
• Multimedia services:

• 17:50: go to movie with Caro.

Yeah, that was the plan. Here's what I actually did:

• Wake up 10:00. Reset alarm for 11:00. At 11:00, reset alarm for 12:00. Finally get up 16:00. Apparently I wasn't giving myself adequate sleep if I had built up a six-hour deficit. Put away bed.
• Eat a hurried breakfast: an alfajor maicena, a fried egg with soy sauce, a cup of English Breakfast tea, and a glass of pear-vinegar-flavored sugar water.
• Remember Violeta is coming today and would be arriving about the time I'd be going to the movie. We arranged that on Tuesday; I can hardly cancel it now on a whim. Also I need to go pick up my blankets from the laundry place (AR$50), but I don't have the money in pesos, and I doubt the laundry place will make change for US$100. I need to go change some dollars to pesos.
• Spend 50 minutes changing dollars to pesos, then pick up the laundry. Apologize to Caro.
• Buy a bottle of bleach (AR$22 for four liters), since I'm almost out, and the cleaning-supplies store is right across the street.

An interesting thing to note was my anxiety level while returning from money changing to go to the laundry place: I worried they might have already closed (I hadn't noted their hours), and that I'd have to spend the night tonight again under only the sheets and bedspread, inconveniencing Violeta as well as myself. My guilt at this possibility was kind of extreme. My relief upon coming in sight of the people sitting around inside the laundry's plate-glass windows was enormous.

This anxiety and guilt is a miniature version of the debilitating version I experience during procrastination.

As it turns out, Violeta is delayed, and so I could have gone to see the movie anyway, which I probably would have found out if I had messaged her to find out her schedule when I got up instead of after the movie had already started. So in this case my lack of communication exacerbated the situation caused by my lack of planning. Rationally, I know that all of this is much more important to me than to anyone else, and that the guilt I'm feeling is out of proportion to the real negative impact it's having on other people, and that that same guilt is what makes me reluctant to initiate those communications --- a vicious cycle.

Now it's 18:00, and I have time to make a revised plan for the rest of the day.

• Cooking:

• Make noodles with white sauce.
• Make lentils.
• Make a salad, using oil, pear vinegar, lettuce, celery, and tomato.
• For the above, buy onions, milk, lettuce, and celery, before house maintenance:
• House maintenance (before Violeta arrives, time permitting):

• Sweep floors. (Sisyphean: I swept them two days ago, and they're already dirty again. This is still an improvement over Augean, I suppose. But it's a very quick task.)
• Bottle soybeans.

• Fold and put away dry laundry.
• Shower and laundry.
• Wash dishes and another bottle, if there's still time.
• Put away stray objects from living room floor.
• Study (alongside Violeta):

• Read dspguide chapter on frequency-domain convolution.
• Read remaining dspguide chapters.

(Planning time: 7 minutes.)

An hour later, I've bought the foods, plus an off-brand Terma, a new kind of bottlable noodles, and some fresh tomatoes. I'm soaking some sun-dried tomatoes now; maybe I can use them in the salad, or maybe Violeta would prefer a tuco sauce rather than a roux.

While I was out, I met a little girl jumping rope on the sidewalk, across the street from the MACABI building, which is surrounded by heavy brick barricades in case of another truck bomb. I hung up the blankets on the patio to air out, since I forgot to specify not to soak them in nose-anesthetizing perfumes.

I was disappointed to find that the bottle I washed yesterday still isn't completely dry. I'd left it to air-dry rather than using alcohol as usual, which seems to have been ineffective. I filled it with about a kilogram and a half of soybeans anyway; hopefully the moisture will diffuse throughout the soybeans rather than just making a few moist spots that can harbor mold.

Violeta was very pleased with the dinner, with which she helped. To the basic roux (butter, flour, and milk) I added salt, pepper, dried parsley, and freshly-grated nutmeg, with excellent results.

We ended up not studying, instead going out for ice cream (AR$25 for a quarter-kilo) before dinner. Dinner ended a bit past midnight, and then we went to bed.

We took a shower; I soaked the laundry (this load is a bucket full of clothes that had gotten moldy from a flood) with sodium percarbonate, but I haven't washed it yet.

Rather than going to the hospital to get an appointment, I decided to sleep instead. (To get an appointment, you have to be in line when they open the doors at 5:00.) This was probably a good idea, because I had another night of sleeping over 12 hours. While my health probably needs the sleep, I worry that it could promote depression.

I woke up at 6:30 with her this morning and made her pancakes with dried apricots for breakfast. As I woke up, I was dreaming that a family of six weasels had divided up my nasopharynx for living space, which was my first sign that I had a ferocious sore throat. I went back to bed when she left at 7:15, sleeping another 7 hours. I had to put on 很愛很愛 by Sammi (an album I bought in Seattle in 2000) and Queen's Greatest Hits on a couple of occasions to enable me to sleep through the child abuse next door. At one point, the man was roaring "¡Callate!" over and over at the small child, and when the child began to cry, he began to laugh. Now he's emitting periodic falsetto screams.

I spent some time reading Wikipedia about different acid-base theories and anatomy.

My lunch (I suppose it's not a breakfast since I had pancakes with Violeta before sunrise) will be breadsticks with the roux, plus some of the lentils. Then I'll wash the laundry that's been soaking overnight,

wring it out to dry, go call Telefónica, and head over to Santiago's house.

To the lunch above I added sesame seeds on the lentils, English Breakfast tea with the rest of the orange syrup, celery, and two fried eggs. I guess maybe my body thinks it's time to stop losing weight. I've left the shower running on the laundry to give it a post-soak rinse; now it's time for a shower and a real wash for the laundry.

Things on my shopping list: sugar, laundry detergent, Scotch tape, milk, butter, light bulbs. I read in Wikipedia that the sale of incandescent light bulbs was banned in Argentina in 2010; if that's true, the light bulbs may be a bit of a challenge.

One of the molded garments, a light-colored pillowcase, seems to have suffered spotty staining from the mold. I'm going to try soaking it in bleach water overnight to see if that helps.

Walking to Santi's house, I stopped by a hardware store which sells replacement fan parts and fan service (of the hardware-repair type, not the anime type). They also sell new desk fans: AR$120. I bought three replacement 28W lightbulbs (to replace 40W bulbs) for the apartment, which cost AR$60 in total. It turns out that the traditional incandescent lights are no longer available, but halogen replacements are.

Since this represents an investment in repairing the furnishings (increasing the value over the original) of about 3% of the rent, I'm going to see if I can persuade José to reimburse me. I guess I should have asked for a receipt.

I bought a helium Mylar balloon from a clown on the sidewalk for AR$40. It has pictures of Mickey and Minnie Mouse, and says in English, "Best wishes". On the top, it has a similar message in Chinese which presumably says the same thing.

Topics

Gaim group chat

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

So I want to make a Gaim (um, I mean Pidgin) extension for group activities such as games. The first activity, of course, is group chat. So here are some thoughts on how to achieve that.

People on the channel are connected in some sort of connected graph and relay broadcast messages to each other. The graph doesn't need to be acyclic; every message has a unique ID and everyone's client remembers the unique IDs of the messages they've seen recently, and only forwards on or displays new messages. Eventually, every message should traverse every link that is live at the time in one direction or the other.

The first question is how much the people should be allowed to know about each other. Of course since they can send broadcast messages visible to one another, they can establish communications if they want; if my client colludes with someone, they can snoop on the channel without anyone else being able to find out that they are there; and if my client colludes with someone, my client can tell them all the people I'm talking to. So, in the framework above, there's no way to enforce either a policy of anonymity or the opposite in the protocol; the question is just what the default and advertised policy should be.

I think I will start with revealing the truenames of everyone on the channel --- truenames being something like "ksitaker on AOL" --- because it has four advantages. It makes it difficult to censor or falsify the utterances of a particular person when others can contact them directly; it can allow the channel to heal when somebody goes offline unexpectedly; it facilitates side-channel conversations; and it reduces the probability of accidentally saying something in front of the wrong person.

The conversations ought to be private, in the sense that the default should be that chat lines aren't transmitted unencrypted and aren't forwarded to unannounced third parties.

The encryption is a bit tricky. There are three straightforward possibilities:

- Encrypt point-to-point between connected clients;
- Encrypt each message with a fresh "session key" and include copies of the "session key" encrypted with the public key of each participating client;
- Negotiate a shared "session key" periodically (at least every time group membership changes) and encrypt all messages with that.

Topics

Solar system scale model

Kragen Javier Sitaker, 2017-04-18 (1 minute)

Suppose you wanted to build a scale model of the solar system at manageable scale. For example, 5 meters in diameter.

You probably have to include at least out to Neptune. The Oort Cloud maybe isn't really necessary. But omitting Neptune would be unreasonable. Neptune orbits at 30.1 AUs, and if your model only has one side of the solar system, then you could make 30.1 AUs correspond to 5 meters.

30.1 AUs is 4.5e12 meters, so the scale of the model is about 1e12:1. At this scale:

• Sun diameter: 1.4e9 meters becomes 1.4 mm (visible with the naked eye!)
• Earth diameter: 13e6 meters becomes 13 microns (will require a microscope)
• Earth orbital radius: 150e9 meters becomes 150 mm
• Jupiter diameter: 140e6 meters becomes 140 microns (visible with the naked eye, barely, but with a microscope you should be able to see the Great Red Spot)
• Jupiter orbital radius: 5.2 AU, 780e9 meters becomes 780 mm
• Neptune diameter: 25e6 meters becomes 25 microns (pretty much still requires a microscope)

It would be super awesome if you could continuously maintain a carbon arc or something similarly bright at the position of the Sun, so that you could see what the Sun looks like from different distances. This might be kind of dangerous though.

Topics

Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection

Kragen Javier Sitaker, 2019-06-16 (updated 2019-07-05) (39 minutes)

Windowing the results of the Goertzel algorithm, or the Minsky circle algorithm applied as a frequency detector, over different-sized windows, can provide frequency detection with variable levels of precision. Some kinds of PLLs can be used in the same way, and you can adapt the Goertzel algorithm or the Minsky algorithm into a PLL. The Hogenauer-filter approach can provide very nice windows very cheaply.

The Minsky algorithm applied as a frequency detector in the most obvious way turns out to be precisely equivalent to the Goertzel algorithm applied to the backward differences of the input signal.

This is a continuation of the exploration I began in Cheap frequency detection (p. 3026), but this note stands on its own, which is to say that it duplicates a lot of the material in that other note.

The Goertzel algorithm and windowing it with prefix sums

The Goertzel algorithm accumulates a sequence of values with a resonant frequency: $s_n = (2 \cos \omega)\, s_{n-1} - s_{n-2} + x_n$. You can verify that, if $x_n = 0$, this holds true of the sequence $s_n = a \sin(\omega n + \varphi)$ for any a and φ. First, consider the case $a = 1$, $\varphi = 0$, and suppose that it's true of s_j for all $j < n$; does it then hold true at n?

$$s_n = (2 \cos \omega)(\sin(\omega(n-1))) - \sin(\omega(n-2))$$

The formula above is a little more symmetrical if we rewrite it with $m = n - 1$:

$$s_{m+1} = (2 \cos \omega)(\sin(\omega m)) - \sin(\omega(m-1))$$

If you don't remember angle-sum formulas from high-school trigonometry, you can trivially rederive them from Euler's formula $e^{it} = \cos t + i \sin t$; if $t = h + j$, this becomes $e^{ih}e^{ij} = (\cos h + i \sin h)(\cos j + i \sin j) = (\cos h \cos j - \sin h \sin j + i(\cos h \sin j + \sin h \cos j))$, so $\cos(h + j) = \cos h \cos j - \sin h \sin j$, and $\sin(h + j) = \cos h \sin j + \sin h \cos j$.

Here, we're interested in these identities:

$$\sin(\omega(m+1)) = \sin(\omega m + \omega) = \sin(\omega m)\cos\omega + \cos(\omega m)\sin\omega$$
$$\sin(\omega(m-1)) = \sin(\omega m - \omega) = \sin(\omega m)\cos\omega - \cos(\omega m)\sin\omega$$

The second one gives us

$$\begin{aligned} s_{m-1} &= (2\cos\omega)(\sin(\omega m)) - \sin(\omega m)\cos\omega + \cos(\omega m)\sin\omega \\ &= (2\cos\omega)(\sin(\omega m)) - \cos\omega\sin(\omega m) + \cos(\omega m)\sin\omega \\ &= \cos\omega\sin(\omega m) + \cos(\omega m)\sin\omega \end{aligned}$$

which above is established as the value of $\sin(\omega(m+1))$, so $s_{m+1} = \sin(\omega(m+1))$.

To establish that this is true for all values of a and φ, we can simply

note that this is a linear time-invariant recurrence:

$s_n = (2 \cos \omega)\, s_{n-1} - s_{n-2} + x_n$

m or *n* does not enter into it except as an index, and the output on the left is linear in all the inputs on the right.

This is the unique way to get a sine wave of a fixed angular frequency ω as such a linear recurrence on the two previous terms, $\sin(\omega n) = s_n = a s_{n-1} + b s_{n-2}$; that is, if you solve that equation for *a* and *b*, the only possible values for *a* and *b* are $2 \cos \omega$ and -1.

I think it's somewhat trickier to show is that, if $s_0 = s_1 = 0$, s_n and s_{n-1} form a linear encoding of $\Sigma_j x_j \mathrm{cis}(\omega j)$ for $j \in [2, n]$. But it's true; unless I'm confusing something, $\Sigma_j x_j \mathrm{cis}(\omega(n-j)) = \gamma_k = s_k - \exp(-i\omega)\, s_{k-1}$. s_k is, of course, a linear function of any such previous sequence of s_n, s_{n-1} and the x_j since that point, since it's a linear function of x_k and s_{k-1} and s_{k-2}, which themselves are such a linear function.

The impulse response of this system makes it clear — an impulse in *x* will kick off a proportional sinusoidal oscillation in *s*.

So at any point along the accumulation of this sequence, we can extract the dot product between the complex exponential e^{ij} and the *x* samples so far. In effect, the Goertzel algorithm computes the prefix sum of a particular frequency component of our signal, just in a slightly obtusely-encoded form.

(Note that this is precisely one of the components of the Fourier transform of the signal over that interval.)

Using s_k as a prefix sum to rectangularly window a frequency component

This means that, if we want to know how much of a particular frequency was present during a given time interval, and we've saved off the s_k values for that time interval, we can just do the γ_k calculation for the two points above, perhaps apply a rotation to bring them into phase (if they aren't separated by an integer number of cycles), and take the difference. (This assumes that we aren't getting rounding errors, but since we're essentially computing a sum table here, our *s* values will grow without bound if the signal *x* contains a nonzero frequency component at the frequency of interest. As with Hogenauer filters, it might be wise to do the calculation in purely integer math to avoid this.)

This subtraction amounts to temporally windowing the signal with a rectangular window. So it convolves the spectral response of the filter with the Fourier transform of that rectangular window, which is a sinc. So, by using different widths of window on *the same* s_k *signal*, we can get different levels of frequency selectivity. And in fact we can do this with only a single multiply-subtract and addition per sample (since the factor $2 \cos \omega$ is fixed as long as we don't change the frequency), even though we're pulling out a number of different window widths from that single resonator.

Goertzel without multiplies and A440 even temperament

The multiply-subtract $(2 \cos \omega) s_{n-1} - s_{n-2}$ can, for some angles, be done with a couple of subtracts and bit shifts: $(s_{n-1} \ll 1) - (s_{n-1} \gg p) - s_{n-2}$. This works when $2 \cos \omega = 2 - (1 \gg p)$ and thus $\omega = \cos^{-1}(1 - (1 \gg (p+1)))$. This gives the following periods for shift lengths from 0 to 20 bits:

```
>>> 360/(numpy.arccos(1-2**-(1 + numpy.arange(20.0)))*180/numpy.pi)
array([    6.        ,     8.69363162,    12.43307536,    17.67813872,
          25.06699928,    35.49668062,    50.23272124,    71.06297418,
         100.51459792,   142.16068241,   201.05374803,   284.33872284,
         402.11976897,   568.68612356,   804.245674  ,  1137.37658591,
        1608.49441598,  2274.75534121,  3216.99036595,  4549.51176711])
```

For the particular case of 44.1-ksps audio, these work out to the following frequencies in Hz:

```
>>> 44100/(360/(numpy.arccos(1-2**-(1 + numpy.arange(20.0)))*180/numpy.pi))
array([ 7350.        ,  5072.67870839,  3546.99048555,  2494.60651377,
        1759.28516646,  1242.3696873 ,   877.91381613,   620.57633403,
         438.74224154,   310.21235444,   219.34433171,   155.09670846,
         109.6688186 ,    77.54717088,    54.83399094,    38.77343753,
          27.41694317,    19.38670028,    13.70846505,     9.69334783])
```

These are far closer to A440 musical note frequencies than we would have any right to expect; 438.7 Hz is about 5 cents flat of A_4 (A above middle C), and 877.9 is only about 4.1 cents flat of A_5, and the frequencies that are not As are even-tempered D♯/E♭ notes. It crosses over from being sharp to being flat in between 2494 and 3546 Hz, precisely where the human ear is most perceptive.

Better windows through Nth-order prefix sums

Usually we don't think of sinc as being a very good filter frequency response, because well into the stopbands, you keep getting these response peaks where an odd number of half-waves fit into your window. The Hogenauer-filter approach to solving this problem is to use N levels of sum tables to get an Nth-order approximation of a Gaussian window, which in theory has the optimal tradeoff between frequency precision and temporal precision — the minimal joint uncertainty, according to the uncertainty principle. If we consider the γ_k given above as representing values in a complex-valued sum table, we could compute running sums (prefix sums) of them to get Nth-order prefix sums, which we can then differentiate N times in the usual CIC-filter way to get a given frequency component with that window.

But calculating the γ_k is considerably more expensive than calculating the s_k: $\gamma_k = s_k - \exp(-i\,\omega)\,s_{k-1}$, so while calculating s_k required a single *real* multiply, addition, and subtraction (presuming all real x_j), calculating γ_k requires a *complex*–real multiply and then a complex–real subtraction: two real multiplies and a real subtraction, which is twice the multiplies. Normally we sweep this under the rug by presuming that we only calculate γ values occasionally, so this expense doesn't matter, but to compute the prefix sum ($+\backslash\gamma$, in APL notation) we would need to do those multiplications for every sample, and then it would matter. Also, computing complex prefix sums would involve twice as many real additions as computing real prefix sums.

But in fact we can avoid this hassle; $\Sigma_k\gamma_k = \Sigma_k(s_k - \exp(-i\,\omega)\,s_{k-1}) = \Sigma_k s_k - \exp(-i\,\omega)\Sigma_k s_{k-1}$, so we can do that whole computation using a sum table for s which we use twice, instead of computing separate real and imaginary Nth-order sums. Then we can compute a γ value windowed with an Nth-order approximation to a Gaussian window

using N real subtractions, a complex–real multiply, and a real subtraction. The same Nth-order sum table will serve for any window size. XXX is this right? Or do we need to either do N subtractions per sample or N(N-1)/2 subtractions per output? Or maybe decimate by the window width M and then do the N subtractions once every M samples?

Roundoff

However, this surely requires special measures to avoid roundoff for the Hogenauer-filter part of the computation; the values in a third-order sum table, over an interval where the signal is roughly constant, will grow proportional to the cube of the constant value. The standard approach is to implement Hogenauer filters using integer arithmetic with wraparound, thus entirely eliminating rounding error. Maybe an alternative exists, representing the sum tables using the tree constructed in the standard parallel prefix-sum algorithm, so that the values being added at each tree node are of comparable size and roundoff lossage is insignificant; but I suspect you need to traverse the tree at subtraction time, adding a logarithmic slowdown, and I'm not sure how this generalizes to second-order and higher sum tables.

Even without sum tables, it's a frequently noted observation that the Goertzel algorithm is prone to numerical error.

Frequency detection

So you could imagine using this approach to narrow down the frequency range where a particular signal might be in your input: first use very short windows to see if there's anything in the frequency range at all, then use longer windows at a larger number of frequencies (some of which might be the same frequency — just using the same prefix sums with a longer window width) to figure out where the signal or signals might be inside that window.

Alternatively, if your signal is sparse enough in frequency space, you can use a temporal window size long enough that only one significant frequency component will be within the corresponding frequency window, and then use its successive phases in successive overlapping windows to determine its real frequency, the way a phase vocoder does. So, for example, in The Bleep ultrasonic modem for local data communication (p. 966), I want to detect tones at 17640 Hz and 19110 Hz. I could very reasonably use the Goertzel algorithm with a frequency window centered on 18375 Hz, whose beat frequency with either 17640 or 19110 Hz is 735 Hz; that means that every 1.36 milliseconds (60 samples at 44.1ksps) the phase relationship between the reference frequency and the real frequency will shift by 90°. So if I perform Goertzel for that frequency over nominally 15-sample windows, I should get a nice clear phase that I can unwrap. I haven't tried this yet.

(18375 Hz is 2.6180 radians or 150° per sample, so the Goertzel recurrence becomes roughly $s_n = -1.732\ s_{n-1} - s_{n-2}$.)

Alternatively, suppose I'm trying to detect the tune I'm whistling as in Whistle detection (p. 357), in the band 600Hz to 1600Hz. Typically if I'm whistling it'll be the loudest sound in that band, maybe 20 to 40 dB louder than anything else nearby. Suppose we have a whistle at 1477Hz and we're trying to detect it by windowing a

1400Hz Goertzel filter; we'll get a phase spinning around the circle at 77Hz. So if our window mostly averages over, say, 3 ms (4.2 cycles of the 1400Hz reference), the phase will rotate around almost one quadrant, attenuating the phasor's average magnitude by about √2, or 1.5 dB. By unwrapping the phase (again, like a phase vocoder) we can determine the beat frequency and phase direction, and thus the real frequency. So windowing a single Goertzel filter about 500 times a second covers about 200Hz of bandwidth, so about five or ten of them should cover the whole band of interest.

As long as the window is pretty close to Gaussian, there's a relatively precise reciprocal relationship between the bandwidth and necessary number of windows per second for each filter. Using a window that's half as long requires us to use twice as many windows per second, but also covers twice the bandwidth in a single filter, so we need half as many filter center frequencies. In the limit, where our windowed filter kernel is just an impulse, we're just trying to "unwrap the phase" of individual samples (or perhaps pairs of samples, since y_k uses s_{k-1} as well). At the other extreme, with a very large number of very narrow frequency bands, each with windows tens or hundreds of milliseconds long, we only need check each band every hundred or more milliseconds, but we start losing temporal precision.

But at that point, wouldn't it be faster to window a downconverted signal and do a Fast Fourier Transform? Then we only have to do O(lg N) work per sample to get each of N frequency bands instead of O(N) work.

(Perhaps the best way to understand the CIC-windowed Goertzel filter in this context is precisely that it's providing a windowed downconverted signal, which we can then Fourier-transform if we like?)

In one line of C

Here's a one-line obfuscated C program from Cheap frequency detection (p. 3026) that generates a sine wave using Goertzel's algorithm:

```
main(s,t,u){for(t=32;u=s,1+putchar(128+(s-=t-s+s/8));t=u);}
```

If you compile it to an executable called `goertzel` on a Linux machine with ALSA or an ALSA emulation, you can run it with the command `./goertzel | aplay`.

(`arecord` and `aplay` default to 8000 unsigned 8-bit samples per second, which is convenient since it means we can use `getchar()` and `putchar()` to read and write samples.)

The Minsky algorithm

One way to think of the Goertzel algorithm's oscillator is by linearly extrapolating to the current sample, then subtracting a second-order correction to make it curve back towards zero at the desired frequency.

$$s_n = (2 \cos \omega)\, s_{n-1} - s_{n-2}$$
$$= s_{n-1} + (s_{n-1} - s_{n-2}) - 2(1 - \cos \omega) s_{n-1}$$

In essence, it's calculating the desired second derivative by multiplying a small negative number, $-2(1 - \cos \omega)$, by the latest

sample. While this is in theory exactly correct, it's easy to see that it could give rise to significant roundoff errors for sufficiently low frequencies. In the limit, where cos ω rounds to 1, it will stop oscillating entirely and merely linearly extrapolate. For angular velocities ω approaching 0, $1 - \cos\omega = \frac{1}{2}\omega^2 + O(\omega^4)$, which is why changing this correction by factors of 2 in the above note about multiplication-free Goertzel resulted in changing the frequency by roughly half an octave.

(However, don't be misled into thinking that this means Goertzel is merely a quadratic approximation that will fail at high frequencies. As shown earlier, if performed with exact arithmetic, it's exact.)

The Minsky algorithm is similar to the Goertzel algorithm, but while the Goertzel algorithm effectively calculates the current derivative of the oscillation from the difference between the last two samples, the Minsky algorithm reifies the first derivative as a separate variable:

$$s_n = s_{n-1} + \varepsilon c_{n-1} + x_n$$
$$c_n = c_{n-1} - \varepsilon s_n$$

XXX why does LaTeX give you a lunate epsilon ϵ by default, reserving normal epsilon for \varepsilon? Should I be using ϵ here?

This requires two real multiplications per input sample instead of Goertzel's one multiplication, but I hypothesize that it should have a smaller roundoff error; $\varepsilon \approx \sin\omega$, which means that for small angular velocities, it's proportional to the angular velocity rather than (as in Goertzel) the square of the angular velocity. So sometimes you can get more accurate results with Minsky.

ε is not precisely sin ω; more precisely, $\omega = 2\sin^{-1}(\varepsilon/2)$, so $\varepsilon = 2\sin(\omega/2)$, as explained in Cheap frequency detection (p. 3026) and *Minskys and Trinskys* and roughly in HAKMEM. For small angles, the approximation is very close.

One reason for roundoff error in Goertzel is that the energy of the resonator *s* alternates between being in the (square root of the) velocity — that is, the difference from one sample to the next — and being in the (square root of the) displacement — that is, the value of the latest sample. When the frequency is in the neighborhood of one radian per second, this doesn't cause much error, but for very low frequencies, the velocities, measured from one sample to the next, can be very small compared to the sample values. For one milliradian per sample, for example, at angles $\pi/4 + n\pi/2$, the velocity and the displacement are both at $\sqrt{2}$ of their maximum value, but the velocity there is 1000 times smaller than the displacement. So if you have eight decimal digits of precision, like a 32-bit IEEE-488 float, your velocity only has about five decimal digits of that precision, which means that every half cycle your signal has been rounded to five decimal digits. The Minsky algorithm doesn't have this problem in this form, since *s* and *c* are of similar magnitude, but it might have a similar problem in that the increments being added to *s* and *c* are potentially very small compared to their magnitude.

Both Minsky and Goertzel are stable, in the limited sense that they don't go to infinity over an infinite interval when the input is zero, if computed without roundoff. (They are *not* BIBO stable, of course.) The argument that Minsky is stable is that we can rewrite the above system as follows when $x_n = 0$:

$$s_n = s_{n-1} + \varepsilon c_{n-1}$$

$c_n = (1 - \varepsilon^2)c_{n-1} - \varepsilon s_{n-1}$

In Unicode-art matrix form:

```
⎡ sn ⎤ = ⎡  1    ε   ⎤ ⎡ sn-1 ⎤
⎣ cn ⎦   ⎣ -ε  1-ε² ⎦ ⎣ cn-1 ⎦
```

The determinant of that matrix is precisely 1. (XXX but its L^∞ norm is 1+ε; what's up with that?)

In a sense, s_n and c_n are temporally offset from one another by half a sample, and we are thus doing leapfrog integration of the harmonic-oscillator ODE $\ddot{s} = -\varepsilon s$. For slow oscillators, this is a small difference, so $s_n^2 + c_n^2$ is a fairly precise estimate of the energy in the system, but for fast oscillators it can be a large one. I think *Minskys and Trinskys* has calculated the exact expression for the correction, but I can't remember.

(It's easy enough to add an exponential decay to either Goertzel or Minsky, which results in the system state at any given time being an exponentially-windowed filter of the frequency of interest, but here I'm focusing on the prefix-sum-based approaches because I think they can produce nicer windows almost as cheaply.)

The Minsky algorithm in one line of C

Here's a one-liner obfuscated C Minsky-algorithm audio oscillator I wrote in 2017:

```
main(x,y){for(y=100;1+putchar(x+128);x-=y/4,y+=x/4);}
```

You can pipe it to `aplay` just as with the above Goertzel program. The tone is agreeable.

Varying ε, and efficiency

The canonical form of Minsky's algorithm above uses the same ε in both steps, but as is done in *Minskys and Trinskys*, you can scale c up and down relative to s by using a separate δ:

$s_n = s_{n-1} + \varepsilon c_{n-1} + x_n$
$c_n = c_{n-1} - \delta s_n$

(Here I have the ε and δ backwards from the use in Cheap frequency detection (p. 3026).)

If we're only interested in how s behaves, and $c_0 = 0$, then it turns out that only the product δε matters; if we vary δ while holding δε constant, the c_n values get scaled up or down by δ, but the sequence of s_n remains constant, barring roundoff errors. In particular, as explained in Cheap frequency detection (p. 3026), we can choose $\delta = 1$ or $\varepsilon = 1$. Say we choose $\delta = 1$; to get the same angular velocity of ω radians per sample, $\varepsilon = 4 \sin^2(\frac{1}{2}\omega)$. So our new equations are:

$s_n = s_{n-1} + (4 \sin^2(\frac{1}{2}\omega))c_{n-1} + x_n$
$c_n = c_{n-1} - s_n$

So now we need one subtraction, two additions, and a multiplication per sample. The Goertzel algorithm requires one subtraction, one addition, and a multiplication. So our multiplication penalty has disappeared, but I think the rounding-error advantage has disappeared with it: c can now have peak values much larger or smaller than s, so we can lose precision when energy moves from one to the other.

If we instead choose $\varepsilon = 1$, we get this equivalent version:

$$s_n = s_{n-1} + c_{n-1} + x_n$$
$$c_n = c_{n-1} - (4 \sin^2(\tfrac{1}{2}\omega))s_n$$

We can reverse the order of the updates, as is usually done, which shifts the indices of c by 1:

$$c_n = c_{n-1} - (4 \sin^2(\tfrac{1}{2}\omega))s_{n-1}$$
$$s_n = s_{n-1} + c_n + x_n$$

We can expand this out a bit:

$$c_n = c_{n-1} - (4 \sin^2(\tfrac{1}{2}\omega))s_{n-1}$$
$$s_n = (1 - (4 \sin^2(\tfrac{1}{2}\omega)))s_{n-1} + c_{n-1} + x_n$$

This is looking suspiciously familiar! What is $4 \sin^2(\tfrac{1}{2}\omega)$, anyway? Well, $\sin(\tfrac{1}{2}\omega) = \pm\sqrt{(\tfrac{1}{2}(1 - \cos \omega))}$, so it ends up being $2 - 2 \cos \omega$. Which means that this is actually:

$$s_n = (-1 + 2 \cos \omega)s_{n-1} + c_{n-1} + x_n$$

If we want to express this purely in terms of s, we need to eliminate c_{n-1}. Since $s_n = s_{n-1} + c_n + x_n$, $s_{n-1} = s_{n-2} + c_{n-1} + x_{n-1}$, so $c_{n-1} = s_{n-1} - s_{n-2} - x_{n-1}$. So we have

$$s_n = (-1 + 2 \cos \omega)s_{n-1} + s_{n-1} - s_{n-2} - x_{n-1} + x_n$$
$$s_n = (2 \cos \omega)s_{n-1} - s_{n-2} + x_n - x_{n-1}$$

We have the not entirely unexpected result that using Minsky's algorithm in this way as a signal filter is equivalent to using Goertzel's algorithm on *the backward differences of* the input signal, which in this context can be understood as a high-pass prefilter introducing a 90° phase shift.

Since the tricks we were playing with scaling c up and down don't affect the sequence of values in s, this result applies to using the vanilla form of Minsky's algorithm, too.

This means that if we want to precisely reproduce the results of Goertzel's algorithm using Minsky's algorithm in the straightforward way, we need to run Minsky's algorithm on *the prefix sum of* the signal. This may be a practical problem in situations where using a prefix sum may result in roundoff errors. Perhaps adding the input samples into c instead, while still reading the output from s (updated after c) would solve that problem? XXX investigate further.

A PLL in one line of C

In 2012, after struggling with software PLLs ("SPLLs") for a while, I wrote this obfuscated one-line first-order PLL in C:

```
main(a,b){for(;;)putchar(b+=16+(a+=(b&256?1:-1)*getchar()-a/512)/1024);}
```

If you compile it to an executable called `tinypll` on a Linux machine with ALSA or an ALSA emulation, you can run it with the command `arecord | ./tinypll | aplay`; it tries to emit a tone that tracks the pitch of your voice, but one octave higher. This works better with headphones, so that the microphone isn't picking up the tone it's emitting.

This isn't a very good PLL, but I think it's an excellent way to show the anatomy of a PLL. It contains two continuously-varying state variables, `a` and `b`. `b`, or rather the low 9 bits of `b`, is the phase accumulator of an oscillator, while `a` is the low-pass-filtered error from the phase detector.

When the program gets an input sample with `getchar()`, it feeds it into a "chopper":

```
(b&256 ? 1 : -1) * getchar()
```

Since the low 9 bits of `b` are the phase accumulator for the oscillator, its bit 8 tells us which half of the oscillation we're in. And, as the oscillator alternates between the halves, it either inverts the input signal, or it doesn't.

How does that chopper compute a phase error? Well, if we suppose that the input signal is an oscillation whose frequency is close to that of the PLL's oscillator, then the sum of that chopped signal over a whole cycle tells us the relative phase. If the two oscillations are perfectly in phase, then we're inverting precisely the negative part of the input signal, and so we get a large positive number. If they're perfectly out of phase, then we're inverting precisely its positive part, so we get a large negative number. And if they're in perfect quadrature, then the samples we inverted are half negative and half positive (and cancel each other out), and so are the samples we didn't invert, so we get zero. If there's a small phase error from quadrature, say so that we're inverting a little more of the negative signal than the positive one, we'll get a small negative number. And any DC bias cancels out.

So a sum over a whole oscillation of this chopped signal gives us some kind of indication of how far away we are from quadrature, and close to quadrature it's linear. So naturally the next thing we do is to add that phase-error sample into our phase-error accumulator:

```
a += … - a/512
```

To sum over precisely a whole oscillation would require keeping in memory an array of values to run a rectangular window over, so here we just use an exponential filter with a time constant of 512 samples, which is 64 milliseconds. (This is guaranteed to be several cycles long over the frequency range we can reach here.) Note that this also means that the magnitude of `a` is about 512 times bigger than the chopped input samples getting fed into it — if our average phase error is about 3 per sample, say, then `a` reaches a steady state at $a \approx 1536$.

So, then we want to drive our oscillator at a frequency that depends on this filtered phase error:

```
b += 16 + (a …)/1024
```

If our input signal is always 0, `a` will decay to 0, and this simply reduces to `b += 16`, which means it will reach 512 and its low 9 bits will wrap around every 32 samples, which is 250 Hz. So that's the "natural frequency" of the chopping. But if `a` is in the range, say, 1024 to 2047, then this is `b += 17` and it pitches up to 265 Hz. On most systems C division rounds towards 0, so if `a` is slightly negative, it doesn't lower the pitch, but once it gets to -1024, the pitch drops to 234 Hz.

So if you're screeching at your computer at 234 Hz, the chopper will initially drift in and out of phase with your screech 16 times a second, but if it ever manages to accumulate a phase error of -1024 in `a`, it enters a steady state where it's chopping your signal just enough out of quadrature to maintain that -1024. If it falls behind your screeching, `a` will decay and the chopping will speed up until `a` is

growing again, and if it gets ahead of your screeching, `a` will grow to an even more negative value and its chopping will slow down.

This sounds ridiculously crude, but there's a certain amount of noise on `a` from the regular increases and decreases of the phase error accumulator four times per cycle (about every 8 samples), plus actual signal noise, so it works better than you would think. Not that well, but better than you'd think.

Finally, it emits an output tone:

```
for (;;) putchar(b…);
```

This is the low 8 bits of the 9-bit phase accumulator in `b`, so it produces a triangle wave at twice the chopping frequency, so, in the neighborhood of 500 Hz. Triangle waves sound really harsh, but because they're so harmonics-rich, they make it easier to hear pitch changes than smoother waveforms would.

Notes on SPLLs in general, and using real sinusoids with them

The ramp up and down of the phase error accumulator at twice the frequency being detected is called "VCO control line ripple" or "reference spurious" in analog PLLs. This is often a thing you want to minimize, though in this case the noise it adds probably improves system performance through stochastic resonance. You could notch-filter this frequency out with a simple feedforward comb filter by the simple expedient of averaging two samples from the phase error signal half an oscillation apart (adjusting this, ideally, to the current oscillation frequency); or, since doing this requires a table of past phase-error samples anyway, use a prefix sum to calculate a simple moving average, a box filter, of that width, and dispense with the exponential. Box filters have a better noise-suppression/tracking-latency tradeoff than exponentials anyway.

I think there's an even simpler solution, though: in a digital system, it probably isn't necessary to update the phase error after every input sample anyway; you can do that processing at a lower sample rate, which filters out fast oscillations by construction.

Although the overall system isn't linear, it has a lot of linear pieces in it. This phase detector, for example, produces an output signal that's proportional to the amplitude of the waveform it's locking onto. If the waveform is near a local extremum where it gets chopped from negative to positive, then the phase-detector output is also locally linear in the phase error. The chopper frequency shift is linear in the filtered phase error signal, and the exponential filter on that signal is linear and time invariant.

Each of these components — the phase detector, the feedback-loop low-pass filter, the variable-frequency oscillator, and the feedback path from the oscillator to the phase detector — has lots of possible realizations. This particular phase detector is called a "type I" phase detector, but there are also "type II" phase detctors. And there are higher-order PLLs, which I don't understand at all. I'm pretty much just sticking to basic PLLs here because that's all I understand at all.

Chopping with a square wave means your PLL is sensitive not only

to its nominal frequency but also its odd harmonics. There are cases where this doesn't matter, but in particular if you're doing this on a sampled signal, the odd harmonics can alias down to other strange frequencies. In my 18375-Hz example above, for example, the third harmonic would be 55125 Hz — probably heavily attenuated by the antialiasing filter on your sound card, but at 44.1ksps, it aliases down to 11025 Hz. The fifth harmonic is even worse: it aliases to 3675 Hz. (Subsequent harmonics repeat this cycle backwards and forwards.)

And, indeed, that's what you see if you analyze the output of this program:

```
#include <stdio.h>
int i;
int main(){for(;;)putchar(i++*18375*16/44100&8?0:128);}
```

Feed it to `head -c 44100 | sox -r 44100 -t raw -u1 - sqwv.wav`, hit ^C, run `audacity sqwv.wav`, and plot the spectrum, and there are three sharp peaks at 18375 Hz, 11025 Hz, and 3675 Hz, just as predicted. And if you were multiplying that oscillation pointwise by an input signal to try to do phase detection, you'd be detecting all three of those frequencies, too.

So you may actually prefer to multiply by a real sine wave at the frequency of oscillation, which of course is what the Goertzel and Minsky resonators do; they also both have the advantage that they can give you a phase readout fairly directly, rather than using these subtle arguments about halves of chopped waves canceling each other out. But to use them for this application, you need to be able to adjust their resonant frequency.

A thing to notice is that, in this single-line SPLL, there's nothing left over that tells you if you actually have a signal or just silence. `a` will be 0 if you're perfectly locked onto a strong 250-Hz signal or if the signal is all zeroes. `a` might be 2048 because it's locked onto a really strong signal around 281 Hz, or a weak one, or because it's randomly being buffeted by strong random noise. To get the signal amplitude, you can chop the input signal with a second chopper in quadrature with the one you use for phase detection; this chopper will be in phase with the input signal instead of in quadrature with it, and so it amounts to synchronous rectification of the input signal. (The Goertzel or Minsky approach does this implicitly.)

This in-phase-chopped signal gives you the amplitude of the oscillation at the frequency of interest (and, as noted above, potentially some other frequencies too). Earlier I had suggested using a box filter in the feedback path to eliminate phase-error ripple, using a prefix sum calculated over some segment of the quadrature-chopped signal. By computing an analogous prefix sum of the in-phase-chopped signal, you can enable the same kind of after-the-fact frequency selectivity described in the earlier sections about fixed-frequency Minsky and Goertzel resonators: by subtracting two nearby samples on the I-chopped and Q-chopped prefix sums, you can detect signals over a wider bandwidth, or by subtracting prefix-sum samples with a large lag between them, you can detect them over a very narrow bandwidth. For better or worse, your detection band chirps along with your PLL oscillator.

There is presumably no PLL equivalent to the phase-vocoder-like

use of the fixed-frequency oscillators that I mentioned above, since the whole point of using a PLL is that it finds the frequency of the signal of interest and follows it.

Altering the frequency has some subtle effects which may introduce numerical errors here. As mentioned earlier, for fast Minsky oscillators, the half-sample offset between the two state variables becomes significant; altering the frequency will alter that relationship, and might rob energy from the system or add energy to it. Similarly, with Goertzel, lowering the frequency can add energy to the system, but won't always — it depends on what part of the oscillation *s* is in. (I don't know if it can also rob energy from it.) Presumably we can work out how to correct these errors.

Related work; contributions?

I ran across a 2015 paper by Sridharan, Chitti Babu, MuthuKannana, and Krithika entitled "Modelling of Sliding Goertzel DFT (SGDFT)…" which seems to have some things in common with the above. They're using a Goertzel oscillator in their PLL, but it isn't the oscillator; it's in the feedback path between the oscillator and the phase detector. They're also using a moving-average filter (a box filter). Moreover, they're using a PI controller in the loop to set the oscillator frequency in order to drive the phase error to zero (since their application is power grid synchronization), rather than the proportional control I used in the example above. As far as I can tell, by "SGDFT" they just mean the Goertzel algorithm.

I haven't found anybody talking about the relationship between the Goertzel algorithm and the Minsky algorithm, about using prefix sums to get variable tradeoffs between precision in the frequency domain and in the time domain without having to redo the multiplications, about using the Goertzel or Minsky algorithm as a combination phase-detector and oscillator in a PLL, or about the fortuitously nearly-A440 Goertzel frequencies we get with just a bit shift. I've never seen anybody talk about doing the Goertzel algorithm in wrapping integer math to avoid roundoff error (though, admittedly, it probably only makes sense in the context of Hogenauer-style windowing). I've never seen a discussion of using the Goertzel algorithm to detect a frequency *near* the target frequency and precisely identify it by unwrapping the phase. But it's possible that that's just because I'm just not that familiar with the literature. Or that the ideas are wrong.

However, most of the PLL and Minsky-algorithm and Goertzel-algorithm stuff is very well known. As far as I can tell, the relationship between the Minsky algorithm and the Goertzel algorithm hasn't been published, but quite possibly it's just too obvious to those skilled in the art to be considered novel.

Topics

Autism is overfitting

Kragen Javier Sitaker, 2019-08-31 (1 minute)

Many of the perceptual deficits of autism — difficulty recognizing familiar faces when a hat or beard is added or removed, stress and meltdown when facing departures from routines or minor changes in environment such as a book out of place on a bookshelf, memory of perceptual details neurotypicals don't notice or consider unimportant, difficulty distinguishing signals in the presence of background noise or environmental chaos, inflexible behavior — while commonly attributed to executive-function deficits, bear a suggestive similarity to a pathology in machine learning and in particular artificial neural networks called "overfitting", in which a learner adapts itself to particular examples from its training set at the expense of generalizability. Perhaps, at the human neural level, overfitting is a general mechanism that causes many of the different characteristic manifestations of autism.

Topics

- Psychology (p. 3669) (18 notes)
- Autism (p. 3334) (2 notes)
- Speculation

Quasimode keyboard

Kragen Javier Sitaker, 2018-07-14 (24 minutes)

Jef Raskin invented the word "quasimode" to describe the user interface technique of setting a mode only as long as a certain key is held down, like Shift, Control, or ⌘. Quasimodes can multiply the expressivity of user interfaces without the usability problems created by full-blown input modes like Caps Lock or Vim's command mode.

I'm going to focus here on text editors, which (for better or worse) are the predominant medium for human expression through computers in 2018, whether you're typing a WhatsApp message, a Tweet, a Facebook status, a spreadsheet formula, or a YouTube comment. And I'm largely focusing on environments for computer programming, because those are the most expressive text editing environments.

Modes

Modes are a usability problem for a couple of reasons. First, because of "mode errors": if the consequences of an action such as pressing the "B" key depends on what mode you're in, and you don't know what mode you're in, you don't know what will happen when you take the action. (The classic example is when you can't figure out how to log in because your Caps Lock key is on when you're typing your password). Second, because of the "gulf of execution" — even if you know that the "B" key can cause the consequences you want in some mode, it can be hard to figure out how to get into that mode.

Emacs is mostly modeless in the sense that the same keys usually do the same thing; its "editing modes" usually have more effect on things like syntax highlighting and where the tab stops are than on what will happen if you press the "b" key, although there are exceptions. Vim, on the other hand, has a "normal mode" for most editing commands and "insert mode" and "replace mode" for actually writing stuff.

Quasimodes in common practice

Emacs uses quasimodes in its user interface to the extent that was possible with the "space cadet" keyboards that were common at MIT when it was designed in the 1970s and 1980s, which is to say that it uses "key chords". The "B" key, for example, inserts "b", except that if Shift is held, it inserts "B"; if Control is held, it moves to the left by one character; if Alt ("Meta") is held, it moves left by one word; if both Control and Alt are held, it moves left by one parenthesized expression; and if Shift is also held while using control-B ("C-b"), alt-B ("M-b"), or control-alt-B ("C-M-b"), the cursor movement creates or extends a text selection ("region"). So the "B" key can invoke eight different functions modelessly — or, let's say, quasimodally. But some of those functions require holding down as many as four keys at a time. This is commonly held to make Emacs a risk factor for repetitive stress injuries, although I don't know of any rigorous studies demonstrating this.

Space-cadet keyboards did not send any data when these

quasimode or "modifier" keys (Control, Meta, Shift, and also Hyper and Super) were pressed and released by themselves, nor when a modifier was released after pressing some modified keys. Neither did ASCII terminals of the 1970s and 1980s, and neither do ASCII terminal emulators today in 2018, 45 years later. Consequently, any editors that aspire to being used through such things are unable to take actions when a modifier key is initially pressed or when it is released, nor can they distinguish between, for example, holding Control and typing "XB" and separately typing control-X and then control-S, releasing Control in between. (Also, in ASCII terminals, the Control key collapses many characters together, typically three, so control-shift-b will do the same thing as control-b, and control-@ does the same thing as control-space, so keys distinguished only by such distinctions are typically assigned to less important commands.)

This means that some of the user-interface conveniences personal-computer users got used to during the 1980s are impossible to implement in this environment. For example, some MS-DOS programs display an on-screen menu listing the functions available on the various function keys, and the menu changes when Control, Alt, or both Control and Alt are pressed, displaying the function that will actually be invoked if you press, say, F8. Likewise, the Macintosh Key Caps applet displays a picture of the keyboard labeled with the characters that will be produced if you press a given key; the labels change if you press Shift or Option. And the task-switching key sequence in Microsoft Windows, introduced I think in Windows 95, cycles through the most-recently-used windows as long as you hold Alt and press Tab repeatedly; no window is activated until you release the Alt key.

The limits of key chording

Emacs and Vim both have a large number of commands. The Emacs I'm typing this in is currently configured with 14165 different commands, which is too many to assign to key chords on the keyboard — even if you have four modifier keys, so that each non-modifier key had 16 different chords, a 105-key keyboard would only have 1616 distinct chords in this way. Also, since the editor can't provide any visual feedback related to modifier keys that are currently held down, the memory load would be quite heavy. So, instead, they use three alternative approaches: prefix keys, a command line, and command mode.

Prefix keys

Prefix keys are sort of like modes that only last for a single keystroke. For example, in Emacs, as I said, control-B moves left by one character, and B by itself inserts "b". But control-X control-B opens up a window listing all the things that are open in Emacs, and control-X B prompts you for the name of the thing you would like to switch to. In effect, control-X puts Emacs into a mode which changes the effect of the next keystroke. Analogously, in Vim, typing "?" (as shift-/ on Qwerty keyboards) does a search backwards, but "g?" decrypts the current selection with the rot13 cipher. The "g" key puts Vim into a mode which changes the meaning of the next keystroke.

GNOME and Microsoft Windows applications can typically be

keyboard-operated by using the Alt key and a letter to select a pull-down menu, then further letters or arrow keys to select submenus or menu commands. This is also an implementation of prefix keys. It shows how modes can actually be more usable than having hundreds of commands on modified keys: it's feasible to display breadcrumbs and guidance.

However, prefix keys still cause mode errors, which are still frustrating, and even in the best cases, they don't help to reduce the gulf of execution much. Among the key sequences currently defined in my Emacs are things like control-X 8 / A (which inserts the letter "Å"), control-X control-K B (which binds the last-defined keyboard macro to a key or key sequence), control-X Enter shift-F (which changes the character encoding used for the filename of the currently open file), and over a thousand others, most of which I've never heard of even though I've been using Emacs almost daily for a quarter century. It can be hard to find out that these even exist, even though Emacs has an active community, comprehensive documentation, and many ways to explore and query.

Command lines

In Vim, you invoke the command line by typing ":"; Emacs has several command lines, including the Lisp command line invoked by alt-:, the editor-command selector invoked by alt-X, and the shell command line invoked by the command "shell". In Firefox and other web browsers, you type control-L or ⌘L to edit the URL, and in Gnumeric or Excel, you can invoke the command line with the F2 key or "=".

The command line is sort of a mode, but the B key generally causes a "b" to appear on the screen, just like in normal life, and the normal editing commands do normal editing things; the difference is that what you're editing is a command which will eventually be executed, usually when you hit Enter. Command lines can still cause mode errors where you want to be typing stuff into a document and you end up typing commands instead, but generally they support more usability features than prefix keys.

The "alt-:" command line in Emacs is for evaluating Lisp expressions, which can invoke editor commands, define new functions, or do useful computations. For example, I recently evaluated "(* 16 101)", the product of 16 and 101, to see how many commands a 105-key keyboard with four modifier keys could invoke. Earlier I evaluated "(+ (* 9 .40) (* 4 .59))", which calculated the BOM cost of a possible circuit design that used 9 chips that cost 40¢ and 4 chips that cost 59¢.

Emacs also supports a different way of evaluating Lisp expressions. Its `*scratch*` buffer is by default in an editing mode called "lisp-interaction-mode" in which the key chord control-J scans backwards from the cursor for a complete Lisp expression, evaluates it, and inserts the results into the buffer on a new line. So, instead of typing `alt-: (* 16 101) Enter`, you can switch to `*scratch*` and type `(* 16 101) control-J`, with the following result:

```
(* 16 101)
1616
```

In "lisp-interaction-mode", control-J is bound to the command "eval-print-last-sexp", which does what I explained above. Also, by default, throughout Emacs, control-X control-E is bound to "eval-last-sexp", which does the same thing except that it doesn't insert the result into the buffer. This is especially useful when you are programming in Emacs Lisp, because it allows you to point at pieces of your code and instantly run them to see if they do what you think they should.

This is a more modeless way of providing a command line; it was pioneered in Smalltalk, where "Do it!" and "Print it!" are commands available on key chords and on menus in every text editing window, and have been since the 1970s. These work on either the current selection or, if none, the current line, treating it as a Smalltalk sentence and inserting the result as a new selection. This is a little bit clumsier than the Lisp approach because Smalltalk sentences are not self-delimiting, but you also don't need it as often — my most common use of this facility in Emacs is to recompile a function I just edited so I can try it out, and Smalltalk has an even smoother way to do that.

However, in a sense precisely because it is more modeless, this approach to providing a command line has suboptimal usability. If I type `(* 16 101) control-X control-E` in this document, it isn't until the last chord that Emacs knows that `(* 16 101)` is intended as Lisp code. If I accidentally type `(*16 101)`, Emacs doesn't know to highlight `*16` as an undefined function until then. If I start typing `(backw` in this buffer and ask for completions of the partially-typed function name, Emacs doesn't know I'm intending to type Emacs Lisp and so can't complete that to `backward-` and list the 18 functions whose name starts with that. By contrast, in the alt-: command line, it can and does.

The alt-X command line in Emacs just allows you to select among the 14165 already-defined commands I mentioned earlier; if you select a command that needs arguments, it either obtains them from context (such as from the current selection) or prompts you for them afterwards (yet another mode).

The alt-X equivalent in Oberon is modeless; it works like the Smalltalk "Do it!" or Emacs `lisp-interaction-mode` — if you click a certain mouse button, it looks for a command name in the text around the point where you clicked, and if it names a procedure, it calls that procedure with no arguments. I think `File.save` is one example, although I don't have my Oberon book here (see A review of Wirth's Project Oberon book (p. 431) for my notes from reading it). The command is not permitted to prompt the user and await input, because Oberon is a single-threaded event-driven system, but it can do things like open a new window for further interaction and put more command texts into that window, or some other window.

Repetition and adjustment

If I want to type a line of asterisks on this Qwerty keyboard, it's pretty easy; I put my left pinky on a Shift key and bang on the 8 key with my right middle finger until I have the right number of asterisks, possibly adjusting at the end with a few backspaces if I overshot. By contrast, if I want a line of alternating periods and colons, it's a real pain to type by hand; I need to use my right ring finger on the period key and coordinate my two pinkies on the semicolon and Shift keys to

get the colon. Repeating a single key is much, much easier than repeating a two-key sequence, and repeating a sequence of a chord and a key is even harder.

Now, Emacs has a keyboard macro facility. Keyboard macros are good for repeating something a number of times. I can type “F3 .: F4”, with the F3 and F4 keys, to define a keyboard macro that inserts “.:”. The traditional key sequence to run the macro is control-X E, and it takes a count, so if I want a line of 20 of them, I can type “control-U 20 control-X E” and then add a trailing period:

.:.

The trouble is that it’s usually a real pain to figure out what 20 should be until you see it. (Why am I counting things when I have a computer to count things for me?)

Vim, and vi, were better at this; you could type “a.: Esc .....”, and each “.” you typed would append another “.:”. You could also type “a.: Esc 20.” or even “20a.: Esc” if you knew the count.

Emacs’s “indent-rigidly” command had the same problem; you had to figure out how many spaces you wanted to indent by ahead of time, say by multiplying 3 by 4 in your head, and type “control-U 12 control-X Tab”. Worse still was “undo”, which used to be just “control-X U”, and which you almost always want to repeat a difficult-to-calculate number of times.

Modern Emacs has a feature called “transient modes” or “transient maps” which temporarily change the meanings of only a few keys — like prefix keys, but leaving most of the key functions undisturbed. In particular, now you can type “control-X E E E” to repeat the macro three times: “.:.:.:”. (Also, you can type F4 to repeat a macro, and control-/ or control-_ to undo.) What they did with indent-rigidly is more interesting: now, if you just type control-X Tab without giving a count first, the right and left arrows indent and outdent the selected region by one space, or by a whole tab stop if you use shift. Because this mode only alters the meanings of a few keys, it rarely causes mode errors, but they do happen occasionally.

The interesting thing here is that these are ways of interactively and almost continuously varying a parameter in either direction while observing the results, much like Bret Victor’s “Kill Math” proposal (and some of his other ideas).

WordStar’s late-1970s approach to this problem, by the way, was interestingly different: you would ask it to start repeating a command, which it would do at a rate you could vary interactively with the keys 0 to 9, with a key to stop it — much like the autoscroll feature on some web sites. I suspect that this was not a viable approach on the timesharing machines where Emacs and vi were born because of the unpredictability of their response times and the high costs of context switches and terminal-screen repaints.

The Cat, LEAP and THE

The Canon Cat was a text-based personal computer whose user interface Jef Raskin designed after being pushed out of the Macintosh project at Apple. Among its daring user-interface innovations was that, not only did it have no mouse, it also had no vertical arrow keys. Instead, it had two “LEAP” keys, one for moving forward and one

for moving backward, which were quasimodal, like modifier keys, but for incremental text search, like Emacs's control-S and control-R commands. Like Emacs's control-R, when the search terminated, you would be left at the beginning of the search hit, but unlike Emacs, the search terminated when you released the LEAP key, making LEAP both faster to invoke and less error-prone — because it wasn't a mode.

Raskin claimed that experiments showed that it was substantially faster for navigating text than a mouse.

Canon mass-produced the Cat and sold a number of them, but it was pretty much a failure in the market. In later years, Raskin was working on an extended reimplementation of the concept as free software, called "The Humane Environment" or "THE", before being struck down with a few weeks' worth of warning by pancreatic cancer. THE never gained significant adoption and has largely been forgotten.

Thoughts about modifier keys for quasimodes

So, that's the historical background. Here's some wild speculation about how to apply it.

You can quite reasonably substitute quasimodes for the search modes used by Emacs and Vim; you could maybe control-F (for PARC bindings; control-S for Emacs bindings) to start a search, which would continue only as long as you held the control key. You wouldn't be able to disambiguate searching for control characters without some kind of additional escaping, but for the common cases of searching for Enter and Esc, both Vim and Emacs already need the additional escape sequence (control-V and control-Q respectively, or control-J to search for Enter in Emacs), so this would actually be an improvement. I'm not sure what to do for next and previous match other than arrow keys; maybe alt-K and alt-J.

For calculations — the thing I most use Emacs M-: for — an RPN calculator quasimode would be quite handy. You could type control-= to launch it, and it would persist as long as you held the Control key, fading away when released. Numbers would be pushed on the stack, separated by spaces; `*+-/^` would do their usual things, but `%` would do divmod; `,` could perform vector concatenation; `dxr` would dup, swap, and rotate; backspace would delete; `pqtscale` would compute π, square roots, tangents, sines, cosines, arctangents, logarithms, and exponentials; `i` would generate sequences; and `(` and `)` would respectively take numbers from the document you were editing and put numbers into it. So to double a number you would type `control-= (2*)`, although it might be snatched from somewhere other than where you were. The stack would remain persistent between invocations, and values on the stack would retain their provenance, operations being reversible with `z`. Vectors would be graphed by default, toggled with `g`. This would be substantially more convenient than the nonsense I'm using now with alt-: and running `units` in a shell.

More ideal still would be the ability to edit input values after the fact and watch the changes propagate, like in a spreadsheet. And it would be nice to have access to more mathematical functions than

there are letters, and to be able to search for them.

Of course, substituting a quasimode for the Emacs alt-X named-command invocation is straightforward; control-X would show you the available commands matching your search (and some kind of visualization of their results) and you could invoke one with Enter.

The Tab key is pretty much off limits for these things because both of my Control keys are on my left pinky.

Substituting a quasimode for the indentation stuff is also super easy; I'm currently using control-< and control-> for this, though control-H and control-L would work if I weren't using Emacs standard keybindings. Mouse movement might be best for providing the input for things like this, and also maybe navigating through undo history.

Some kind of thing that would insert a "form" into your current document, with like fields and stuff, would be super handy. Especially if you could like search through a list of currently defined forms and maybe fill in some fields with default values and already existing context data, and maybe have a command button or two to take actions that had side effects, and if the fields could recompute automatically and maybe satisfy constraints. And also if this was a generalization that included mathematical relationships (cylinder volume is $2\pi rl$, surface is $4\pi r + 2\pi rl$, from which you can deduce any two of those variables from the other two; see Relational modeling (p. 1102) for more thoughts on this) and values in mutable tables that you could assert and retract from. Maybe this is a different idea that doesn't really have to do with quasimodes at all.

Window switching, task switching, and so on is probably the most common fully-quasimodal keyboard interaction technique in use today, via the Microsoft Windows alt-Tab keybinding that has been widely adopted since then. The standard approach is to cycle through a least-recently-used list of windows, displaying thumbnails; the iswitchb-mode and ido-mode alternatives to this in Emacs are modal rather than quasimodal, but also winnow the list according to a search string as you type it. Now that I've been using iswitchb-mode for ten years (I'm trying ido-mode) this winnowing seems like the obviously right thing. But it would be better if it were quasimodal.

Similarly, term lookup (whether library functions, dictionary words, or Wikipedia articles) is a possible useful quasimode.

Multiple-cursor editing, as implemented in Sublime and IDEA, is arguably a "mode," but it may be difficult to make it a quasimode; it's sort of an alternative to keyboard macros, which means that you need to be able to use all the usual editing commands.

Mice, multitouch and quasimodes

I've focused on keyboard interfaces, but of course mouse drags and touchmoves are even more common interactions than switching windows with the mouse, and they're quasimodal. But I think quasimodes are even more promising for multitouch interaction than for keyboard interaction.

Topics

Improvising high-temperature refractory materials for pottery kilns

Kragen Javier Sitaker, 2013-05-17 (4 minutes)

We were talking today at a "natural construction techniques" workshop about how to build a ceramics-firing kiln. We'd spent the weekend making adobe from materials found onsite and making buildings out of it.

A pottery kiln needs to withstand heating to at least 1050° for earthenware and bricks, and at least 1200° for stoneware or porcelain.

The most common way to build a kiln is apparently by lining it with kaolin-derived refractory brick. But there's no kaolin in the native dirt on site. What could be done about that, aside from going to the ceramics supply store to buy some kaolin?

Well, it might be possible to extract kaolin from glossy paper; it's one of the two minerals used to give common magazine paper its gloss, the other being calcite. Or it might be possible to mill porcelain to powder (known as "grog" when it's sand-sized, or "pitchers" when coarser) and then sinter the powder. But I think wet milled porcelain powder isn't going to have the plasticity of kaolin. You'd have to stick it together with something that will hold it in place until it fires.

My first idea for such a binder was salt, because I had the mistaken idea that sodium chloride didn't melt until 1500° or so. But in fact its melting point is only 801°. But kaolin won't sinter until 1200° or more. It's possible that the salt-binder idea might happen to work --- if the salt lets the porcelain chunks sinter at a lower temperature, but doesn't flux them enough for them to melt entirely at the temperatures the refractory must withstand --- but it's likely to fail one way or the other.

A somewhat more likely binder candidate is simply the other clays available. You'd mix a small amount of clay into a large amount of milled porcelain.

You might be able to press the milled porcelain into bricks using something like a compressed-earth brick press, then firing the bricks. This way, the porcelain particles are in direct contact with each other, and there's no possible flux in the way. Achieving porosity might be difficult, because I think you couldn't simply mix in sawdust, the normal way; its elasticity would force the porcelain particles apart when you took the brick press. Charcoal or carbon black might work.

Another possible alternative would be to make the refractory bricks from some other mineral found onsite. Quartz, for example, is abundant almost anywhere, and if you can purify it and get it to sinter, its melting point is 1670°; in theory, you could also turn quartz and alumina into kaolin, but I have no idea how. ("Chemical weathering of rock", says Wikipedia. But what reactions chemically weather quartz and corundum into dust?) Magnesium oxide (periclase) might work? It doesn't melt until 2852°, and you can presumably make it from electrolyzed seawater, but I don't know

how to separate it out from dirt, or if it's common.

The traditional refractory used for gas lantern mantles is a mixture of uranium and thorium oxides, made from monazite, a common ingredient of sand. However, even if we could practically separate those out (separating the monazite should be easy because of its high density, but separating the rare earths involves lye and hydrochloric acid at above-boiling temperatures), people might complain that the furnace was too radioactive.

Calcium oxide (quicklime) doesn't melt until 2572° and would therefore work, and it's easily made by heating limestone to 825°, if there's limestone onsite, which there probably is. However, it's dangerously reactive with human bodies and other sources of water.

Magnetite is maybe the easiest mineral to separate out of sand. It oxidizes to hematite, ferric oxide, if heated with oxygen, and that doesn't melt until 1566°. I haven't heard of it being used for refractories, but it seems like it would work.

Topics

- Materials (p. 3560) (112 notes)
- Ceramic (p. 3371) (17 notes)
- Kilns (p. 3538) (8 notes)
- Construction (p. 3388) (5 notes)
- Clays

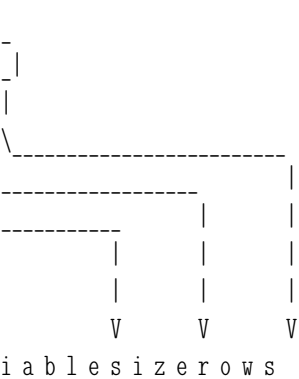
Raggedcolumns

Kragen Javier Sitaker, 2015-08-28 (3 minutes)

The standard C way of doing two-dimensional arrays accommodates two possibilities in the same syntax `c[i][j]`: arrays of arrays like `int c[10][7];` and arrays of pointers like `int *c[10];`. The first representation wastes no memory on pointers, but requires each row to be the same length, while the second one allows each row to be of a different length, but spends a pointer and often a dynamic memory allocation on each row. Also, it needs to store the lengths of the rows somewhere, unless they're NUL-terminated strings or something, which has problems of its own. So the overhead is something like two or three words per row.

Historically, array languages like APL and libraries like Numpy have preferred rectangular N-dimensional arrays rather than ragged ones. This isn't ideal for things like lists of strings, where each string may be of a different size. I've been thinking about how to extend array languages consistently to such objects, both in abstract semantics and in practical machine implementation.

Speaking of practical implementation, it occurred to me that if your N rows are stored one after another in a contiguous block, as in the C array-of-arrays case, but you have a pointer to the beginning of each one, as in the C array-of-pointers case, then for N+1 pointers, you can store N variable-size rows. Like this:



Here, we've stored four strings with no delimiters between them, with an array of five pointers. This allows optimally efficient multidimensional indexing with range-checking.

(As a digression, if you're really hard up for space, you could block the array into 16-string-or-so blocks, store an array of block start offsets, and store the 16 string lengths in another array, either limiting each individual string to 64KiB or 256 bytes or something, or using a variable-length encoding in the other array and an array of pointers to every 16th string length. Then a typical block of 16 short strings like the above will cost you about 64 bytes of string data, 4 bytes of offset into the array, 4 bytes of offset into the lengths array, and 16 bytes of string length data, for a total of 88 bytes. That is, assuming that no single array is over 4 GiB.)

This ragged-array thing can be multidimensional: you can store a list of variable-length lists of variable-length strings in the same way, with the first level of pointers pointing to the beginning of each string list pointer array, and the pointers in the string list pointer array pointing to the string starts. Only three allocations and all the lengths available at run-time.

Additionally, and importantly for use in array languages that tend to construct new large arrays by applying operations to existing large arrays, this multidimensional ragged structure can be constructed incrementally, one character at a time, without knowing ahead of time the size of any of the rows.

Topics

• Programming (p. 3658) (286 notes)
• Arrays (p. 3326) (17 notes)

Clisweep

Kragen Javier Sitaker, 2018-06-06 (3 minutes)

The first relatively modern file manager I used was a thing on CP/M called "NSWEEP". Like the CP/M command line, it was designed essentially for a teletype interface, but it used a completely different paradigm, one rather similar to the Macintosh Finder, which I think came somewhat later: it displayed you a list of files (one at a time, normally), and you would issue it keystroke commands to invoke operations on the current selection — normally a single file, but optionally some set of "tagged" files. ("The selection" was not, I think, NSWEEP's terminology; it's the modern terminology.) To respect the speed limitations of the teletype interface, it would normally only display you a single filename after each action — most actions, including tagging and untagging, moved to the next file, but you could also move to the previous file. And, if I recall correctly, there was a command to list all the files (with asterisks to indicate whether they were tagged) without requiring keystrokes in between each one.

This was about halfway to the direct-manipulation noun-verb user interface the Macintosh Finder used, which was of course prefigured by GUI work on the Alto and Star at PARC, which probably preceded and may have inspired NSWEEP. You would first select the file you wanted to act on, perhaps by moving down to it one line at a time with keystrokes (I think Enter or N would move to the next file?), then invoke the action you wanted to invoke on it. The Unix editor `ed` worked fairly similar, but with lines of text in a file rather than entire files, and with less feedback by default.

Speaking of strange user interfaces, the MH mail reader was, in the late 1980s and early 1990s, a relatively popular way to read email. I never used it, but as I understand it, you had short commands to do things like list the messages in the current folder, display the next and previous message in a folder, reply to the current message, or move it to another folder. The novelty is that these commands were invoked from the Unix command line, rather than a separate CLI like Berkeley mailx, or a screen interface like elm or mutt. So it was easier to script them, and you changed folders with the usual `cd` command, and so on.

So I was thinking it would be interesting to try something like this for file management: a set of commands you could use from the Unix command line that would give you a noun-verb next-prev kind of user interface for file management. And I was thinking it would be nice to support some kind of hashtags for saving, editing, and combining previously defined selections, since it's 2018.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Cli

Food storage

Kragen Javier Sitaker, 2013-05-11 (updated 2013-05-17) (54 minutes)

Storing food at home

A related post, Only a constant factor worse (p. 1648), discussed a food-buying strategy that would give you an unboundedly-expanding stored-food supply, as long as you're spending more than about US$0.64 per day (US$234/year) on food, and advocated doing so if you're poor, for several reasons: you can buy food only when it's on sale or free, you can buy food in bulk for better prices, and you can keep a wide variety of food on hand to avoid dangerous dietary monotony. Even if you accept that it would be a good idea, it's reasonable to ask how you can practically store such a stockpile.

First off, let's cap the stockpile at a year's supply. Not even the Mormons store up more than a year's supply of food; many people just store up 72 hours' worth for disaster-preparedness. So let's not even consider the possibility of storing two, five, or ten years' worth of food.

Second, let's consider how much space we can reasonably use. I think it's reasonable to dedicate some constant fraction of your living space to food storage, not too much. If 25% of your house is full of stored food, it's likely to significantly diminish your quality of life. Let's figure 6.25% as a reasonable upper limit.

Would you suffer a lot if your house were 6.25% smaller? The smallest house I've lived in was a Volkswagen Vanagon camper bus, which was about two meters by four meters by one and a half meters (although you could put the top up to get a ceiling of over two meters in part of the bus when you were stopped). Getting at daily-use stuff was trivially easy; it was almost always within arm's reach. Getting at more specialized stuff (tools for putting the damn muffler back on again, say) required reconfiguring the furniture, which was kind of a pain.

That was 12 cubic meters of space. 6.25% of it would be three-quarters of a cubic meter, which was about the size of the humongous toolbox we lugged around to fix the thing. Another three-quarters of a cubic meter would have been feasible, but it would have blocked the back windshield; we would have had to put it on the floor when driving or sleeping, and in the bed space when we needed the floor. That is, we could have done it, but it would have been a pain.

However, if your house contains 12¾ cubic meters, you can damn well spare those extra three-quarters of a cubic meter for food storage. If that's useful, which it is, and if it's necessary, which as we'll see, it isn't.

The apartment I'm living in at the moment has a living room, a kitchen, a bathroom, and a storage room, which was intended as the bedroom but has a weird mold smell, so I don't want to sleep in it. I basically live in the living room, which is about four meters by six meters by three meters tall, a total of 72 cubic meters, six times the size of the Vanagon. 6.25% of this would be 4.5 cubic meters; if it's

vertical to the ceiling, it would be 1.5 square meters; along the four-meter-long wall, it would shorten the six meters of the room by 38 centimeters. Eminently livable. How much food can you store in 4.5 cubic meters?

In that related post, I was talking about foods like flour, polenta, brown rice, soybeans, salt, and sunflower oil, with total consumption between 600g and 700g per day. These have somewhat varying densities, but they're generally around the density of water: 1kg/ℓ, or one tonne per cubic meter. Which means you can store about four or five tonnes of food in 4.5 cubic meters. That would be about 19 years' worth!

In fact, storing 650 grams of dry food per day, a year's worth of food is only 237 kg. That's not even the three-quarters of a cubic meter that would be 6.25% of the Vanagon. It's more like *one* quarter of a cubic meter.

Lifetime and spoilage

This brings us to the problem of the lifetime of stored food. If you just buy bags of food and pile it up in a corner, you will not have a food stockpile that eliminates risks to your food budget; you will have a serious vermin infestation, maybe a mold problem, and probably a lot of expired food. Plus, if you have to move, you'll have one more heavy thing to bring with you. At different times and in different places, I've had to cope with cockroaches, rancid oil, weevils, red flour beetles (or confused flour beetles), and moths in my food. This is where I explain how you can make sure you never have these problems.

I know nine basic strategies for managing the limited-food-lifetime problem:

- Rotation: first in, first out.
- Hermetic sealing.
- Vermin-proof packaging.
- Cold.
- Dehydration.
- Darkness.
- Chemical preservation.
- Sterility.
- Liveness.

Rotation

Things like rice are typically guaranteed good for 18 months from packaging. Even unopened, unrefrigerated mayonnaise is guaranteed good for nine months. So if you consume your rice within 18 months of buying it, you're all good. But that means that, if you open up the pantry and pick between a two-month-old bag and a 17-month-old container of rice, you'd better pick the 17-month-old one. Otherwise, it's going to expire before you finish the last one.

This is known as "proper stock rotation" or "first in, first out" in business. You need to mark each item you have in stock with a date (either its date of origin or its date of expiration) and always use the oldest item.

That means that, if you manage to work your way up to a one-year stockpile, you'll always be eating year-old food. The alternative is to keep buying more food than you need and just throw out the

year-old food, wasting it.

You can mark items in lots of ways. I like to use polyester cloth ribbon, tied around the necks of bottles, marked in ballpoint pen. It looks pretty, it's easy to read, it's easy to make, and it's pretty cheap.

Hermetic sealing

Most spoilage problems are caused by stuff getting into food from the outside: moisture, bugs, or in the case of rancidity, mere oxygen. So to keep your food from spoiling, it helps a lot to keep it hermetically sealed. Historically, this was really difficult, and that was a major cause of food spoilage. Now, there are lots of ways to do this: heat-sealed bags, cans, jars with hermetically-sealing tops, Ziploc bags, bottles with corks, and my favorite, screw-top soft-drink bottles.

Screw-top bottles typically have a soft plastic gasket on the inside of the screw-top lid, which the thread compresses against the top of the bottle neck. This seal must be sufficiently airtight to keep the soft drink from going flat. Some, but not all, bottled-water bottles are inferior. You can test a candidate bottle by filling it with water and freezing it, which will stretch the bottle and put the contents under a lot of pressure; if the bottlecap doesn't seal well, the last of the water will be forced out and drip past the cap, forming an icicle in your freezer.

One thing to beware of is that hermetic sealing means that odors and contamination won't have a chance to dissipate. This means that if your hermetic container is leaching stuff into your food, it will build up, rather than bubbling out. This is why "tin cans" traditionally were coated in tin and nowadays are coated in plastic: to keep the steel from ending up in the food. Soft-drink bottles probably won't leach anything toxic into your food (they don't contain bisphenol-A, and the concerns about leaching of antimony-based catalysts have not proven out so far) but they might make it taste like Coca-Cola.

Oxygen rancidifies oils and helps to spoil beer (I think by souring it), and most plastics are fairly permeable to oxygen, including the polyethylene terephthalate used for soft-drink bottles and the polyethylene and polypropylene used for most plastic bags. This is the major reason that beer has not been sold in plastic bottles until recently. Some new formulations of polyethylene terephthalate include a bentonite clay filler to dramatically reduce their oxygen permeability, which has made it possible for plastic-bottled beer to keep. I have no idea how to tell if you have such a bottle. Multiple layers of enclosure, perhaps including a layer that's not plastic, may be a useful way to reduce oxygen permeability.

Latex condoms --- preferably unlubricated --- or gloves may provide a useful level of additional sealing for storage that you don't expect to open and close repeatedly. They have the advantage that they can be stored in very little space when empty.

Vermin-proof packaging

You can have a container that's hermetically sealed but not vermin-proof. In particular, weevils and flour beetles can tunnel through even fairly thick polyethylene plastic bags, so if one bag in a pile is infested, they will eventually colonize the other bags. So far, I haven't had them chew through soft-drink bottles. (This is one of the

reasons soft-drink bottles are my favorite.) I'm sure they can't chew through ceramic, glass, or metal.

Some varieties of cockroaches can chew through wood, so wood may not be a good choice for vermin-proof packaging. It's hard to get wood to seal hermetically anyway.

The Anasazi had a kind of container called a "seed pot" which was vermin-proof but not hermetically sealed. It was a small pot with a tiny hole on top --- maybe 2mm in diameter --- to fill it with dried seeds. Apparently the vermin they had to deal with were mostly mice, not flour beetles, and mice are far too big to get in through such a hole. You break the pot to get access to the seeds.

Mice and rats will not have much trouble chewing through soft-drink bottles, I think, so if you have to deal with them, it might be a good idea to keep your soft-drink bottles inside something mouse-proof, like a metal box.

Because your food might already have vermin in it when you put it into storage, it's a good idea to have at least one level of vermin-proof packaging that's fairly small, say, around a kilogram or so. It would be a real shame if the progeny of two tiny flour beetles were able to spoil 20 kilograms of your stored-up flour. The Anasazi seed pots held about 100 grams. I typically use soft-drink bottles of 1½ liters, although I occasonally use 2¼ℓ and 3ℓ bottles too.

Cold

Most kinds of spoilage follow the Arrhenius law, increasing exponentially with temperature; so dropping the temperature even a few degrees can slow a spoilage process to the point where it's effectively stopped --- where it won't proceed to the point of mattering by the time you eat the food. This also means that what matters is not the *average* temperature at which food is stored, but the *maximum* temperature, so simply reducing the size of temperature variations will extend the lifetime of food.

Refrigerators, of course, are the most ubiquitous manifestation of this storage strategy in the modern world, but wine cellars are another. Deep-freezes --- large, low-temperature freezers that open on top rather than to the side --- are among the most effective ways of storing food.

I've speculated in the past on kragen-tol about using a superinsulated icebox in place of a refrigerator, and have actually done some experiments with freezing two soft-drink two-liter bottles of water in a freezer and transferring them daily to a 2cm-thick styrofoam cooler in the Buenos Aires summer. Heat leakage was enough to melt about one of them per day. This suggests that 10 or 20 centimeters of styrofoam, or twice that of straw, would be sufficient to give you a low-cost major expansion of your refrigerator capacity, at the cost of transferring ice bottles once or a few times a week.

I also recently did some calculations about using a drying-closet to evaporatively cool air to near the wet-bulb temperature, using the temperature change of the air to drive a constant airflow down a cold-air drain, thus accelerating the drying process. Preliminary psychrometric-table consultations suggest that this could lower the temperature of whatever is below the drain by some 5 degrees under ordinary circumstances.

A pot-in-pot evaporative cooler design won some design awards a few years back. Basically you fill the space between an inner terra-cotta pot and an outer terra-cotta pot with sand, and you keep the sand damp with a watering can so that the inner pot remains at the wet-bulb temperature. This doesn't keep things as cold as an actual refrigerator --- and in humid places it doesn't do much at all --- but in many circumstances it can preserve food substantially longer than nothing.

Dehydration

Dehydrating food for storage has some drawbacks --- the flavor and texture changes, and in most food, not for the better; you typically need to rehydrate it to cook it; and it can be labor-intensive, since only occasionally will food dehydrate without human intervention. Usually you have to spread it out, perhaps cut it into slices, perhaps warm it up, and perhaps drive airflow over it. On the other hand, it dramatically reduces the mass of food, making it more portable; and only a very few kinds of spoilage can attack dry food: rancidity, of course; weevils, flour beetles, and moths, which can get sufficient water by digesting its carbohydrates; and large vermin such as mice and rats, which I fortunately have never yet had to deal with, can drink water elsewhere and then eat your food.

Lack of proper dehydration for storage is a major risk factor for cancer, because aflatoxin-producing molds will infest grains and legumes that are stored insufficiently dry.

One thing to keep in mind is that, once vermin have infested food, they tend to moisten it. Flour beetles and weevils are not, in themselves, toxic, although flour beetles do produce nasty flavors --- but if present in sufficiently high quantities, they will moisten the remaining food enough that it will spoil soon, even if you kill them. (Some people think they add protein to the food, but they do not. They probably do slightly increase the protein fraction of the food, but by reducing its carbohydrate content, not by increasing its protein content. Animals, such as humans and confused flour beetles, do not produce protein. We must get it from our food.)

Some foods have a hard, dry shell protecting a tasty inner core, such as nuts, flax seeds, sesame seeds, sunflower seeds (unshelled), and so on. This only works as long as the outer shell remains dry.

I speculate that empanadas and their kin in other lands originally gained popularity in part because of long shelf life. The surface of the empanada is completely dry when you finish cooking it, and it's impregnated with oil to keep it from moistening from either within or without. The interior, having just been cooked thoroughly, is sterile. I suspect that an unpunctured empanada, especially of the fried type, would have a shelf life measured in weeks or months if you can keep vermin off of it, but I have not yet dared to do the experiment.

Darkness

A few kinds of spoilage are caused by exposure to light, especially ultraviolet light. Potatoes exposed to light will produce potentially deadly levels of solanine and turn green; beer exposed to light will sour faster; and I think oils exposed to light will rancidify faster. Refrigerators conveniently provide darkness, but other kinds of enclosures may also be useful. And, of course, light produces heat,

which can dispel cold and accelerate spoilage.

Even a cardboard box with holes in it can provide useful levels of darkness.

Chemical preservation

Aside from the obvious and noxious chemical preservation methods, using nitrites, sulfites, formaldehyde, and so on, we have jellying, salting, pickling, alkaline preservation, spicing, and antioxidants --- methods which were more ubiquitous before the advent of refrigeration, but which still extend food lifetimes in concert with refrigeration.

Jelly, jam, syrup, and preserves have too much sugar for much life to survive in them, although they can sometimes grow mold on their surface, where perhaps condensation locally reduced the relative sugar content. I think this kind of preservation works by reducing the osmotic pressure to the point that any invading microorganisms are sucked dry. (Unlike with my stores of beans, rice, legumes, flour, and spices, I've never had an insect infestation in my sugar; the most that happens is that ants may come to carry it back to their anthill, grain by grain.)

Salting works similarly, by reducing osmotic pressure. Thus both jellying and salting work synergetically with dehydration --- dehydration reduces the salt or sugar needed, and salt or sugar reduces the dehydration needed, to reach a given level of longevity.

Mayonnaise is, in part, a way to pickle eggs so they can last for months without refrigeration; and many vegetables can be enjoyed for months or years by pickling them, producing an environment too acid for bacteria to grow in.

Alkaline preservation is not as widely used, but, for instance, from the stench, I think thousand-year eggs are preserved with ammonia, and lutefisk, also notoriously stinky, is preserved and softened with lye.

Many spices are poisonous enough to retard food spoilage. Capsaicin, which makes chile peppers hot, is also a fungicide; and cloves and bay leaves, among others, are sufficiently poisonous to discourage many kinds of spoilage-causing organisms.

Antioxidants such as BHA, BHT, and vitamins A, C, and E (IIRC) can slow down rancidification dramatically, even in the presence of ample oxygen. Palm and camellia oils supposedly have very long shelf lives in part because of their high levels of naturally present antioxidants. I don't know if you can use this property of theirs to impart longer shelf lives to other prepared foods. (Nitrites, I think, are antioxidants too, which is why they keep preserved meat pink. But, in meat, they generate carcinogenic nitrosamines, and the consumption of nitrite-containing meat is associated with a significantly higher risk of stomach cancer, so I don't think the risk is worth it.)

Sterility

Many food spoilage problems are caused by life forms, like beetles, bacteria, or fungi; some other food health problems are caused by the survival of parasites in the food. Contrary to the medieval and earlier hypothesis of spontaneous generation, if these life forms are initially absent from the food and have no opportunity to get into it, they will never be present, and the food will never experience that kind of

spoilage. (It might still get eaten by rats or go rancid.)

So ensuring that the food contains no spoilage-causing life forms is one way to prevent it from spoiling. This is, of course, the main thrust of canning: you cook the can thoroughly after sealing it. Approaches other than cooking include irradiation (sadly, not practical to do at home) and deep-freezing. If you have a small number of weevils or other beetles in your flour or rice or whatever, freezing it for a couple of days can be sufficient to kill them. (If you have a large number, you may still want to freeze the container before you throw it out, to keep them from spreading from it.)

Liveness

In the opposite direction, many foods are still alive when you harvest them, for example because they are seeds or fruits. They have their own immune systems to fight off disease and spoilage, for a while at least. Green beans can survive and remain free of spoilage for weeks under even fairly unfavorable conditions. Winter squash and potatoes can last for months.

Physical structure

So suppose you have 250 kilograms of dry food with a density of 1kg/ℓ. You need to rotate it properly so as to keep it from expiring, and keep it in hermetically-sealed vermin-proof containers of around a kilogram each, ideally in the dark, at a consistent, low temperature. How do you do it?

A big part of the solution is to store most of the food in 1¼-ℓ soft-drink bottles, making sure it's in pieces small enough to fit through the necks without jamming. These bottles are about 32 cm tall and about 9 cm across. You need about 200 of them, or to allow for the fact that some of them will be partly empty at any given time, 300.

These bottles have a few other advantages over other alternative food storage methods that I haven't mentioned above: they're very difficult to break (I've watched a bus drive over an empty soft-drink bottle without breaking it), they're designed to be food-safe, they're very lightweight, they come in a variety of pleasing shapes and colors, they can be deflated when empty for compact storage and transport (at some cost to their sturdiness and pleasing shape), and they're free, in the sense that they're discarded in huge numbers.

300 of these bottles standing next to each other on the floor would occupy a rhombus consisting of two equilateral triangles some 162 cm on a side, which would actually hold 324 such bottles; its total area is about 2.3 square meters, for a total volume of about 0.73 cubic meters. (That means this arrangement is mostly empty space, which is because the bottles are partly empty and in themselves only occupy about 61% of the volume of the hexagonal prism enclosing them).

The coffee-table pantry

These 2.3 square meters (say, a rectangle of 1.6 meters by 1.4 meters) could fit inside a coffee table, for example, which could have doors on the side that open or a top that swings open. 162 cm provides 18 bottles per long row and 17 bottles per short row; 140 cm provides 18 rows, half short and half long. The total is then 315 bottles. A hexagonal or rhomboid coffee table would be both more space-efficient and more entertaining. A few centimeters of

styrofoam around the outside and thermal mass such as sand in the bottom would go a long way to keeping temperatures consistent. (If you labeled the bottle bottoms, you could stick them neck-down into the sand, so the sand wouldn't diminish the bottle storage capacity. An average depth of 5cm of sand would need 114 liters of sand, which would weigh about 260 kg, doubling the mass of the coffee table, and probably roughly doubling its thermal mass as well.)

The curtained-wall pantry

What if you want to shelve the bottles instead? If you divided that area into eight equal-area shelves (spaced 37½ cm apart, floor to ceiling) each shelf would need to be about 0.28 square meters. At a minimal shelf depth of 9 cm --- one bottle --- it would cover a bit over three meters of three-meter-high wall space. If you extended the 9-cm shelving to fill the whole four-meter wall of my living room, you'd have space for 44 bottles per shelf, 352 in all. You'd probably want to hang a curtain in front of it to provide insulation, darkness, and aesthetics. (Although orderly rows of shapely bottles filled with dried foodstuffs of varying visual textures has its own aesthetic appeal.)

The loft pantry

While both of these approaches are livable, they still do reduce the available living space somewhat. Your 500-kilogram coffee table would be a hell of a thing to stub your toe on. An approach that would enhance rather than compromise the architectural merits of this living room would be an overhead loft space for the bottles: a single shelf of the 2.3 square meters, some 40 cm below the ceiling, would be 47 centimeters wide, providing an alcove at one end of the living room. Rather than ensmallening the living space, this would just transform a part of it into a cozy nook, enhancing the rest of the room. The ceiling in the nook might be 2½ meters high instead of the 3 meters of the rest of the room.

The chest pantry

If you wanted to reduce the physical dimensions of your year's worth of stored food to a minimum without abandoning the bottle strategy, for example to make it more portable, you could probably have an upper layer of bottles neck-down, sticking through round holes in an upper shelf, with their conical necks nestling into the gaps between the necks of the bottles in the lower layer. Arranged this way, I think the second layer of bottles would add only about 22 cm of height to the thing, for a total of about 54 cm high in 1.14 square meters of floor area: 107 cm square, or a hexagon of around 120 cm from corner to corner.

You'd probably want to build the upper shelf in four to six pieces that could be lifted separately, since it would contain 125 kg of food when full, and you'd need to lift it off to access the bottles below. Each of four pieces would weigh at most 32 kg, which most people can lift safely.

Adding a third layer of bottles neck up would increase the total height to some 86 cm and diminish the floor area to about 87 cm by 87 cm. This is probably close to the minimal-dimension configuration, so let's work out the details a bit more.

If you have at least 300 bottles, at least 100 in each layer, you could

make the bottom and top layer hexagons of 127 bottles --- 7 bottles on each side of the hexagon, 13 from corner to corner. Then the middle layer would be an uneven hexagon of 101 bottles.

This gives you a total of 355 bottles, a distance across the corners of 117 cm, and a distance across the flats of 101 cm. Not as small as you might hope.

But 355 seems a little excessive if we just want to accommodate 200 bottles plus some extra to compensate for some bottles being partly empty. Suppose we use smaller hexagons: 6 bottles on each side of the regular top and bottom hexagons, for a total of 91 each, and 75 in the irregular middle layer:

Then we have 257 bottles, still 86 cm tall, 11 bottles (99 cm) across the corners, and about 86 cm across the flats. This is probably the minimal-dimension configuration.

With the Terma bottles I'm using for my estimates here, you can get a full 10 cm of interpenetration of layers by putting the second layer neck down. But without putting the second layer neck down, just spreading out the spacing of the bottles by about half a centimeter, you can still get about 8 cm, and you can get it with both of the upper layers, instead of just one. This turns out not to be an improvement.

If you actually built this chest full of potentially 321 liters of stored food, you'd probably want to put it on heavy casters so you could move it around. Otherwise it still wouldn't be very portable unless it was mostly empty. 320 kilograms is not a weight you can reasonably lift.

The underground pantry

If you don't live up in the air somewhere (I'm about four meters off the ground as I write this), you can get extremely consistent temperatures, constant darkness, and little oxygen just by burying things less than a meter into the ground. Many vegetables can keep at least a year this way, and roots like potatoes, ginger, onions, beets, manioc, and turnips have the additional advantage that they will do the work of burying themselves for you, if you just let them grow. And if vermin attack your root vegetables, well, it's a shame, but at least they're not inside your house.

Geopolitically, root vegetables are credited as a major force in peasants' attempts to evade conquest and taxation in, for example, Zomia (if I recall the name correctly) and Ireland. You can't burn a field of potatoes, you can't tell how many potatoes there are in it, and you can't confiscate them if the farmer won't tell you where they are.

I suspect that you can get some of these root vegetables to last longer than a year by burying them deeper so they don't sprout in the spring.

Growing food at home

This brings me to growing food at home, instead of storing it. Among my earliest memories, I remember my mother plowing the back yard with a Roto-Tiller to plant a garden, mostly corn. All my life she's had at least tomato plants, and often sunflowers too; and last I saw, my father (who left her decades ago) had a few rows of corn in his backyard in Minnesota.

That is not the way to save time and money. That's gardening as a hobby, not gardening for frugality and resilience.

To garden for frugality and resilience, you need to focus on the things that provide the highest return --- the least effort for the greatest benefit. As I documented in the other post, you can buy all the corn you can eat for two years for about US$200, at retail. Unless you have less than a few hundred dollars a year to spend on food, or you're preparing for a total collapse of civilization, you shouldn't spend your scarce gardening time on raising more corn or potatoes.

Instead, focus on foods that require very little effort and very little land area, and are difficult to store instead of growing. Don't grow anything you can easily buy dry in quantities you'd eat in a week. For example, here in Buenos Aires's temperate climate:

- Sprouts. My first attempt at sprouting turned about 100g of dry, boring, high-carbohydrate mung beans into about 400g of crunchy, fresh, low-carbohydrate, high-protein, juicy stir-fryable vegetables, which lasted several days. It took a few minutes a day for three or four days. My second attempt turned about 100g of dry, boring, high-protein soybeans into a moldy mess. If I figure out how to do this right, I'll be eating some bean sprouts every week, year round.
- Rosemary. It's nearly impossible to kill, and although you can dry it for storage, dry rosemary is a pale shadow of fresh rosemary. A single plant in a pot can give you enough rosemary to spice every meal. I think I've heard you can use it to keep moths out of clothing, too, but I haven't tested that.
- Hot peppers. Really hot varieties can also provide you more than enough spice for every meal from a single plant in a pot. Some

varieties are also spectacularly gorgeous.

• Mints. Similarly, a small pot can provide enough mint to spice every meal, and it's hard to kill; the usual problem is that if it's not in a pot, you have to police it aggressively to keep it from choking out everything else in the area. There are a lot of different mint varieties with widely varying flavors.

• Basil. Likewise, but you'll need several large pots rather than one small one if you like pesto. Try to keep it from going to seed.

• Squash. Squash vines are also quite aggressive and hard to kill, and when their season comes, they tend to produce volumes of squash that go beyond anything reasonable. You can't really raise them in a small pot. Summer squash, once picked, goes bad quickly enough that I fear buying it if I'm not planning to use it immediately, but I think it stays good for weeks on the vine.

• Nasturtium. Better known as a decorative flower, it's also edible with a kind of mustard-like spicy flavor. Both the leaves and the flowers are edible, and the flowers are gorgeous. It requires very little care.

• Bay, aka laurel. It's a hardy evergreen tree, and its leaves can be used to keep moths away and to spice food; they're strong enough that you usually only need a fraction of a leaf per serving.

• Tomatoes. If you have a few meters of fence that you can plant thickly with tomato vines, when they come into season, you can easily have thousands of tomatoes. This requires more care than the other plants mentioned, since the tomatoes are pretty attractive to pests, and they don't keep nearly as long on the vine as the other plants mentioned.

• Other herbs: oregano, parsley, cilantro, thyme, sage, lavender, lemon verbena, and so on. Many herbs are much better fresh than dry, will rot if you cut them and don't dry them, and can't be bought in small enough quantities for a single meal. If you have a dozen pots of a dozen different herbs, you can choose among a dozen different fresh herbs for every meal. This can go a long way toward making frugal food palatable.

• I've heard that chard also has pretty good frugality performance, is dead easy to grow, and grows year round, but I haven't tried growing it yet myself.

• Olives, if you inherited an olive tree. Otherwise, maybe plant one and hope your grandkids will benefit.

• Mushrooms, maybe? I don't know how to grow mushrooms, but they don't need access to sunlight, they can live on garbage (right?) and one or two of them really dramatically improve the flavor of a lot of foods. But they don't last long, and you often can't buy just one or two.

• Blackberries, if you have a lot of land, or mulberries. Blackberries are thorny, but otherwise easy to pick, and they're very hard to kill; you don't need to worry about pampering them. They make a wonderful jam, which you can eat year-round if you can it when they're in season. I haven't seen them growing in Buenos Aires, but they grew wild and abundant in the Presidio in San Francisco, whose climate is similar. Mulberry trees are also quite hardy once established --- IIRC they can grow deep taproots to survive long droughts --- but they take years to mature, and your chances of getting a male tree that produces no fruit is 50-50. Mulberries essentially cannot be

bought, because they don't last long enough once picked.

Storing water at home

Stockpiling food, as explained above, is a potent weapon against poverty and geopolitical uncertainty. Stockpiling water will probably not help you get out of poverty, since you already buy it in bulk, you don't benefit from variety, and it's never on sale. And water supplies typically remain available anywhere inhabited by people, except for short periods of time, even when your city is being bombed to smithereens.

A week's worth of water in the bathtub

Sometimes, however, a "short period of time" might be three or four days, or a week, due to nothing more serious than a power outage. A sufficient stockpile of water can improve your resilience against power outages. It's really hard to cook dried food without water.

At Burning Man, we had to bring our own water. The recommendation was 8 ℓ/day, for cooking, drinking, and showering in an extremely hot, dry climate. A week's worth would then be 56 ℓ. That's about half a bathtub full, and if you have a bathtub and forewarning of a possible interruption, filling up the bathtub is a reasonable response.

If you want to be prepared for a week-long water interruption without forewarning, it's not reasonable to keep your bathtub full of clean water all the time. The water will go stagnant and breed algae, mold, and mosquitoes. You'll stink from not bathing. Instead, you need containers, you need to be able to seal them, and you need to remember to rotate them.

Water bottles

For Burning Man, we actually used 56-liter military-surplus bacteriostatic plastic water bottles. Rotating 56 liters of water every few months is going to be a huge pain in the ass if you're trying to use containers much smaller than this. 20-liter bottles might be reasonable, and they're a lot easier to move around than the 56-liter ones are. 3-liter bottles are not reasonable.

Barrels

A 200-liter barrel is probably a thoroughly reasonable and widely available solution. Food-safe, sturdy 200-liter plastic barrels with sealing screw tops are a standard item all over the world, as are devices for moving them around when they're full. They go for about US$10 to US$20 used in the US.

For your house, though, you'll probably want to fill and empty the barrel with a hose.

Soy-sauce bottles

A more ghetto, but also quite effective, container is the 20-liter soy-sauce bottle, which is small enough that you can carry it around when full, rectangular, and unlike the mil-surplus bottles we took to Burning Man, stackable. I wash my laundry in a bucket cut from the bottom of one of these that I found near a dumpster outside a Chinese restaurant last month, labeled "DO NOT REUSE". They're made of thoroughly unbreakable polyethylene, they have handles to lug them

around with, they're food-safe, and originally they ship with a hermetically-sealed screw top. They're not bacteriostatic, though, and the one I fished out of the trash stream had been long separated from its screw top.

Three 20-liter soy-sauce bottles, a week's supply of water, would occupy a little less space than the year's supply of food considered earlier.

Chlorination

It's recommended to chlorinate your stored drinking water with bleach to a level where it's not really safe to drink, I think 100ppm or 200ppm. Water thus chlorinated should last for months or years. (I forget what the official recommendation on rotation for water thus chlorinated is, but I think it's six months or a year.) Before you drink it, you dechlorinate it by letting it sit in an open jar overnight. (I think you can boil it, too, if you're in a hurry.)

There are other poisons you could put in the water to keep it sterile, but they're harder to remove.

Storing a year's worth of water

If you wanted to store enough drinking water for a year, perhaps because you were insane, the 8 liters a day multiplies out to 2920 liters for the year: fifteen 200-liter barrels whose contents you'll have to rotate once or twice a year. This is three tonnes of water, occupying three cubic meters. This would occupy less than 6.25% of my living room, so it might be a reasonable thing to have in your house --- or, better, outside your house where it won't wreck everything if it springs a leak --- if you had some other use for it, other than preparing for a disaster that is unlikely ever to happen.

For example, three tonnes of water is a fairly large thermal mass, and eminently capable of internal convection. Raising or lowering its temperature by a degree requires 13 megajoules, which is 3.5 kilowatt hours. Freezing or thawing it requires almost a gigajoule, 260 kilowatt hours. This could allow you to time-shift the use of energy to warm or cool something to a time of day when energy is easily available, or to harvest ambient temperature at a reasonable time of day.

Or you could put it in a swimming pool and filter it if you wanted to drink it. Swimming pools are still a pain to maintain, but at least they're fun, in a way that a psycho apocalypse water barrel complex isn't.

Speaking of the same thing more prosaically, if you have a tank of hundreds of liters of water, it's not going to change temperature much between day and night unless it's really, really shallow, so you can take a cold bath any time, day or night. In a heat wave, this can save your life.

Storing fuel at home

A lot of the food I've described here isn't edible without cooking. Some commonly-eaten legumes are actually dangerous to eat raw, especially kidney beans. But if you can't pay the bill one month and they turn off your gas or electricity, or if there's an earthquake, or if somebody's air force bombs your city's power plants or transmission lines, or there's a heat wave and consequent power outages, you could

be out of luck forever if you don't have fuel stored up.

And energy is important for other things as well as cooking.

Wood and charcoal

The most common response to this situation is to cook with wood, dried poop, or charcoal, which isn't very convenient and is dangerous --- it produces carbon monoxide and, except for charcoal, carcinogenic and irritating smoke. (This is the biggest cause of childhood death today, now that we've dramatically ameliorated the diarrhea situation.) The fuel is also kind of a pain to keep stored, and it can harbor wildlife, including termites, deadly spiders, mice, rats, and cockroaches.

Compressed gas

The next-most-common response is to store LP gas or CNG in a tank, either inside or outside your house. This works fine, is reasonably safe, and is extremely convenient because you can adjust the flame instantly. It does involve a certain amount of up-front investment if you're not renting the tank, and it's not completely safe; people have produced spectacular explosions by accidentally driving cars into LP gas tanks.

Alcohol

A less-common approch is to cook with alcohol. The Penny Stove is a stove burner which can cook a meal with tens of milliliters of alcohol, weighs a few grams, and can be made from two discarded aluminum cans and a worthless coin. Ethanol is a particularly safe fuel: it won't explode if its container is breached, doesn't usually burn the surface it's burning on top of (so it probably won't burn your skin if you get splashed with it), can be extinguished with water, produces very little carbon monoxide when burning, and is much less toxic than alternatives like gasoline or methanol. It's also usable for disinfecting water or wounds and as a solvent for some things, and because it doesn't require high-pressure containment, it's more portable than LP gas or CNG. It's much more expensive than those fuels, though. Ethanol here currently costs about AR$20 per liter, and that liter might cook 50 meals --- AR$0.40 per meal. In the Penny Stove, it's also less controllable than gas fuels, but I think there are camping stoves that provide similar levels of control to a regular gas stove.

If you're poor enough to worry about US$1 per day of food costs, cooking on alcohol may be a low-upfront-investment alternative to buying a gas cylinder or hookup and gas stove. It's a lot more convenient and safer than wood or charcoal. It might even be cheaper than those fuels if you have to buy them, but most people scavenge them instead.

I've previously written on kragen-tol about inherently-safe ethanol storage XXX by floating your ethanol bottles in your water stockpile, so that almost any conceivable storage accident will result in the ethanol mixing with the water and becoming nonflammable.

Solar cooking

Solar cooking is a zero-fuel-consumption alternative, but it's less convenient than cooking with fire unless you have a bunch of heat-storage bits that are not very frugal yet. Still, I think you can

make reasonable solar cookers entirely out of garbage, assuming you have sunlight access.

Specifically, if you have a thermal storage tank for molten salt, you can use that to cook with solar heat whenever you want; the eutectic mixture of nitrates of potassium and sodium melts at 221°, hot enough to bake with, and has a heat of fusion near that of water, so you can store a huge amount of heat in a tiny space. But this is very far from being a practical option to save money.

Refrigeration

Refrigeration also suffers when energy supplies are interrupted. If you have a superinsulated icebox as I suggested earlier, or even if you keep your refrigerator and freezer full of bottles of water when they have available space, you may be able to weather some days without refrigeration power.

In our Vanagon, we had a dual-fuel refrigerator, which could run off the 12-volt van electricity (for hours) or the 12-liter LP gas tank (for days or weeks). It was of the ammonia-absorption type, which is somewhat more hazardous than the usual HCFC-compression type to have inside your house, but has no moving parts other than the gas valve. Gas-fueled refrigerators are common in houses in rural parts of the US.

With a somewhat more elaborate system, I think you could run an ammonia-absorption chiller from a solar concentrator to freeze ice (maybe salted) during the day, and use the ice with a secondary heat transfer circuit to keep your food cold. This is not very practical for individual frugality unless somebody commercializes it.

Space heating

Jesus, you think it's cold in your house without the heat on? Do you have a medical problem with body-temperature regulation? Otherwise, you know, in igloos, people get naked, even though the igloos aren't melting. You can adjust. Insulate your house, chink the cracks, put on a sweater, put a mylar blanket over your regular blanket, snuggle up to somebody, and quitcher complaining.

(Some solar thermal mass might help too.)

Air conditioning

This is a bigger problem; many more people die in heat waves than in cold waves. Their houses get too hot, because they don't have air conditioning or because the power went out, and they die. As the climate changes, we'll see more and more heat waves, and more and more mass deaths as a result. Air conditioners require a lot of energy; you need a hefty generator to run one, and it will gobble up lots of generator fuel. In theory you could run one off solar heat (see earlier comment under Refrigeration) but it's simpler to just have lots of thermal mass around you, thick enough that it'll stay cool past the end of the heat wave. Traditional adobe construction was about a meter thick, and solved that problem.

In theory, you could build a Thermal Mass Stockpile in your house just like an emergency food or water stockpile. In practice, it's considerably more difficult and not really practical. It would be a little alcove of adobe (or brick, or stone, or concrete), just big enough for you to be comfortable inside, with meter-thick walls, ceiling, and floor, and a closeable door. If you made it horizontal --- a

thermal-refuge version of a Kapuseru Hoteru bunk --- you could probably get by with a one-meter by one-meter by two-and-a-half-meter inside volume, enough of a sturdy metal frame to keep it safe from collapse, a mattress, and a light. The outside dimensions would be 3 m × 3 m × 3½ m, for a total volume of 31½ cubic meters, of which 29 cubic meters would be adobe. About 64 tonnes of adobe. This is going to be more than 6.25% of your living space, and your downstairs neighbors are going to raise hell about the cracks in their ceiling.

(You could actually keep your food and water stockpiles inside of it, keeping them at a consistent temperature and slightly cutting down on the adobe requirements all at once.)

Failing that, if you have, or recently had, a working refrigerator, soft-drink bottles filled with frozen water can make a huge difference. Sleep with one, wrapped in a towel, at night; snuggle up to one during the day. If the air temperature gets above about 35°, snuggle up to an ice bottle or two under blankets; instead of keeping your body heat from escaping into the air, the blankets keep the air's heat from escaping into your body and the ice bottles. In a sense, your ice bottles are your "stockpile of air conditioning".

Or you could take a cool bath, if you have or can get cool water.

If you live on the ground instead of up in the air like me, you can just dig a meter-deep trench in the ground, shore it up to keep it from collapsing, put a cover over the top, and lie down in it. This has its potential problems (radon, groundwater, propane or freon) but digging out five or six tons of dirt beats the hell out of molding 64 tons of it into bricks or heaping 64 tons of it on top of your box.

All of these thermal-mass stockpile approaches become slightly more practical when they're for more than one person. If you have four people, you can build a 4×2½×1 meter capsule with a meter of adobe around it, and instead of 64 tons of adobe for one person, you have 5×3½×3 - 4×2½×1 = 42½ cubic meters of adobe, or 93½ tonnes for four --- less than 25 tons each!

Topics

Why John Backus Was on the Wrong Track

Kragen Javier Sitaker, 2007 (updated 2019-05-05) (48 minutes)

(This document is unfinished.)

So I've been reading John Backus's Turing Award paper from 1977 [Backus 1977], and a lot of the assertions he makes therein seem dubious to me. Perhaps this is a little unfair, since von Neumann computers are now 60 years old, and were only 30 years old when Backus wrote; and much of the interesting stuff in the intervening 30 years has been elaboration on Backus's ideas. And I'm just some wanker, while Backus was a Turing Award winner, so probably whatever I can say here is pretty obvious now; but quite a bit of it was anticipated by Dijkstra's EWD-692 response immediately afterwards.

According to Eliezer Yudkowsky, Max Gluckman once said: "A science is any discipline in which the fool of this generation can go beyond the point reached by the genius of the last generation."

So maybe this article is a demonstration that computer programming is a science.

Still, I thought it would be useful to collect 30 years of perspective on the very influential ideas he expresses in this paper.

Summary

Backus's initial complaint about the obesity of von Neumann languages was myopic, coming as it did on the heels of five years of remarkable innovation in the invention of small von Neumann languages. Although he correctly identifies "word-at-a-time thinking" as a serious obstacle to good programming, its cure is garbage collection, not the removal of the assignment statement.

The point-free style he advocates could have been adopted a decade or more before he wrote his paper, but it wasn't, apparently because programs in that style are fucking hard to read. Many languages available today reduce the problems he identified with von Neumann languages of the time, as illustrated by his inner product example, but without using that point-free style; some of those languages were already available when he wrote, but were not in the mainstream.

Most people who have commented on the paper don't seem to have read most of the meat of the paper, probably because it's badly presented. I suggest another order of presentation that I think makes them easier.

Backus's pessimism that formal semantics would make von Neumann languages more tractable was wrong, although this is in part because those languages have been redesigned over the years to be analytically tractable. Similarly, his assertion that the word-at-a-time nature of von Neumann languages makes them non-extensible is both unsupported and simply mistaken, as illustrated by ample counterexamples from his time and our own.

Myopia About Existing Languages

Backus: "Each successive language incorporates, with a little

cleaning up, all the features of its predecessors plus a few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. . . . For twenty years programming languages have been steadily progressing toward their present condition of obesity; as a result, the study and invention of programming languages has lost much of its excitement."

The C programming language was already in use at Bell Labs and a few universities when he wrote this, although Kernighan and Ritchie's manual (120 pages, I think?) didn't come out until, I think, the next year. C is not a minimal language, but as K&R says, it demonstrates that a language that doesn't have everything can be easier to use than those that do. Other smallish languages in use in 1977 — almost all substantially created during the previous five years — included BLISS-10 and BLISS-11, Scheme [Sussman 1975], Prolog, Smalltalk, and, of course, Forth, which is perhaps the ultimate in simplicity of mechanism among high-level languages.

Backus does say, in the next paragraph, "Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular." But I think this dramatically understates the reality of 1977, as seen from 2007; there had been a renaissance in small, elegant languages over the several years immediately preceding Backus's paper, languages whose simplicity, elegance, and power has kept many of them in use until the present day. Some of them were not widely known at the time, but perhaps Backus's disgust owes more to overexposure to PL/1, which had been standardized the previous year after 13 years of drafts, and to Algol 68, than to a paucity of alternatives.

Dijkstra makes a similar comment in EWD-692:

He writes that "smaller, more elegant languages than Pascal continue to be popular", where in the case of Pascal "rapidly gaining in popularity" would be more accurate.

The Intellectual von Neumann Bottleneck Isn't

Backus says, "Not only is the tube [between the processor and memory] a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand."

This is necessarily true of programs for the Altair 8800, which shipped with 256 bytes of memory and could run Altair BASIC in 4096 bytes, or an un-upgraded Apple][, which I think had shipped around the time of Backus's talk. But, as demonstrated by APL, the various FP and FL systems that came after this paper, and the "Lambda: The Ultimate" series of papers around the same time, the limitation is not really imposed by the hardware architecture; it takes some cleverness but not a huge amount of code to allow those single words to represent the larger conceptual units in question.

In a sense, McCarthy's 1960 Lisp article that Backus cites also showed this — or rather, the Lisp implementation that followed it in the next few months. That Lisp (modulo its bugs and its dynamic scoping, which had to be papered over with the FUNARGS (XXX?)

form) was capable of more or less directly implementing all of the primitive functions and functional forms that Backus presents in this paper.

His sample program from section 5.2 is:

```
Def Innerproduct = (Insert +)o(ApplyToAll ×)o Transpose
```

or

```
Def IP = (/+)o(α ×)o Trans.
```

where "o" is the function composition operator, "×" multiplication, and "α" the alpha he uses for ApplyToAll. We can define the operations needed to write this quite concisely in modern Scheme, with lists for vectors, as follows. (Remember that Backus's functions are to be applied to a single argument which is a vector of its arguments.)

```
(define (plus args) (apply + args))
(define (x args) (apply * args))
(define (o f g) (lambda (x) (f (g x))))
; Note that this implementation doesn't discover whether its
; argument has a unit so that it can do the right thing in the
; case of an empty vector.
(define (insert fn)
  (lambda (args) (if (null? (cdr args)) (car args)
                     (fn `(,(car args)
                           ,((insert fn) (cdr args)))))))
(define (alpha fn)
  (lambda (args) (if (null? args) '()
                     `(,(fn (car args)) .
                       ,((alpha fn) (cdr args))))))
(define (transpose1 heads tails)
  (if (null? tails) '()
      (cons (cons (car heads) (car tails))
            (transpose1 (cdr heads) (cdr tails)))))
(define (transpose0 lst)
  (if (null? lst) '() (cons '() (transpose0 (cdr lst)))))
(define (transpose lst)
  (transpose1 (car lst) (if (null? (cdr lst)) (transpose0 (car lst))
                            (transpose (cdr lst)))))
(define ip (o (o (insert plus) (alpha x)) transpose))
(begin (display (ip '((1 2 3) (6 5 4)))) (newline))  ; outputs "28"
```

You will notice that the definition of "ip" here is identical to the one from Backus's paper, except for being in prefix order with parentheses. This code makes heavy use of closures, but with some cost in readability could use the Lisp 1.5 FUNARGS (XXX?) form instead, and similarly quasiquotation could be replaced with explicit calls to "cons". Maybe it would be 30 lines of code instead of 22. (I don't have a Lisp 1.5 implementation handy to test on, so I can't say for sure.)

So, however tied we may have been to "word-at-a-time thinking" in the first 30 years of von Neumann computers, there had been systems in use since 1964 or so that could express Backus's functional forms in a few lines of code each.

And APL had already shown some of the way from the mid-1960s on. While APL expressions' values were internally represented as pointers, they were presented to the user in a very non-word-at-a-time fashion; and indeed, in many cases, they represented finite maps with domains of integers less than some limit, or integer pairs, and so on.

And, of course, Church's SK-combinators (which Backus mentions in his paper) are straightforward to implement even in ancient Lisp, sufficient to express any computable function, support the writing of higher-order functions, and don't name their arguments. (But I think the first efficient implementations of SK-combinators on computers, using graph reduction instead of tree reduction, either did not precede Backus's paper by much, or actually followed it.) XXX find out, asshole

So why had programming with higher-order functions that don't name their arguments — what we now call "point-free style" — not taken off in the 1960s? Was it really because programmers were too tied to "word-at-a-time thinking"? That doesn't seem plausible to me.

The alternative I suggest — after 40 years of our collective experience with "point-free style" in Forth, as well as 30 years in languages inspired by this very paper, languages such as Miranda, Haskell, and ML, plus more than 50 years of APL one-liners that are nearly point-free — is that while point-free programs are indeed much easier to "refactor" (i.e. perform semantics-preserving transformations on), they are fucking hard to read, maybe because they're too abstract.

This is based not only on my own limited experience, but the experiences of other hackers who have spent years programming in point-free style. Even the HaskellWiki Pointfree page [HaskellWiki Pointfree] admits that it can sometimes be hard to read:

> Point-free style can (clearly) lead to Obfuscation when used unwisely. As higher-order functions are chained together, it can become harder to mentally infer the types of expressions. The mental cues to an expression's type (explicit function arguments, and the number of arguments) go missing.
>
> Point-free style often times leads to code which is difficult to modify. A function written in a pointfree style may have to be radically changed to make minor changes in functionality. This is because the function becomes more complicated than a composition of lambdas and other functions, and compositions must be changed to application for a pointful function.
>
> Perhaps these are why pointfree style is sometimes (often?) referred to as pointless style.

(However, other parts of the same page advocate point-free style as "clearer", "cleaner", and "good discipline".)

Today's Programs for Inner Product

So today, the state of the art (due largely to work in languages inspired by Backus's paper) is something like this Python program:

```
def innerproduct(a, b): return sum(ai * bi for ai, bi in zip(a, b))
```

Which compares favorably to Backus's 1977 "von Neumann program for Inner Product":

```
c := 0
for i := 1 step 1 until n do
  c := c + a[i] × b[i]
```

(This could be translated to C as follows, gaining a bit of clarity but not changing in any substantial way:)

```
c = 0;
for (i = 0; i < n; i++) c += a[i] * b[i];
```

Backus lists seven undesirable properties of this program:

a) Its statements operate on an invisible "state" according to complex rules.

This is not really true of the Python program above. Although it is a user-visible feature that the expression `(ai*bi for ai, bi in zip(a, b))` results in a stateful iterator that eventually runs out of values, this program does not make use of that property.

The rules that govern the evaluation of the program are more complex than those of Backus's FP, though.

b) It is not hierarchical. Except for the right side of the assignment statement, it does not construct complex entities from simpler ones.

Actually, the assignment statement contains four nested compound expressions, made out of five atomic variables and one another. The Python program contains `zip(a, b)`, `ai * bi`, the generator expression, and the `sum` expression — exactly the same number; but perhaps `zip(a, b)` and the generator expression would be more to Backus's liking, since they express exactly the return values of the Trans and (α ×) operations in his functional IP program.

c) It is dynamic and repetitive. One must mentally execute it to understand it.

This is not true of this Python program, although it is true of many Python programs. Incidentally, this is exactly Guido van Rossum's reason for wanting to remove Backus's "insert", called `reduce`, from Python [van Rossum XXX] — he finds he has to mentally execute the reduction process to understand it.

d) It computes word-at-a-time by repetition (of the assignment) and by modification (of variable i).

This is true of the actual execution of the Python program, as it is of the execution of Backus's FP programs or any other programs on a von Neumann machine; but if you try to interpret it as a statement about the user-level semantics of the code, it reduces to criticism (c).

e) Part of the data, `n`, is in the program; thus it lacks generality and works only for vectors of length `n`.

Not true of the Python program.

f) It names its arguments; it can only be used for vectors `a` and `b`. To become general, it requires a procedure declaration. These involve complex issues (e.g. call-by-name versus call-by-value.)

Almost as true of the Python program as of the C version; in fact, the procedure declaration constitutes fully half of my program as written.

"Call-by-name" here refers to evaluating procedure parameters when their values are needed, rather than before entering the procedure; it's like lazy evaluation, but may evaluate the procedure parameter more than once. In effect, this allows you to construct any arbitrary zero-argument closure, called a "thunk", subject to the limitations of what you can express in an expression in the language in question.

In the cases where the parameter will always evaluate to the same value, it is therefore just a gratuitously inefficient version of lazy evaluation; in the cases where it can evaluate to different values, its value therefore depends on some information that could be thought of as a parameter to the thunk, but which generally needs to be passed in by mutating some apparently-unrelated variable, and so in those cases, it is just a gratuitously bug-prone version of a general closure facility (limited to passing the closures downward through the stack).

"Call-by-name" was therefore replaced by a combination of call-by-reference and a general closure facility, albeit one limited to "downward funargs", in Pascal (XXX did Algol-60 have downward funargs? I think so); and C and most languages designed since then have simply stuck to call-by-value. So, although call-by-name does indeed involve complex issues, it should have been an irrelevancy long before 1977, and merely adding procedure declarations to your language does not imply that your semantics will suffer the slings and arrows of outrageous call-by-name.

XXX sometimes people say "call-by-name" when they mean "lazy evaluation"; maybe that's what Backus meant?

(Of the other languages I've used or mentioned in this note, C, Scheme, Python, Smalltalk-80, APL, OCaml, 1960 Lisp, Miranda, Haskell, ML, and Forth just use call-by-value; Perl 5 just uses call-by-reference; C++ has both call-by-value and call-by-reference; Algol 60 has both call-by-value and call-by-name; Prolog uses something else entirely; Altair BASIC didn't have procedures; I'm not sure about BLISS-10 and BLISS-11; Lisp 1.5 had call-by-value and, I think, also FEXPRs; I think Smalltalk-72 did something vaguely FEXPR-like; and PL/1 and (I think) Algol 68, consistent with the rest of the language design, have every horrifying deformed argument-passing convention you can imagine.)

However, the distinction between call-by-value and call-by-name includes another complexity — strictness. A function "f" is strict in some parameter if it can't compute its return value without getting the value of that parameter. Backus calls this property "bottom-preserving", and not having procedure declarations has not saved him from strictness; throughout the paper there are a number of minor logical errors having to do with this particular extra complexity.

Backus implies that procedure declarations import some other complexities that he hasn't mentioned; the ones I can think of are variable scoping (lexical vs. dynamic vs. broken) and the creation of closures (downward-only or generalized; interchangeability with non-closure function pointers; etc.).

This absence of variables is the central difference, as I read it, between Backus's proposed programming system and most of the purely applicative systems that preceded it: unlike them, it doesn't have lambda substitution. It is interesting to note that nearly all of the systems inspired by Backus's paper over the ensuing 30 years have acquired lambda substitution rather quickly.

g) Its "housekeeping" operations are represented by symbols in scattered places (in the for statement and the subscripts in the assignment). This makes it impossible to consolidate housekeeping operations, the most common of all, into single, powerful, widely used operators.

This is still somewhat true of the generator expression (`... for ... in ...`) but not the `sum` and `zip` operations. Backus never defines "housekeeping" anywhere in his paper, but as additional examples, he gives function composition, matrix transposition, apply-to-all, and "insert", more commonly known as "reduce" or "fold".

A modern C++ program, using the STL, is somewhat similar in structure, although dramatically wordier. C++ is probably the poster child for obese von Neumann languages; here I have included the entire file with a `main()` function, because that turned out to be a little tricky for me to write concisely with my limited knowledge of C++, and I would hate for anyone else to have to suffer the same way. It's only the 4 lines inside `innerproduct` that correspond to the 2-3 lines of C given at the top of this section or the half-line of Python.

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

template<typename T>
T innerproduct(vector<T> a, vector<T> b) {
  vector<T> result;
  transform(a.begin(), a.end(), b.begin(),
            back_insert_iterator<vector<T> >(result), multiplies<T>());
  return accumulate(result.begin() + 1, result.end(), result[0], plus<T>());
}

#define arrayend(array) ((array)+sizeof(array)/sizeof((array)[0]))
int main(int argc, char **argv) {
  int aa[] = {1, 2, 3};
  int bb[] = {6, 5, 4};
  vector<int> a(aa, arrayend(aa)), b(bb, arrayend(bb));
  cout << innerproduct(a, b) << endl;
  return 0;
}
```

(I think that's purely standard C++. I tested it with g++ 4.1.2. There's no particular reason `transform()` couldn't have been defined to produce an `InputIterator` instead of consuming an `OutputIterator`, but it wasn't. Oh, and the STL contains an `inner_product()` template function that does this already, but you have to give it a starting value, as with `accumulate()`, so it's hard to make a call to it generic across element types.)

(A note about performance: the above program compiles to 3632 bytes of machine code, on the order of 750 instructions, containing obvious inefficiencies; the inner-product function itself, instantiated for vectors of integers, is 86 instructions, and on my laptop, it takes 3300 nanoseconds on Backus's example vectors. By comparison, the analogous non-generic C function, with the code at the top of this section, is 23 instructions, contains no calls to other functions, and consequently executes in 54 nanoseconds. So much for the STL's vaunted "absolute efficiency". The C++ version also requires four dynamic libraries at run-time.)

In "A Short History of STL" [Stepanov 2007], Stepanov writes about how his work was inspired by Backus's FP work.

In OCaml, which is very much inspired by Backus's paper, we could write:

```
List.fold_left2 (fun a b c -> a + b * c) 0
```

Although that's not generic across numeric types the way the Python and C++ versions are.

In Squeak Smalltalk, it looks like this, as a method on `SequenceableCollection`:

```
innerProduct: aCollection
    ^ (self with: aCollection collect: [:a :b | a * b ]) sum.
```

I don't know if `with:collect:` and `sum` existed in Smalltalk-80, let alone the Smalltalk that existed in 1977; the oldest version in Squeak was in 1999 by "di", which is presumably Dan Ingalls, but older versions may exist.

In R5RS Scheme, if I don't restrict myself to trying to reproduce the structure of Backus's program exactly using facilities that I'm sure were present in early Lisp, I can write it quite concisely as well:

```
(define (innerproduct a b) (apply + (map * a b)))
```

I don't know when Scheme's map acquired the ability to apply to an arbitrary number of arguments, but I suspect it was a bit later than Backus's paper.

The Smalltalk and Scheme versions, too, are generic across all numeric types; although the Scheme version, unlike the Python, C++, and Smalltalk versions, doesn't support user-defined numeric types.

These programs are improvements over the C version along Backus's criteria in the following ways: Python: abceg; C++: beg; OCaml: abceg; Squeak: abceg; Scheme: aceg.

So, in sum, von Neumann programming languages — the avant-garde, if not yet the mainstream — have adopted features that provide most of Backus's desiderata, but without requiring the point-free style he considers crucial. Some of them, I think, had those features before his paper.

Garbage Collection is the Key to Not Thinking Word-at-a-Time

In section 4, Backus writes, "The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does. Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result."

But actually, even years before Backus wrote his paper, PL/1 supported assignment statements that copied entire arrays, and C

eventually acquired the ability to copy `structs` in simple assignment statements as well. But you can't write the functional forms used in Backus's "InnerProduct" program in those languages, the way you can in Python, OCaml, Smalltalk, or Scheme. The key is that, even though assignment statements in Python or Scheme only copy single words (unlike those in C and PL/1!) those single words can effectively stand for much larger data structures that they point to.

In C, a single word can point to a much larger data structure, but you must constantly be aware of the difference between the pointer and the thing it points to, because you must be careful to free the thing it points to before overwriting the last copy of the pointer, or letting it go out of scope. C++ can do better here because its destructors and copy-constructors allow you to automate that work.

So, even though C's assignment statements and parameter-passing can copy entire `structs` (say, complex numbers, or begin-end pointer pairs), they can't copy structures of irregular and unpredictable size, such as the transposed list of pairs in the inner-product function. Garbage-collected languages can universally pass structures of irregular and unpredictable size without worrying about who deallocates them, and so you can toss around a list, or a parse tree, or an arbitrary-precision number, or an arbitrary-size matrix, or a drawing, as easily if it were a single byte.

(As long as you don't mutate it. Once you start mutating it, you have to worry about aliasing. But most of the time, that doesn't impose a large practical complexity penalty on your programs — just your proofs.)

In sum, Backus misidentified the source of word-at-a-time thinking; it is not the assignment statement and its ability to transfer only a word at a time, but rather the hassle of manual memory management.

This is perhaps somewhat surprising; garbage collection had been invented for Lisp in 1958 or 1959, and Algol-68 implementations were required to provide it. But it seems that in 1977, Backus didn't have much experience programming in garbage-collection languages. Perhaps this is because, despite Algol-68's mandate, garbage-collected programming environments weren't in wide use by 1977, largely because garbage collection was grossly inefficient until the invention of generational garbage collection (in 1982? XXX).

However, this myopia about garbage collection was not limited to Backus; Dijkstra comments in EWD692, about this very paper:

> In the first step [of the program MM] each of the component functions in the construction ("1" and "trans o 2" , respectively) is combined with the total operand —a sequence of two matrices— from which in the last step (Selection) each extracts the half it really needs. If the matrices m and n are sizeable [sic], a naive [sic] implementation that first copies those matrices and then kicks out half of it again seems absolutely unacceptable on *any* machine —von Neumann or not—.

But That's Not the Point!

But the "abcdefg" desiderata in section 5 aren't the real point of the paper! The real point of the paper, found in section 9, is that languages that are more mathematically tractable permit "algebraic laws" to be "used in a rather mechanical way to transform a problem into its solution." Backus proposes to solve this problem by inventing a new programming style that is particularly easy to manipulate

mathematically.

There is definitely merit in using programming styles that make it easy to prove things and to mechanically transform programs without changing their semantics; this was the idea of Dijkstra's "structured programming", which Backus mentions in passing.

Unfortunately, while I think Backus was correct that purely-applicative programs in general, and point-free purely-applicative programs in particular, are particularly amenable to formal manipulation, he doesn't do a very good job of presenting this in the paper.

Backus's Pessimism About Formal Semantics Was Wrong

In section 9, Backus writes:

> Axiomatic semantics [Hoare 1969] precisely restates the inelegant properties of von Neumann programs (i.e. transformations on states) as transformations on predicates. ... [Its practitioners'] success rests on two factors in addition to their ingenuity: First, the game is restricted to small, weak subsets of full von Neumann languages that have states vastly simpler than real ones. Second, the new playing field (predicates and their transformations) is richer, more orderly and effective than the old (states and their transformations). But restricting the game and transferring it to a more effective domain does not enable it to handle real programs (with the necessary complexities of procedure calls and aliasing) ... As axiomatic semantics is extended to cover more of a typical von Neumann language, it begins to lose its effectiveness with the increasing complexity that is required.

Although Scheme contains a large and useful applicative subset, it is certainly a von Neumann language in the sense that Backus is describing. R5RS includes a formal denotational semantics for Scheme. It's only two and a half pages, and handles the primitive forms of the whole language; another two pages are concerned with formal definitions of rewrite rules that reduce the other special forms in the system to those primitive forms. Real programs in Scheme have had useful properties proved of them, or so I hear. I haven't written or read any of those proofs myself. XXX PreScheme

(Axiomatic semantics is a different approach from the denotational semantics used by R5RS, as Backus points out in section 12, but his skepticism is not confined to one or the other.)

More recently [Leroy 2006], Xavier Leroy's team at INRIA has constructed a compiler in OCaml from a large subset of C, which they call Clight, to PowerPC assembly. The compiler is automatically extracted from a machine-checked proof of its correctness written in Coq. While the source language Clight is, at this point, still a toy language (it lacks `structs`, `unions`, `typedef`, `goto`, `switch`, and, I believe, casts) its compiler is definitely a "real program".

On the other hand, while the problems Backus identifies are not as severe as he thought, neither are they nonexistent. Apparently [Leroy 2006] it is still the case that nobody has published a formal semantics for C, which was the most-widely-used von-Neumann-style language during much of the 30 years since Backus's paper (say, 1984 to 1994, or possibly even until today). So the early hopes accompanying formal denotational semantics work were overoptimistic, essentially for the reasons Backus identifies.

And Backus's other skepticism expressed later has turned out to be correct: "If the average programmer is to prove his programs correct,

he will need much simpler techniques than those the professionals have so far put forward." And point-free purely-applicative programs have indeed turned out to be among the easiest to manipulate in this way.

Dijkstra writes in EWD-692:

...his objection is less against von Neumann programs than against his own clumsy way of trying to understand them.

(But over the next few dozen EWD notices, Dijkstra seems to have changed his mind about the ease of proving properties of functional programs.)

Von Neumann Languages Can Be Powerfully Extensible

Backus writes:

Let us distinguish two parts of a programming language. First, its *framework* which gives the overall rules of the system, and second, its *changeable parts*, whose existence is anticipated by the framework but whose particular behavior is not specified by it. For example, the for statement, and almost all other statements, are part of Algol's framework but library functions and user-defined procedures are changeable parts...

Now suppose a language had a small framework which could easily accommodate a great variety of changeable features entirely as changeable parts. Then such a framework could support many different features and styles without being changed itself. In contrast to this pleasant possibility, von Neumann languages always seem to have an immense framework and very limited changeable parts. What causes this to happen? The answer concerns two problems of von Neumann languages.

The first problem...a von Neumann language must have a semantics closely coupled to the state, in which every detail of a computation changes the state. The consequence of this semantics closely coupled to states is that every detail of every feature must be built into the state and its transition rules.

Thus every feature of a von Neumann language must be spelled out in stupefying detail in its framework. ...many complex features are needed... The result is the inevitable rigid and enormous framework of a von Neumann language. . . .

The second problem of von Neumann languages is that their changeable parts have so little expressive power. Their gargantuan size is eloquent proof of this; after all, if the designer knew that all those complicated features, which he now builds into the framework, could be added later on as changeable parts, he would not be so eager to build them into the framework.

His "first problem" is a non sequitur. Why would having every detail of a computation change the state, and therefore semantics being closely coupled to states, result in not being able to define "features" such as the `for` statement? Backus never explains what "feature" means, nor does he explain any kind of connection between operational semantics and lack of extensibility.

The only additional complication operational semantics bring to extensibility is that when you define a new expression or statement type — such as a `for` statement — you must specify if, when, and how many times each subexpression is executed, while purely-applicative languages need only specify what to do with their results.

In fact, there already existed many highly-extensible von Neumann languages when he wrote this, Lisp and Forth being among the prime examples of the type. Smalltalk-72 was more extensible than Smalltalk-80, but even Smalltalk-80 supports easy user-level definitions of things such as the Algol `for` statement. Here's the definition of Number>>#to:do: from Squeak, which is a

Smalltalk-80 implementation:

```
to: stop do: aBlock
        "Normally compiled in-line, and therefore not overridable.
        Evaluate aBlock for each element of the interval
        (self to: stop by: 1)."
        | nextValue |
        nextValue _ self.
        [nextValue <= stop]
                whileTrue:
                        [aBlock value: nextValue.
                        nextValue _ nextValue + 1]
```

This allows you to write a for loop like this:

```
1 to: 10 do: [:i | Transcript show: ('item ', i asString); cr]
```

As the comment explains, the compiler specially recognizes the #to:do: selector and inlines an implementation equivalent to this one; but you can use the code to define another loop structure that does the same thing under another name.

There are several different approaches to defining highly extensible von Neumann languages:

• Lisp macros: by running arbitrary user-defined code at compile-time, you can add arbitrarily complex language features. R5RS defines Scheme in terms of only six primitive expression types: variables, `quote`-expressions, procedure calls, `lambda` expressions, `if`, and assignments. Everything else — `cond`, `case`, `let`, `let*`, `letrec`, `begin` (sequencing), `do` (the `for` loop), even access to the macro system — is defined in terms of the macro system. While Lisp macros had their shortcomings at the time Backus wrote (hygienic macros wouldn't be successfully implemented for, I think, another ten years) they were already in heavy use in, at least, MacLisp. R5RS defines `do` in the following fairly tricky way:

```
(define-syntax do
  (syntax-rules ()
    ((do ((var init step ...) ...)
         (test expr ...)
         command ...)
     (letrec ((loop
               (lambda (var ...)
                 (if test
                     (begin (if #f #f) expr ...)
                     (begin command ...
                            (loop (do "step" var step ...) ...))))))
       (loop init ...)))
    ((do "step" x) x)
    ((do "step" x y) y)))
```

(Some Scheme dialects also have a Common-Lisp-style macro

system in which the code to rewrite the tree is normal Scheme rather than this `syntax-rules` crap, but it's somewhat trickier to use because of name-capture problems.)

C++ uses a similar approach; like standard Scheme, it has, in effect, a completely different programming language that runs at compile-time, based on pattern-matching of C++ types.

Forth uses this approach to the extreme. The Forth equivalent of the `for` loop is the `DO LOOP` loop, which looks like `10 0 DO I . LOOP` — 10 is the loop limit, 0 is the starting value, `I .` is the body of the loop (which prints the value of the loop counter), and `LOOP` ends the loop. There are variants, but that's the basic structure. Here's the implementation of `DO LOOP` in F-83, a 1983 implementation of Forth; this version is for MS-DOS, and so some structures are written in an RPN version of 8086 assembly for speed. (I gathered this together from several different parts of `KERNEL86.BLK`.)

```
ASSEMBLER HEX
CODE (DO)   (S l i -- )   AX POP   BX POP
LABEL PDO   RP DEC   RP DEC   0 [IP] DX MOV   DX 0 [RP] MOV
  IP INC   IP INC   8000 # BX ADD   RP DEC   RP DEC
  BX 0 [RP] MOV   BX AX SUB   RP DEC   RP DEC   AX 0 [RP] MOV
  NEXT END-CODE
CODE BRANCH   (S -- )
LABEL BRAN1   0 [IP] IP MOV   NEXT END-CODE
CODE (LOOP)   (S -- )   1 # AX MOV
LABEL PLOOP   AX 0 [RP] ADD   BRAN1 JNO
  6 # RP ADD   IP INC   IP INC   NEXT END-CODE
FORTH
: DO      COMPILE (DO)   ?>MARK                    ; IMMEDIATE
: LOOP
    COMPILE (LOOP)  2DUP 2+ ?<RESOLVE ?>RESOLVE    ; IMMEDIATE
```

Essentially everything is the "changeable parts" in Forth. The eForth 1.0 model from 1990, in an effort to define a maximally portable Forth, has 171 instructions of actual machine code constituting a two-instruction "interpreter" and 31 primitive "words" or procedures.

I want to emphasize that, although some of these words are written in assembly, and some run at compile-time, in no way do they form an unchangeable part of the framework in the sense that Backus deplores; any user of the system can define new words in assembly at any time, or define new words that run at compile-time to define new control structures, and those words are immediately available.

Although the above exercise in archaeology is from six years after 1977, the relevant attributes of Forth were in place from some time before Backus wrote.

• Lightweight closures: Smalltalk-80 gets most of its extensibility from a very lightweight syntax for (restricted) closures, as shown in the above code for Number>>#to:do:. This has undesirable effects occasionally — in `((index <= limit) & (anArray at: index))`, the second half of the conjunction is evaluated regardless of the value of the first half. There's another method that allows you to write `((index <= limit) and:`

`[anArray at: index]`) and only evaluate the second part if the first one is true; but the notation is undesirably asymmetrical and non-infix.

Perl 5's prototypes and Ruby's block arguments give them a similarly lightweight closure syntax in some circumstances, which can provide similar facilities.

• Reflection: if there's very little done at compile-time that your code can't change at run-time, such as constructing new classes, enumerating the methods of classes, constructing method names and calling the named methods, evaluating strings of source code, and so on, you can get a certain amount of extensibility without any special features of the actual language. Smalltalk's #doesNotUnderstand: method, for example, allows classes to reuse the method namespace for their own purposes.

In sum, Backus's assertion that the changeable parts of von Neumann languages are necessarily quite limited in their expressiveness is poorly reasoned and simply wrong, and ample counterexamples existed even before he wrote it, although he may not have been aware of them.

The Design of His FP System

Backus's FP system is a bit different from modern "functional programming" systems, although it inspired the explosion in them over the next few years. They are generally based on the lambda-calculus, and so it is easy to define new higher-order functions in them, and they incorporate parameter substitution as a fundamental mechanism.

Space Usage

Backus writes, "[This MM program] is an inherently inefficient program for von Neumann computers (with regard to the use of space), but efficient ones can be derived from it and realizations of FP systems can be imagined that could execute MM without the prodigal use of space it implies."

Here's the MM program:

```
Def MM = (α α(/+)∘(α ×)∘ trans))∘(α distl)∘ distr ∘[1,trans ∘ 2]
```

The most obvious interpretation of his remark follows, and under that interpretation, the remark is wrong.

What he means is that the result of the next-to-last step of MM is a four-dimensional array with dimensions (A, B, C, 2), where the the original matrices have dimensions (A, C) and (C, B) and the result has dimensions (A, B). So, for instance, taking the product of two 1000×1000 matrices, producing a million-number result, would require a two-billion-number intermediate result, which is then reduced to a million numbers in a billion multiplications.

Dijkstra made a similar assumption; see the quote from him above in the section about garbage collection.

But, as he says, there's no reason to physically realize all the values of that intermediate result. "An APL Machine" [Abrams 1970], written several years before Backus began his work, describes an architecture for evaluating APL programs that avoids materializing many such intermediate results. In this case, it's sufficient to copy the

sequences by reference, even if you materialize them all.

I was thinking that linear logic was going to be helpful here, since obviously there's no point in materializing a sequence of values that gets used only once when you might as well produce it on the fly (modulo locality of reference and instruction-level parallelism concerns), but actually the values of concern here are the rows and columns, which are indeed used multiple times; distl and distr are not "linear" in Girard's sense.

This straightforward Python translation of the above MM program can multiply a 10000×40 matrix by a 40×10000 matrix, producing a 40×40 result, in 5MB. If it created an intermediate 40 × 40 × 2 × 10000 matrix with no sharing, as Backus seems to have envisioned, that intermediate matrix would contain 32 million values. It takes about five minutes on my computer.

```
#!/usr/bin/python
# A Python implementation of some of John Backus's "FP" system.
def selector(n):
    return lambda x: x[n-1]
def insert(f):
    def rv(x):
        rv = x[-1]
        for xi in x[-2::-1]: rv = f([xi, rv])
        return rv
    return rv
def apply_to_all(f):
    return lambda x: [f(xi) for xi in x]
def compose(*fs):
    if len(fs) == 1: return fs[0]
    f = fs[0]
    g = compose(*fs[1:])
    return lambda x: f(g(x))
def construct(*fs):
    return lambda x: [fi(x) for fi in fs]
def transpose(x):
    return [[x[i][j] for i in range(len(x))]
            for j in range(len(x[0]))]
plus =  lambda (x, y): x + y
times = lambda (x, y): x * y
distl = lambda (y, z): [(y, zi) for zi in z]
distr = lambda (y, z): [(yi, z) for yi in y]
innerproduct = compose(insert(plus), apply_to_all(times), transpose)
print innerproduct([[1, 2, 3], [6, 5, 4]])
mm = compose(apply_to_all(apply_to_all(innerproduct)),
             apply_to_all(distl),
             distr,
             construct(selector(1), compose(transpose, selector(2))))
matrices = [[[1, 0, 1], [0, 1, 0]], [[1, 2], [3, 4], [0, 0]]]
print mm(matrices)
bigmatrices = [[range(10000)] * 40, [range(40)] * 10000]
print mm(bigmatrices)
```

It could be that Backus was merely considering the necessity to allocate *any* variable-sized space for intermediate results as an

"inefficient" use of space, since all you really need for a matrix multiply is three loop counters and an accumulator.

Presenting Some of Backus's Results Better

Backus's section 12.5.1 is a correctness proof for a recursive factorial function, defined as follows:

```
f = eq0 -> constant(1); times o [id, f o s]
s = - o [id, constant(1)]
eq0 = eq o [id, constant(0)]          [from section 11.3]
```

XXX so the idea here is to work through the proof from section 12.5 to get f = / times o [id, id o s, id o s o s, ... constant(1) o crap] and then generalize it. But I'm too sleepy right now. And I probably want a "recursion lemma for f" and a "linearly expansive lemma for f". `p = eq0;  g = constant(1);  h = times;  i = id;  j = s`

Misc Crap

From HaskellWiki: "To find out more about this style, search for Squiggol and the Bird-Meertens Formalism, a style of functional programming by calculation that was developed by Richard Bird, Lambert Meertens, and others at Oxford University. Jeremy Gibbons has also written a number of papers about the topic, which are cited below."

Other Stuff to Read

TODO reformat this section for inline links

[Backus 1977] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. 1977 ACM Turing Award lecture, Communications of the ACM volume 21, number 8, with this funny number: 0001-0782/78/0800-0613.

http://www.stanford.edu/class/cs242/readings/backus.pdf

[Hoare 1969] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM volume 12, number 10 (October 1969), pages 576-583

http://ls14-www.cs.uni-dortmund.de/ls14Medien/Dokumente/Lehre/PaperdesMoonats/hoare.pdf

[Knuth 1973] Donald Ervin Knuth. Structured Programming with Go To Statements. Computing Surveys, volume 6, number 4 (December 1974), pages 261-301.

http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf

[Leroy 2006] Xavier Leroy. Formal Certification of a Compiler Back-end: or: Programming a Compiler with a Proof Assistant. POPL '06, with this funny number: 1-59593-027-2/06/0001.

http://pauillac.inria.fr/~xleroy/publi/compiler-certif.pdf

[Sussman 1975] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349 (AIM-349), December 1975. This was the

initial definition of Scheme, preceding AIM-452, the Revised Report on Scheme.

ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-349.pdf

[van Rossum XXX] that thing he wrote about removing reduce and lambda from Python 3000

[Stepanov 2007] "Short History of STL", by Alexander Stepanov (contributed to Evolving a language in and for the real world: C++ 1991-2007 by Bjarne Stroustrup)

http://www.stepanovpapers.com/history%20of%20STL.pdf

[Abrams 1970] "An APL machine", Philip S. Abrams, SLAC technical report SLAC-114, February 1970, the paper defining "D-machine", "E-machine", "drag-along", and "beating".

[HaskellWiki Pointfree] The "Pointfree" page on HaskellWiki as of 2008-01-08.

http://www.haskell.org/haskellwiki/Pointfree

Topics

Golang has a *lot* built in, compared to C; for example, it has garbage collection, an interesting new way to do ad-hoc polymorphism, parametrically polymorphic finite maps, a range type, lightweight and relatively safe concurrency constructs, complex numbers, type-safe variadic parameters, run-time type identification, exception handling with FIFO automatic cleanup, a string type with equality and concatenation, and a print statement. However, without importing at least some packages, there's no way for a Go program to obtain input, and there are a lot of facilities that are easily accessible in the standard library that can save you a lot of time.

I'm just starting to learn Golang, so I'm taking some notes on the standard library from http://localhost:6060/pkg/, which is unfortunately organized alphabetically, which means that really recondite stuff is mixed in with really basic stuff. So this is my attempt to summarize what I think is most important.

The standard Golang library seems to be entirely lacking facilities for interactive terminal I/O and GUIs.

Really basic packages

Kragen Javier Sitaker, 2019-02-08 (20 minutes)

These are all you need for a pretty wide range of stuff: `testing`, `io`, `fmt`, `os`, `strconv`, `bytes`, `strings`, `math`, and `sort`. Without any one of these, you'd be pretty handicapped.

testing

`go test` runs automated test suites written using functions named `TestFoo`, that take pointers to `testing.T` objects, in `foo_test.go` files. It also has benchmarking and doctest-like functionality. I have to admit I haven't tried this yet because so far I haven't written any Golang packages, just standalone programs, and I'm not sure how to use it with those.

The `test/quick` subpackage does generative property-based testing, like Hypothesis.

fmt

`fmt` does formatted I/O, including `Printf`. It can use reflection to dump out structs (`%v`, applicable to any type, or `%+v` to see field names) and even Golang-syntax representations (`%#v`). Naturally there's a `fmt.Formatter` interface you can implement to override the default formatting.

Because of reflection, `fmt.Fscan`, `fmt.Scan`, `fmt.Sscan`, etc., don't need a format string at all — you just give them some interface values pointing at where you want to store the results — and the same is true of `fmt.Print`. Scanning can be overridden with a custom `Scan` method.

There's also a `print` function you can use without importing `fmt`.

io

`io` is where you find the Reader and Writer protocols, among others; `fmt.Fprintf` takes an `io.Writer` rather than a file. It also contains things like `io.Copy`, which normally copies a `Reader` to a `Writer`, but when possible uses more efficient methods, which I assume means `sendfile(2)`; plumbing utilities like `io.LimitReader`, `io.LimitWriter`, `io.Pipe`,

`io.MultiReader` (cat), `io.MultiWriter` (tee), and `io.TeeReader` (tee); `io.ReaderAt`, suitable for concurrent random-access record I/O from multiple goroutines; etc.

An interesting difference from Unix (or Python) is that `Read()` is supposed to return the error `io.EOF` at EOF rather than just an empty byte count. This has the annoying result that if you `import "os"` and do file I/O with it, you probably also need to `import "io"`.

The subpackage `io/ioutil` contains `ReadFile`, `WriteFile`, and tempfiles.

os

`os` is where you find `Args` and most of the Unix API, including things like `Open(filename)`, `Getpid()`, and `os.Stdout`, which is an unbuffered `io.Writer`, which you can buffer using `bufio`. Opening a file for writing requires calling either `os.Create` or `os.OpenFile`.

There's a separate `File.WriteString` method for when you have a `string` rather than a `[]byte` to write to a file. This is sort of strange because you can convert from `string` to `[]byte` with `[]byte(s)`, which makes me think this method may be left over from an earlier version of Golang where you couldn't do that.

The design is a mix of very Unixy and somewhat portable. File permissions are a 32-bit int with no place to put, for example, a separate "delete" permission bit, as on VMS. But the inode returned by `os.Stat` (`FileInfo`) isn't even a concrete type at all, but an interface. (And it includes the file's basename, but not e.g. ctime, so it's only sort of an inode.)

The `os.Process` API is an interesting design, clearly designed with non-Unix systems in mind. (There's no `os.Fork`!) It's somewhat weak; there's no nonblocking `waitpid`, for example, and thus no way to get an `os.ProcessState` for a running process! `os.exec` has a more convenient interface, but it's apparently built on `os.Process` and therefore can't do anything `os.Process` can't.

There also doesn't seem to be a way to call select(2), although of course you can spawn off a goroutine or two per fire descriptor.

strconv

`strconv` is where you look for conversion to and from strings, although `fmt.Sprintf` gives you a more capable way to build strings.

bytes

Things you would expect to be utility methods on strings (`ToUpper`, `Join`, `Split`, `TrimRight`, `Compare`, `HasPrefix`) are here, as are `io.Reader` and `io.Writer` interfaces for in-memory "files". There's a corresponding `strings` module for strings.

There are actually two different `io.Reader` implementations: the read-write `bytes.Buffer` and the seekable `bytes.Reader`. `bytes.Buffer`, in addition to converting output-generation functions into string-building functions, also allows you to buffer your output in a more controllable way than the `bufio` module mentioned below — it guarantees to return no errors when you write to it (it will `panic` instead if it runs out of memory) and allows you to accumulate the bytes written until you're ready to do something more interesting with them, like send them over a socket.

strings

This is almost the same as the `bytes` package, but for strings, even to the point of including the unnecessary `Compare` function. Naturally, though, it omits the read-write `Buffer` interface.

math

This has the usual set of floating-point functions; you know, `Acosh`, `Cos`, `Ceil`, `IsNaN`, `Max`, Bessel functions, `Erfc`, and so on. Angles are in radians, logarithms are natural where not otherwise specified. There are functions to convert floats to and from raw bits.

sort

`sort` provides an exchange sort, with stable and unstable variants, and it's somewhat less comfortable than its Python equivalent not only because you have to import it explicitly, but because it requires that the sequence you're sorting implement `sort.Interface`, which is impossible for raw slices. (The *sequence* implements the interface, not the data items being sorted.)

This module contains heapsort, insertion sort, binary quicksort with some optimized pivot selection, and mergesort; the quicksort fails over to heapsort if it goes badly, an approach sometimes known as "introsort".

The mergesort is apparently a 21st-century optimization of binary mergesort which uses only logarithmic space rather than the usual linear space, at the cost of some extra swaps.

The package also implements binary search as `sort.Search`, generalized to the case of a general function over integers 0..n (i.e. not just for searching sorted sequences in memory).

For convenience, there are a number of wrappers for sorting and searching built-in data types.

Somewhat less basic packages

These are very common things but not quite as absolutely basic as the ones listed above. These are `flag`, `log`, `bufio`, `math/rand`, `time`, `net/http`, `net/url`, `net/mail`, `mime`, `regexp`, `encoding/gob`, `encoding/json`, `encoding/binary`, `image`, `C`, and `yacc`.

flag

This is the standard way to implement command-line parsing. For better or worse, it doesn't support combinable Unix-style single-letter flags, just long-only options, and it doesn't support GNU-style options following non-option arguments.

log

`log` is the standard way to log errors and other messages. This is how you debug your programs, and `log.Fatal` or `log.Fatalf` is how you report fatal errors and exit. There's no tagging, logging levels, complex object serialization, or filtering, but you can set a prefix and twiddle a couple of formatting flags. You can create different `log.Logger` objects with `log.New`, sending information to the same or different files, and pass them around as you like.

I wanted to put this in the "Really basic packages" category, since I import it in every Golang program I write, but the truth is that you

can get by without it a lot more easily than you can get by without `math`, `sort`, or `strconv`.

bufio

This adds I/O buffering to an `io.Writer` (for performance) or an `io.Reader` (so you can put back already-read bytes or runes, for parsing reasons, as `fmt.Fscan` does). It also contains `bufio.Scanner`, which parses things like (limited) CSV. (There's also a CSV parser in `encoding/csv`.)

math/rand

This is a random number generator, which includes a shuffler in the form of `rand.Perm`.

time

This package mixes together access to the real-time clock (`time.Now()`), calendrical calculations, formatting, time zones, and scheduling. The time format used internally is nanosecond-precision and has its zero value at the beginning of year 1 CE in UTC.

This package includes a syntax for durations, which are int64 nanosecond counts; the syntax accepts "300µs". ♥

Because Go has no operator overloading, the Time type has .Add and .Sub methods to interact with Duration.

net/http

This includes a fairly easy HTTP/1.1 and HTTP/2 client and server with TLS support. An HTTP server fits in a Tweet:

```
package main

import . "net/http"

func main() {

        HandleFunc("/", func(w ResponseWriter, r *Request) { w.Write([]byte("hello
oo")) })
        ListenAndServe(":8080", nil)
}
```

This also provides a convenient interface to the query-string parsing in `net/url`; the `http.Request` object above has a `FormValue` method.

net/url

`net/url` does URL parsing, construction, escaping, unescaping, relative URL resolution, and query-string encoding and decoding.

```
u, err := url.Parse("http://bing.com/search?q=dotnet")
```

net/mail

`net/mail` parses RFC-822 (well, RFC-5322) mail messages and addresses, but not MIME, and it doesn't format mail messages, just parses them.

mime

`mime` implements "parts of the MIME spec", including reading /etc/mime.types and the b-encoding and q-encoding used in mail headers. Subpackages implement quoted-printable and multipart encoding, but I don't think there's anything here that will give you a decoded email message body directly; this package is more aimed at handling mail headers and HTTP messages.

regexp

This implements a fairly Perl-compatible regexp engine, including non-greediness and the Python extension of named `(?P<foo>bar)` capture groups, but with no backreferences. Given the history of Russ Cox and Ken Thompson in writing high-performance regular expression libraries, using DFAs that can't handle backreferences, you would think that this library would be super fast, but usually it seems to be noticeably slower than Python's and Perl's. It does, however, avoid the exponential behavior characteristic of NFA engines in cases like this:

```
// Pathological regexp.
// The Python equivalent takes exponentially long:

// __import__('re').compile('(x|xx)*y').match('xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxo
oxxxxxxxxxxx')
package main

import "regexp"

func main() {

        regexp.MustCompile("(x|xx)*y").FindString("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxo
oxxxxxxxxxxxxxxx")
}
```

encoding/gob

This is a generic serialization/deserialization system, using a Go-specific serialization, and it seems to be the closest equivalent to Python's `pickle`, though it doesn't support circular references. It supports maps, structs, arrays, slices, bools, signed and unsigned integers, floating-point numbers, complex numbers, and strings; structs include only exported fields, and functions and channels are omitted. Interface types are supported with a little more hassle.

It is not necessary to register user-defined types before sending them or for them to implement any particular interface, although there is an interface that they *can* define to override the default marshaling. There is some limited support for schema evolution even in the default marshaling.

encoding/json

`json.Marshal` converts a Golang value into JSON; it supports bool, numbers, strings, arrays, slices, maps from string, and structs (except for unexported fields or fields tagged with "-".) This is actually the only use I've seen so far of struct field tags.

Naturally, of course you can override the JSON serialization by implementing an interface.

`json.Unmarshal` takes an optionally-nil destination argument, which I suppose allows it to determine what types to deserialize things as.

This module *does* support reading a sequence of JSON values from the same input stream; perhaps more interestingly, it has some minimal support for sequentially reading JSON items inside an outer JSON wrapper.

encoding/binary

`encoding/binary` contains functions `Read` and `Write` which can be used to parse and generate binary data in an externally-imposed format. They're slow because they use reflection ("Python-class slow," as Tommi Virtanen explained to me). This provides an easy way to do what the Perl `pack` and `unpack` functions do, or `struct.pack` and `struct.unpack` from Python.

image

This module and its submodules provide support for encoding and decoding PNG, GIF (including animation), and JPEG files. You generally will load the images by calling `image.Decode` without explicitly mentioning the specific image format module, except to import it, but to save e.g. a PNG you need to call `png.Encode`.

The `image.Image` interface looks impossibly inefficient, requiring an indirect function call and some coordinate arithmetic for every pixel.

It looks like the PNG loading is eager and ahead-of-time, storing the image in RAM, although `image.Image` doesn't strictly require that.

The `image` module also contains some basic 2-D graphics stuff, like `image.Rectangle` (bounding boxes) and `image.Point`.

C

cgo isn't actually a Golang package; it's a way to invoke statically-linked C libraries. But you import it as a package named "C", and you add some special comments before that import in order to link in the stuff you want.

...and then the Go compiler searches your directory for files named `*.c`, `*.s`, `*.S`, `*.cc`, `*.cpp`, or `*.cxx` (but not `*.C`) to compile with the C, C++, or assembly compiler. Except that I'm not clear which compiler it uses for this: GCC or the Plan9 C compiler?

Presumably if you want to load and invoke shared libraries written in C, this is the way to do it.

yacc

If you want to parse something more than regular languages, `go tool yacc` is probably the thing to use. It's mostly undocumented, but it's similar enough to Bison, ocamlyacc, etc., that it should be reasonably attainable.

Somewhat more recondite packages

I've also taken some notes on some other packages that seem less central to me.

html/template

`html/template` gives you convenient XSS-safe HTML templating. Its capabilities are fairly elaborate, and there's also a `text/template` package for doing the same kind of thing without the HTML escaping.

syscall

This deprecated package provides access to basically the entire Linux (or other) native system call interface, including weird things like epoll(7), `ETH_P_IRDA`, inotify(7), `SCM_RIGHTS`, and openat(2). The `unsafe` package has some details on directly invoking Linux system calls.

This is also the package where you get errno values, if you want to test those against the `Err` attribute of e.g. an `os.PathError`.

encoding/xml

`encoding/xml` contains an XML parser and emitter, some HTML parsing, and something that looks suspiciously like another generic serialization system like Gob and the one in `encoding/json`, but really isn't.

The serialization system handles arrays, slices, structs, and interfaces, but not maps, and like the JSON serializer, it optionally uses struct field tags. It seems to be designed to allow you to produce arbitrary XML by marshalling structs, and parse almost arbitrary XML by unmarshalling.

This module doesn't provide DOM or SAX interfaces, but it has interfaces that are sort of similar.

net

This library provides sockets, basically, but with a somewhat nicer interface. Like, actually a dramatically nicer interface. TCP sockets have SetNoDelay and SetLinger methods, for example, and you can invoke `net.Dial("tcp", "google.com:http")` to open a connection.

index/suffixarray

`suffixarray` provides an in-RAM search index that supports regular expression searches, though using a log-linear-time construction algorithm rather than a linear-time one. On my laptop, it takes about 1.8 seconds to index a megabyte, about 26 seconds to index 10 megabytes, and 540 seconds to index 100 megabytes; then, each match for a simple regexp in the 10MB index takes 1.3 ms, and each match in the 100MB index takes 2.5 ms. It uses 1.6 GB of RAM to index 100 megabytes, which is maybe a bit excessive.

You can write the index to a file, which is about 4.8 bytes for each byte that you originally indexed. My laptop can then read the index in at about 100 megabytes a second, which is a necessary prerequisite to doing searches with it.

As a point of comparison, I tried `index/suffixarray`, the raw regexp engine, and the Python regexp engine to find the '^From ' lines delimiting messages in a 100MB mbox. The indexed search found the 3593 `From_` lines in 17.5 seconds. Just reading the file in and running the regexp engine over it found them in 12.6 seconds. The Python version took 2.0 seconds.

This seemed like unreasonably poor performance, so I tried searching for a spammer's email address. The Python version found

the three matches in 430 ms; the Golang brute-force version found them in 210 ms; and the indexed version found them in 4.3 seconds, of which 4.0 seconds were spent reading the index.

Just for kicks, I tried a Perl version. It produced the 3593 `From_` lines in 220 ms and the three hits for the spammer's address in 195 ms, but upon trying to find all the lines *containing* the address (regexp `^.*VanceE.McCray.*`), I killed it after 21 minutes. Python managed to finish that job in 2.7 seconds, the Golang brute force version in 36 seconds, and the Golang indexed version in 45 seconds.

So I haven't been able to find any cases where `index/suffixarray` is faster than doing a brute-force regexp search on the file data in RAM, even though it uses several times as much RAM.

runtime

`runtime` provides control over the garbage collector (including setting finalizers), a `Caller` function to walk the stack, and a bunch of profiling stuff I don't know how to use yet. (I guess I could call the `runtime.debug.PrintStack` function periodically and save the results, and that would be a crude profiler.)

net/rpc

`net/rpc` is an RPC system using the `encoding/gob` serialization; a JSON-serialized variant is in `net/rpc/jsonrpc`. It doesn't enjoy a lot of syntactic sugar.

zip

`zip` lets you read and write zipfiles, but only items compressed with the "store" and "deflate" methods. It uses the same `io.Reader` and `io.Writer` interfaces everything else does (or rather `io.ReadCloser` and `io.Writer`.)

Topics

Ghettobotics: making robots out of trash

Kragen Javier Sitaker, 2013-05-17 (41 minutes)

I've been thinking for a while about low-cost electronics projects. What could you make out of the garbage? People throw away electronics all the time, and recycling it is an interesting sustainability and autonomy experiment.

The ultimate goal of Ghettobotics is a self-sustaining industrial economy that consumes nothing but discarded electronics and other trash and produces, with a minimal amount of human effort, useful robots. To this end, every tool and material used in the Ghettobotics program can be made out of trash, by Ghettobotics itself.

A big part of the problem is that you can't make much out of electronics without some fairly specialized tools. So the first problem is how to build the tools out of garbage --- maybe using a previous generation of tools you bought in a store.

So what tools do you need? These would be desirable:

- A soldering iron.
- Desoldering equipment.
- A basic VOM: volt-ohm-milliammeter.
- An LCR meter.
- An oscilloscope.
- A display.
- Current-limited variable-voltage power supplies.
- A logic analyzer.
- A JTAG board.
- A serial terminal.
- A PROM burner.
- A pick-and-place robot.
- A solder stencil cutter.
- A reflow oven.
- A CNC mold cutter.

You also need some basic electronic and mechanical parts; the following are desirable:

- Solder.
- Wires.
- Connectors.
- Resistors, capacitors, inductors, transistors, diodes.
- Displays.
- Speakers.
- Cases.
- Motors and motor controllers.
- Amplifiers.
- Analog-to-digital converters.
- Digital-to-analog converters.
- Presence sensors.
- Light sensors (pixel array or otherwise).
- Motion measurement.

• Microcontrollers.
• Memory.
• Batteries.
• LEDs.
• Other lights.
• Buttons.
• Touchpads.
• Wireless communications modules.
• Range measurement devices.
• Power supplies.
• Solder paste.
• Temperature sensors.
• Relays.
• Shift registers.
• Wheels.

In taking apart and buying electronics, I'm highly impressed by the degree to which they seem to have been designed by people who can't program, and therefore prefer electronic or even mechanical complexity to simplify software. Now that you can get low-power microcontrollers for under a dollar, this seems absolutely idiotic to me, except in cases where software simply won't cut it. I conclude that knowing how to program is an incredible secret weapon in electronics design.

A soldering iron

This is basically a resistive wire that can get hot and has a pointed tip. This should be relatively easy to improvise, I haven't tried. In a pinch you don't even need the resistive part; you can heat up a chunk of metal with fire.

For cleaning the tip, if cellulose sponges are hard to find, you can use wet paper or cardboard.

A temperature-controlled soldering iron would be really useful; you can use a thermal-resistance "voltage divider" with your hand as the known-temperature "ground" to measure the tip temperature with pretty much any old temperature sensor. Then you just need a little hysteresis and something to turn mains power on and off with; a triac? Not sure. A high-power relay would work.

Desoldering equipment

Desoldering parts with just a soldering iron is feasible, but tricky and messy. It's especially difficult using just a low-wattage pencil-type iron, which simply can't handle desoldering high-current or heat-sinking connections; a high-power soldering gun would have less risk of damaging chips. Hot-air heating is apparently the standard approach here for rework stations, but I imagine you could also use a reflow oven or hot plate.

Rework stations also have desoldering irons different-shaped tips for desoldering different parts quickly, heating up all their pads at once. I don't know how to improvise this.

Beyond just heating solder up, it's also handy to have other metals to mix into it to lower its melting point. Standard lead-tin solder alone can help if you're trying to desolder modern lead-free devices; lead might work too. The other alternatives that occur to me sound either expensive or, in the case of mercury, dangerous.

Solder braid or a solder sucker also helps a lot. Presumably if you have some copper wire sitting around you can braid it into solder braid, but a solder sucker would help more if you want to be able to recycle the solder. An improvised solder sucker could maybe be made from a metal tip with heatsink fins on it and a rubber squeeze-bulb, but it's going to be difficult to make something that performs as well as a standard Teflon tip.

A VOM

The traditional way to build a VOM is with a galvanometer, a network of resistors with a selector switch, and a battery for the ohmmeter bit. You can still do this (hard disks are built around voice-coil actuators that should work well for the galvo) but it's probably more reasonable to use a cheap analog-to-digital converter that measures voltage directly, then use a microcontroller to convert the reading to something comprehensible.

Historically, VOMs have been designed around linear components, because nonlinearity is hard to compensate for in the analog realm. It's really easy to compensate for in the digital realm, so it probably makes more sense to not worry so much about linearity.

The electrically simplest design for a VOM is something like a microcontroller with a built-in ADC, a voltage divider (one large resistor between the voltage input and the ADC pin, and one small resistor from there to ground), a speaker or piezoelectric sounder, and a battery; total of five components. Current inputs connect directly to the ADC pin and ground; voltage inputs go through the large resistor, enabling the measurement of voltages much larger than the microcontroller can deal with directly. An additional input pin with a pullup resistor and ADC input is used to measure resistance. The microcontroller uses speech synthesis to communicate the results of the measurement.

If you're willing to require an external display, you can ditch the speaker and send the measurement results over a serial or USB connection; if you're willing to require an external power supply (say, an unplugged laptop on battery, connected via serial or USB), you can ditch the battery too, getting down to three components, two of which are resistors.

Back-of-the-envelope calculation says that with a 10-bit ADC with selectable 5V and 1.1V range (like what you get on an Arduino) and a 50:1 voltage divider, you can measure up to 250V with 0.25V precision and up to 55V with 0.05V precision. If you then manually connect smaller voltages (below 5V) directly to the ADC pins, you can get up to 5V with 0.005V precision and up to 1.1V with 0.001V precision.

You can use a microcontroller without an ADC if you can convert the quantity you want to measure to a measurement of time to exceed a digital threshold; for example, you can hook up a capacitor between a couple of pins of the microcontroller, instead of a resistor, and measure an RC time constant. You may be able to get better precision this way, but it depends on your capacitor's temperature-constancy and on your oscillator's constancy. You may be able to calibrate to temperature, both by charging the capacitor under microcontroller control at start time, and by using a temperature sensor. (Many microcontrollers have temperature sensors

built in.)

A very useful feature of VOMs in general, lacking in the normal ones, is to correct for noisy connections. If you average over time, a shaky connection to a 10-volt power source might look like 9 volts, 5 volts, or 1 volt; but if you look at the voltage waveform at microsecond resolution, you can tell that it's 10 volts with a shaky connection, not 5 volts or whatever. VOMs (and LCR meters) built around ADCs should be intelligent enough to do this correction.

Nonlinear frontend

It would be interesting to examine some diode-resistor networks to see if you can get a wider range with good precision by building a nonlinear input circuit, avoiding the need to switch ranges in order to measure large and small voltages --- and, perhaps more importantly, the possibility of blowing up your meter by hooking it up to a too-large input signal. I'm thinking something like using a high-current power-rectifier diode in place of the low-value resistor in the voltage divider, so that the voltage signal you measure is the diode's minimal turn-on voltage drop, plus an additional voltage that's nearly logarithmic in the current that passes through (which is more or less the input voltage E divided by the larger resistance R between the input and the ADC pin).

Something like the ON Semiconductor MBRAF1100T3G Schottky power rectifier diode would be ideal: assuming 25°C, at 20 mA it drops 0.42 volts; at 200 mA it drops 0.54 volts; at 2000 mA, which is above its maximum rating, it drops 0.70 volts; and at 20 A, it drops 0.98 volts. (Serious temperature compensation would be needed to get a reasonably accurate measurement, because the voltage varies by more than a millivolt per °C.) It would be very difficult to drive the voltage across this diode into a range that would damage a microcontroller, even one without 5V-tolerant inputs.

If your ADC precision were 1mV, this would give you measurement precision of around 1% for currents in this range, at the cost of imposing a significant voltage drop.

If we assume that the diode doesn't turn on until 0.42 volts, which is probably about right, then you'd have effectively an ideal voltmeter up to 0.42 volts, and above that, nonlinear response depending on your input resistor. You probably want your input resistor to be quite large, since the diode isn't going to provide much input impedance above half a volt, but you can't make it too large or you won't be able to measure voltage at all. With an absolutely minimal requirement of being able to measure a doubling of input voltage per 1mV increase at the ADC pin, and only trying to measure up to 100V (the maximum reverse-bias rating for this diode), you need 100V to give you 0.428 volts at the ADC pin, which would be about 23mA through the diode, so your input resistor can be at most about 4kΩ. With this network, you can get better voltage precision than the quarter-of-a-digit that this implies, but only at the cost of a lower input impedance.

I have no idea how common Schottky power rectifiers are. I've found lots of power rectifier diodes in the garbage but I have no idea what kind they are. I assume most of them are large standard silicon diodes.

So you might be able to do better with a more elaborate network,

or using a separate input pin for voltage measurement. You probably want a diode that's rated for a lot less than an amp. The Vishay SD101A, for example, goes from 200mV at 0.01mA to 1000mV at 15mA, and is rated for 60V max reverse voltage; the same 4kΩ input resistor and 1mV measurement precision at the ADC pin would then give you some ±2% precision on your voltage measurement over that range and down to 50mV, rather than the +100%,-50% you'd get with the power rectifier. That is, with this configuration, your meter can measure voltages covering three orders of magnitude with a consistent ±2% error. That's probably better than the analog multimeter I normally use.

(Incidentally, this configuration might also be useful for digitizing audio and other signals; you only get about 34dB of SNR, but it's a consistent 34dB across 60dB of dynamic range, instead of having a noise floor 34dB below the strongest possible signal. You get 94dB separation between the strongest measurable signal and the weakest measurable signal, as if you were using a 17-bit ADC instead of a 10-bit ADC. Analog Devices recommends this in their application note MT-018, and explains that this resistor-diode configuration is how Bell originally implemented μ-law for voice digitization!)

An LCR meter

If you're fishing inductors and capacitors out of the trash, or even just using surface-mount ones, you probably need to be able to measure them; the device you need is an "LCR meter", and they are available new for less than US$40. But variants on the previous section's ohmmeter circuit should be able to provide reasonable measurements.

A really handy physical design for this is the "smart tweezers" design, where the probes for your LCR meter are flexible and elastic, so you can pick up the component under test between them, or even touch them to the component in-circuit before bothering to desolder it.

Many microcontrollers have ADCs but no DACs, which means that stimulating a device under test with a sine wave is out of the question without further hardware. Measuring the impulse response of the component is theoretically sufficient, but you can also generate digital white noise to stimulate the component with, then compute the Fourier transform of the noise and the measured current waveform.

This test provides enough information that you can automatically test the hypothesis that the component you're measuring is actually just an RL or RC circuit, and rather than giving dubious numbers when it's clearly not, you should try some more elaborate hypotheses --- more elaborate network models, or nonlinear components like diodes and transistors.

There are several articles about homebrewing LCR meters with microcontrollers:
http://www.kerrywong.com/2010/10/16/avr-lc-meter-with-frequency-measurement/, http://electronics-diy.com/lc_meter.php,
http://web.archive.org/web/20080405215220/http://ironbark.bendigo.latrobe.edu.au/~rice/lc/,
http://py2wm.qsl.br/LC_meter/LC_meter-e.html,
http://reibot.org/2011/07/19/measuring-inductance/. Cypress has

an example project using their analog PSoC chips at http://www.cypress.com/?app=forum&id=2492&rID=76890.

An oscilloscope

This can be usefully separated into data acquisition and display.

Data acquisition

At times I've had some success using my sound card in place of an oscilloscope. If the signals are within the audio range, a standard 8Ω speaker can also provide some valuable information about them. But what you really want is to digitize the signal and display it on a screen.

You can, of course, use any ADC. But a standard cheap analog oscilloscope has 20MHz bandwidth, which would require a 40Msps ADC, and actually a 20MHz analog oscilloscope can still detect signals much higher than 20MHz. 40Msps ADCs are not commonplace. AVR ADCs can be run at up to a couple of Msps with reduced resolution (about 6 bits), and I found a three-channel 12-bit 2Msps ADC in a discarded flatbed scanner. Some such scanner ADCs can digitize a single channel at three times their three-channel speed.

Many of the remarks about cheap-shit VOM frontends apply here too. You can use a nonlinear input network and restore linearity in software.

If you want to examine a high-frequency signal that is repetitive or nearly so, you should be able to downconvert it into a lower frequency band in order to digitize it with a lower-data-rate ADC. In fact, if the sample-and-hold circuit on your ADC is fast enough, you may be able to do this with no extra analog circuitry!

Another approach is to use many ADCs in parallel, triggering them in a round-robin fashion. The guts of about 8 scanners would suffice to produce a 40Msps data-acquisition system this way, with a little bit of coordinating circuitry. This, however, also depends on the sample-and-hold circuits being sufficiently fast.

Oscilloscope display

If you have a laptop, you should use that, because you can afford dramatically greater amounts of computation and storage for display that way than if you build it yourself out of garbage. See below.

Analog oscilloscopes provide great variation in intensity, carrying additional information. Oona Räisänen demonstrated the rather impressive difference in her post Rendering PCM with simulated phosphor persistence:

Now how cool is that? It looks like an X-ray of the signal. We can see right away that the beep is roughly a square wave, because there's light on top and bottom of the oscillation envelope but mostly darkness in between. Minute changes in the harmonic content are also visible as interesting banding and ribbons.

A display

For lots of measuring-equipment stuff, you need a display. The traditional approach is to integrate the display into the measuring instrument, but it probably makes more sense to separate them, so that you can amortize the effort of building a good display over several measuring instruments, each of which can then be extremely

simple. You may want a simple seven-segment display on the measuring instrument itself (easily recoverable from discarded calculators or watches) but more detailed data display is probably better with a separate display device.

People are throwing out CRTs all over the place, but here in Buenos Aires, it's difficult to find an intact discarded CRT --- cartoneros break off the yoke to recycle, ruining the tube. But if you can find an intact CRT, you can probably drive it with a VGA or composite NTSC or PAL video signal from a microcontroller, without having to worry about any of the electronics inside the case.

The TVout library for AVR microcontrollers works quite well with just two output pins and a couple of resistors to generate a blocky one-bit black-and-white NTSC video signal, which is dramatically better than nothing, but I think you can do better with even a few bits of ADC. A three-bit or four-bit R-2R ladder DAC is easy to construct.

Trickier is that the TVout library uses a lot of memory (kilobytes) for a framebuffer. This dramatically limits the resolution, and inhibits the use of grayscale. More elaborate software could display higher-resolution vector graphics.

A perhaps more interesting approach is to use displays from discarded cellphones, which typically have their own framebuffer. You send them commands to write (and maybe even read) their framebuffer, so you don't need to have enough memory yourself.

Audio output is another, cheaper option. Aside from simple approaches like beeps for continuity testers, there are several implementations of speech synthesis on AVR microcontrollers, one of which is Cantarino (LGPL); I think you can get comprehensible speech at under 100k 8-bit multiply-accumulates per second, which is within the capability of many microcontrollers; it should be about 0.2% of the CPU cycles of an Arduino. Arjo Chakravarty has even done speech *recognition* on an AVR, called µSpeech.

Bench power supplies

For testing, you need a bench power supply with selectable output voltage, one that's not necessarily highly efficient, but is hard to burn out. I found an adjustable-voltage DC-DC converter chip in a discarded scanner, but I haven't yet tried using it for anything.

I don't know how to build robust power supplies, but I assume that big power clamping diodes and big current-limiting resistors are involved.

A logic analyzer

This is especially important for reverse-engineering the undocumented controllers often found in discarded electronics, but you also need it to diagnose problems in the things you build. You can make a slow logic analyzer (say, a few million samples per second) with just a microcontroller using its GPIOs; what remains is only to view the waveforms.

A fast logic analyzer probably needs shift registers or something.

A JTAG board

Lots of embedded systems can be put into debug mode and even

reprogrammed with a JTAG interface; you can also use JTAG's boundary-scan functionality to map out board connectivity and diagnose connectivity faults in your own devices. Four pins on a microcontroller, hooked up to a standard 6-pin JTAG header, suffices to control other things through JTAG.

One of the most common JTAG cable chips is the FTDI FT2232, which you're not likely to find in the garbage, unless it's garbage from an electronics lab. But you probably want to emulate it in software so you can use existing JTAG software on your laptop.

Felix Domke wrote a paper about JTAG for reverse engineering in 2009, which he presented in much more detail in a talk at 26C3, and the NSA@home project used JTAG to reverse-engineer boards that had FPGAs on them to repurpose the FPGAs for hash cracking; they published software for this called "jrev".

Among other things, JTAG lets you identify JTAG-attached chips automatically and use them as shift registers.

A serial terminal

Any old computer will work as a serial terminal, with its own display; but you may want a microcontroller-based serial terminal. A "terminal emulator" program that does what the Arduino's serial monitor does is quite simple; it doesn't even support backspace. A more full-featured serial terminal can easily fit into a microcontroller.

A PROM burner

If you're recycling old garbage electronics, their ROMs can be useful for several different purposes:

• Dumping the ROM can allow you to repair other instances of the same device that have damaged ROMs, or to reverse-engineer the controller that runs the program found in the ROM.
• If the ROM is reprogrammable, you can use it to store other data, or even...
• if the ROM is programmable and asynchronous, use the ROM as a programmable logic device whose inputs are the address lines and whose outputs are the data lines.

To reprogram an EPROM or EEPROM you need at least a PROM burner and possibly an EPROM eraser, which is a shortwave ultraviolet light. PROM burners are relatively straightforward to build with a microcontroller, although they do often need control of 12-volt voltage, which you can achieve with many of the chips you'd use for motor control.

A pick-and-place robot

If you're building your own circuits, eventually you will benefit from being able to assemble them automatically instead of by hand. I don't know all of what is involved in this, but I think that a lot of the mechanical and electronic difficulties of high-precision X-Y positioning are already dealt with by many inkjet printers.o

A solder stencil cutter

This lets you squeegee solder paste onto a printed circuit board so you can solder your surface-mount components all at once with a

reflow oven instead of one at a time. It's basically a plotter. You should be able to use overhead-projector transparency film for the stencils themselves, but X-ray film might be an alternative where you don't have sufficient overhead transparency film handy.

I don't know what's involved in actually cutting these. I think a vinyl-cutting machine might work.

A reflow oven

A reflow oven bakes your circuit board until the solder paste melts and all of your components are glued into place. SparkFun has a tutorial on their site on how to use a hot plate for this instead, but presumably you can also build your own reflow oven from a metal box, some heating elements, and a thermostat.

A CNC mold cutter

Michal Zalewski's "Guerrilla guide to CNC" explains that often it's more reasonable to make even one-off robot parts by cutting molds for them and then casting them than to cut the parts directly. You can go a long way with a three-axis robot for this, which might be the same one that does your pick-and-place.

Long ago I saw a video of an X-Y robot that avoided the difficulties of gantry construction by using two turntables in parallel planes, one of whose edge passes over the center of the other; by this means, you need only rotary motion rather than linear motion. This way, X-Y positioning needs no more mechanical complexity than a couple of bicycle wheels driven by belts from motors, which is considerably less than the mechanical complexity of an inkjet printer.

Solder

Recycling solder is a little tricky, but I think a solder sucker is probably sufficient. You probably need to add new flux; I think pine pitch is sufficient, but haven't tried it. You probably also want to keep your RoHS lead-free solder separate from your traditional lead-tin solder.

Wires

Ethernet cables typically have nice solid wires that are easy to work with.

Connectors

It's a pain in the ass to deal with multi-board circuits that are connected by wires soldered to all the boards; they tend to break easily. Connectors are easily recyclable. The most robust connectors are from automotive systems.

It's also valuable to be able to connect to standard cables: audio, video, network, etc. Ethernet cables are especially useful for unintended uses, since they tend to be long.

Resistors, capacitors, inductors, transistors, diodes

Small ones of these are all over the place (many of them are literally a dime a dozen new from Digi-Key). The tricky part is keeping them

categorized, since you need so many of them. It may be worthwhile to catalog them with an LCR meter in situ, only desoldering them from their original board when you need them after looking in a search engine. Smart tweezers on a pick-and-place robot could theoretically categorize them into little boxes.

Larger items are somewhat harder to find. Discarded old fluorescent light fixtures typically contain large inductors as "ballast"; I've found large resistors in discarded microwave ovens, and of course they're often used as heating elements; large diodes are often found in power supplies. Large transistors, capacitors, and transistors, I have no idea.

Displays

As components rather than tools --- I've mentioned CRTs, cellphone displays, and 7-segment displays from calculators and watches, but there are many other premade displays out there.

LCDs can be recovered from broken laptops.

I haven't yet tried to connect new wires to a broken e-ink screen, but if it's feasible, broken Kindles would be a rich source of excellent displays.

Vacuum fluorescent displays are among the most visually appealing to me, but they also require a high-voltage source, so they're tricky to work with.

Laser displays are traditionally done with a couple of galvanometer-driven mirrors, and they look awesome. I've speculated about building laser displays by deflecting mirrors using speaker cones instead, which would make them cheap as dirt; and you could use a laser display with a light sensor as a camera or scanner, too.

Speakers

These are the easiest components to scavenge: half-broken earphones, dynamic speakers in broken TVs, piezoelectric speakers from greeting cards, and so on.

Cases

When I was a kid, we used cardboard boxes for our electronic projects; but nearly anything can be used as a case: empty bleach bottles, books, tin cans, discarded pots, blocks of wood, hunks of styrofoam (modulo ESD issues). Many discarded electronics include particularly robust steel cases, but toys provide plastic cases that are easier to drill.

Motors and motor controllers

Inkjet printers typically have two relatively powerful (around 20W) stepper motors and PWM-controlled H-bridges to run them. Flatbed scanners will have one, which will be lower power. Laser printers have motors too. Power-supply fans have brushless DC motors. CD drives need at least two motors, one to spin the CD and one to move the head, but may have a third one to open and close. Disk drives have high-speed motors with the finest bearings in the world. Washing machines, refrigerators, and the like have much higher power motors, into the hundreds of watts and beyond; car

starter motors are *really* high power, into the tens of kilowatts, but only work on a very short duty cycle.

Even small stepper motors probably need more than 5 volts. Chips like the ULN2003 can help with this, but then there are PWM H-bridge chips like the LB1845.

Recent Epson inkjet printers use proprietary motor control chips which seem like they'd be really useful if they can be reverse-engineered.

Bigger motors need things that can control several amps of electricity, often relays.

Amplifiers

You need amplifiers for lots of things: driving speakers, controlling motors, and boosting delicate analog signals so they can get to where you digitize them. Amplifiers tend to be fairly specialized. Audio equipment tends to have lots of amplifiers in it, including opamp chips, that won't work much outside the audio range; but an awful lot of useful signals are in the audio range.

Analog-to-digital converters

ADCs are fundamental to interfacing with the outside world. Some microcontrollers have them built in; if not, you may be able to do delta-sigma DAC with not much more than an analog comparator, such as an opamp, and a resistor and capacitor. But lots of discarded electronics has separate ADCs in it, because keeping the ADC on a separate chip reduces electromagnetic interference. So you can find separate high-quality ADCs in scanners and some audio equipment, at least.

Digital-to-analog converters

Even more audio equipment has high-quality digital-to-analog converters, and you can cheat by using PWM or PDM at a high frequency; just be careful your high carrier frequency isn't going to blow up any tweeters.

Presence sensors

Optical presence sensors are ubiquitous in printers and scanners; the light from an infrared LED is blocked, or not, on its way to a phototransistor. Typically they use them as a "home position" sensor.

Light sensors

Scanners, digital cameras, cellphones with digital cameras, and optical mice contain light sensors. LEDs can also be used as (slow, noise-sensitive) light sensors. (I haven't been able to reproduce the Mitsubishi paper that reported microsecond-range sensing times with reverse-biased LEDs; I get more like hundreds of milliseconds.)

Motion measurement

Robotics involves being able to measure how far things have moved and where they are --- at least parts of the robot, and maybe things the robot interacts with. There are lots of ways to do this. Stepper motors emit pulses as you turn their shafts; servomotors typically use integrated potentiometers; optical encoders like those

used in ball mice give you a quadrature-encoded signal that tell you how fast things move; optical mice take a high-frame-rate video of the table the mouse is sitting on. I've seen inkjet printers that use linear optical quadrature sensors to sense the printhead position, as well. At the high end, shaft encoders give you an absolute shaft position readout, typically using Gray code.

Digital cameras in general can be used with machine vision algorithms to measure position and motion. This is easier with structured illumination, e.g. a laser pointer.

Binary chain codes are a promising alternative to quadrature encoding to measure absolute positions instead of just relative motion, with some bootstrapping time. Random noise, as long as it's repeatable, should work almost as well as a designed binary chain code.

Accelerometers and gyroscopes are a particularly important kind of position sensor for many purposes: balancing on wheels, flight control, and protection from falls, for example. Off-the-shelf MEMS accelerometers and gyros are what current cellphones and laptops use, but they aren't old enough for many of them to have made their way into the trash stream yet. The traditional kind of accelerometer is a sprung weight controlling a potentiometer slider, and it seems like that sort of thing shouldn't be too hard to improvise, given other kinds of position and distance measurements.

Microcontrollers

This is one of the biggest problems for actually bootstrapping with garbage. You need *something* you can program to take advantage of all the amazing digital circuitry you can fish out of the garbage. You can get really awesome Flash microcontrollers from Digi-Key for under a dollar these days, but options in garbage are more limited; garbage electronics tend to be old, and mass-produced microcontrollers often use mask ROM instead of Flash, or have fuses blown that prevent them from being reprogrammed.

One option is to use a regular old computer running a real-time operating system like Linux-CNC's EMC2.

Another option is to find something that has the controller program stored in a separate ROM chip, and replace that ROM chip. This requires you to understand the instruction set of the controller, which is often not documented. Lots of reverse engineering is required here.

It turns out that lots of interesting devices from the last few decades actually do keep their program in a separate chip.

Memory

Lots of devices have RAM, though usually small; fewer have Flash. It might make more sense to look for discarded SD cards, which will be hard to notice in garbage, just because they're so tiny; but they're supposedly easy to interface with, because they support SPI. I've encountered a DVD player with a one-megabyte Flash chip in it, and a discarded cellphone with a 1GiB mini-SD card.

Actual old disks might make more sense for heavier robots. A disk that's "only" ten gigabytes is too small to be worth bothering with if you have any money; but ten gigabytes is quite a bit compared to what microcontrollers typically have built in. I'm not completely

sure, but I suspect SATA is easier to deal with than parallel IDE/ATA.

Batteries

Few discarded devices have working batteries. Some have marginal batteries. Fewer have workable rechargeable batteries.

Really building batteries out of trash will probably require re-purifying the materials from dead batteries and building fresh cells.

LEDs

LEDs are in just about every discarded electronic device. They wear out a bit after long use, but they remain useful.
High-brightness, blue, and white LEDs are recent (since about 1990), but you can still find them in trash occasionally.

Other lights

LCD displays and flatbed scanners typically include tiny cold-cathode tubes and high-voltage power supplies for them.

Occasionally you might find neon lamps and the like. Neon lights last *forever*.

When fluorescent tube filaments burn out, people typically discard the tubes. But I think it's possible to still run them as cold-cathode tubes; you just need a high-voltage power supply.

Buttons

You can use capacitive touch sensing to make buttons out of aluminum foil and paint; but also many devices contain multiplexed button matrices. Any light sensor can be used as a button.

PS/2 keyboards are commonly discarded and easy to interface to microcontrollers. Sometimes the discarded keyboards themselves work fine; at other times it is more to your advantage to hook up a couple of the keyboard matrix wires to some other kind of buttons.

Touchpads

I think you can make a resistive touchpad that works like a touchscreen out of pencil lead on paper.

Wireless communication modules

Only occasionally will you find a discarded device with working radio communication, other than cellphones, which are useless except for talking to the cellphone network, but there are DIY options: infrared, ultrasound, 900MHz, 2.4GHz; software-defined radio? Basically any discarded audio or video device (or, here, air conditioner) will have an IR receiver for the remote control, which will also have been discarded --- but possibly separately.

(Recent cellphones also have Wi-Fi, but I'm not seeing them yet in the trash.)

Presumably you could also use the parts of optical presence sensors as infrared transmitters and receivers.

Range measurement devices

Not only do you need to be able to tell where parts of your robot

are, you also need to be able to tell what's nearby.

Aside from the camera approaches mentioned earlier, ultrasound is a common ranging approach. I think you can do ultrasonic ranging with a couple of piezoelectric sounders, but I haven't tried it yet.

Power supplies

Nearly every discarded electronic device has some kind of power supply. Often it's broken, and fixing it or replacing it will fix the entire device. Sometimes it's partly broken, and some of its output voltages work fine. Power-supply chips are versatile and useful for many things.

Solder paste

If you want to reflow solder a surface-mount board, you need solder paste. You aren't going to find solder paste in discarded electronics, and in fact it has a quite limited shelf life. It's also difficult to buy in small quantities. I have no idea how to make solder paste out of garbage.

Temperature sensors

A lot of things (reflow ovens, temperature-compensated measurement, emergency scram on overheat) benefit from measuring temperature. Essentially every electronic component has significant behavior changes when its temperature changes, so in theory any electronic component can be used to measure the temperature!

Some current microcontrollers have built-in temperature sensors, but if not, you can probably build a bridge circuit with a couple of different kinds of resistors to give you a voltage that changes dramatically with temperature. Carbon-composition and wire-wound resistors have temperature coefficients that are opposite in sign. Alternatively, you could measure the RC constant of an RC constant.

Relays

All kinds of high-power devices (microwaves, refrigerators, washing machines, etc.) have relays controlling them. These are handy for robots too. Typical relays are only good up to a few kHz, but reed relays are good up to tens of kHz.

Shift registers

A crucial limitation on microcontrollers in general is the number of available digital GPIO pins. You can use a shift register to turn two or three GPIO pins into an arbitrarily large number of slower GPIO pins. But you aren't going to find a lot of discrete shift-register chips in modern discarded electronics.

JTAG to the rescue! Basically any JTAG-enabled device can be used as a shift register by using JTAG's "boundary scan" functionality. That piece of shit undocumented SoC you ripped out of some scanner, which won't even let you read its program from Flash, will work fine as a 200-bit shift register driven through JTAG!

Wheels

This is skipping quite a bit of mechanical stuff, but wheels are

important for mobility. Wheels are often available from discarded cars or discarded bicycles; bicycle wheels are likely more practical, and back wheels already come with sprockets attached, dramatically reducing the need for gearboxes.

Heavy robots probably need fairly powerful motors to propel their wheels.

Topics

B-Tree ropes

Kragen Javier Sitaker, 2019-09-24 (updated 2019-09-25) (19 minutes)

I just hacked together a quick rope-based string system in Lua, but it has rather alarming worst-case performance characteristics. I was thinking about improving such characteristics with B-trees.

The string and rope problem

If you build up a long string through successive concatenations, many string systems will suffer an $O(N^2)$ slowdown; for example, in LuaJIT this takes 2.4 seconds, which works out to 42 kilobytes per second; PUC Lua 5.2.4 is only slightly faster at 1.9 seconds:

```
N = 100000
s = ''
for i = 1, N do s = s .. 'x' end
```

CPython used to be really slow at this, but has a special optimization for this case now, so it takes such a small amount of time that it is difficult to measure accurately; the following slight variation still takes 700 ms, which works out to 140 kilobytes per second:

```
N = 100000
s = ''
for i in range(N): s = t = s + 'x'
```

This shows the expected $O(N^2)$ curve, taking 2.5 seconds for $N = 200\,000$ instead of $N = 100\,000$.

Moreover, in these systems, if the same large string occurs as part of M other strings, it uses up M times the space, and many of the concatenation operations are redundant.

It's common for such string-concatenation operations to consist of essentially variable interpolation — filling variable holes in otherwise-constant templates. Ideally we wouldn't be looping over all that unchanging data every time we render a web page or whatever.

Ropes

Ropes are trees representing immutable trees which originated in Cedar; you could describe the essential core of the idea in OCaml as follows:

```
type rope = Leaf of string | Cat of rope * rope
```

The idea is that `Leaf "foo"` represents the constant string `"foo"`, while Cat represents the concatenation of two ropes; `Cat (Leaf "foo", Leaf "bar")` is one possible representation of the immutable string `"foobar"`. This gives you constant-time string concatenation (if garbage collection is okay) and plenty of structure sharing, and you can convert the rope to a flat string in linear time when necessary — or just a `struct iovec` to send over the network with `writev`.

If the leaves are nonempty, this data structure has worst-case linear

space overhead, although it can be quite large, on the order of 64× the plain string.

If you augment this structure with lengths, you can additionally index and slice it in logarithmic time, if it's well balanced:

```
type rope = Leaf of int * string | Cat of int * rope * rope
```

If we define the function `rope_length`

```
let rope_length = function Leaf(a, _) -> a | Cat(a, _, _) -> a
```

we can state the invariant that `rope_length (Cat(a, g, d)) == rope_length g + rope_length d` (using "g" and "d" for *gauche* and *droit*) and maintain this with concatenation and lifting functions:

```
let leaf s = Leaf(String.length s, s)
let cat a b = Cat(rope_length a + rope_length b, a, b)
```

and define a function to drop the first *n* bytes:

```
let rec rope_drop n = function
 | Leaf(a, s) -> leaf (String.sub s n (a - n))
 | Cat(a, g, d) -> if n < rope_length g
     then cat (rope_drop n g) d
     else rope_drop (n - rope_length g) d
```

It is straightforward to define analogous functions to take the first *n* bytes, etc.

These functions will take logarithmic time and space if the tree is well balanced, but they can take linear time and space if the tree is imbalanced. Looking at the structure of `rope_drop` we can see that it's closely analogous to a binary-tree search where the search key in each `Cat(a, g, d)` node is `rope_length g`, though augmented by the past search keys. It's binary-searching the tree for the breakpoint.

It's also straightforward to write a function that converts a `rope` as defined above into a flat byte sequence; in OCaml, we invoke `Bytes.create` with the size of the string to be created, use `Bytes.blit_string` to copy each of the leaf nodes into the new sequence, and finally invoke `Bytes.to_string`; alternatively you can build up a string list and invoke `String.concat ""` on it, which does the same thing under the covers. This takes linear time and linear space regardless of the balance or imbalance of the tree, but it is necessary to be somewhat careful to avoid stack overflows.

My Lua implementation `macrope`

This implementation has four kinds of nodes rather than two — it additionally contains "variable nodes", representing template variables to be replaced, and "environment nodes", which provide values for those variables. This allows you to instantiate a template rope once and then use it repeatedly with different variable bindings without having to copy it around to modify it.

Because the size of the variable nodes can vary depending on their environment, the nodes don't know their size, so these ropes can't be sliced and indexed efficiently as the above OCaml code does.

The module exports a function `var` to define variable nodes and a

function `macrope` which idempotently coerces strings to macropes. Concatenation and parameter passing are done with the Lua `..` concatenation operator and the normal parameter-passing mechanism:

```
> macrope = require 'macrope'
> v = macrope.var 'name'
> s = v .. ' is the best friend of ' .. v
> = s { name='Bob' }
Bob is the best friend of Bob
```

Macropes can support large strings with linear, though poor, efficiency:

```
> x = macrope.macrope 'x'
> for i = 1, 25 do x = x .. x end
> =#tostring(x)
33554432
```

The first two lines execute instantaneously; the third line takes about 30–40 seconds on my laptop. No attempt is made to cache the results.

The $O(N^2)$ code above runs faster with `macrope` as follows, for N above about 10,000:

```
> s = macrope.macrope ''
> for i = 1, N do s = s .. 'x' end
> s = tostring(s)
```

The source code is organized as follows. `macrope` calls a `const` function if necessary, which is forward declared because I'm leery of Lua's scoping. Each node type has its own metatable:

```
# (in macrope.lua)
local catmeta, envmeta, varmeta, constmeta, const, macrope
```

All these metatables "inherit from" a common prototype metatable, but using a function that generates copies from it, rather than delegating to it using `__index`. They *do* share an `is_macrope` property via `__index`, which allows the `macrope` function to be idempotent.

```
local function meta()
   return {
      __index = {is_macrope = true},
```

The string concatenation operation is overridden to construct concatenation nodes:

```
      __concat = function(car, cdr)
         return setmetatable({car=macrope(car), cdr=macrope(cdr)}, catmeta)
      end,
```

The function-call operation is overridden as follows to create an environment node; note that each variable binding is coerced to a `macrope`:

```
      __call = function(self, vars)
         local nvars = {}
         for k, v in pairs(vars) do nvars[k] = macrope(v) end
         return setmetatable({vars=nvars, child=self}, envmeta)
      end,
```

Finally, coercion to a string as implemented by the standard `tostring` function (invoked implicitly by `print`) is done by using an explicit stack — because the worst cases I was alluding to above cause LuaJIT to kill the function if the stack gets more than a few tens of thousands of stack frames deep:

```
      __tostring = function(self)
         local items, stack, env = {}, {}, {}
         local function put(item) table.insert(items, item) end

         self:visit(put, stack, env)

         while #stack > 0 do
            local item = table.remove(stack)
            item(put, stack, env)
         end

         return table.concat(items)
      end,
   }
end
```

Mostly what remains are the `visit()` methods, which avoid recursion by pushing continuation closures on the explicit stack — which I managed to do in the wrong order at one point:

```
catmeta = meta()
function catmeta.__index.visit(self, put, stack, env)
   table.insert(stack, function(...) self.cdr:visit(...) end)
   return self.car:visit(put, stack, env)
end
```

The environment node needs to modify the environment, then arrange to restore it after its descendants finish executing:

```
envmeta = meta()
function envmeta.__index.visit(self, put, stack, env)
   local saved = {}
   for k, v in pairs(self.vars) do
      saved[k] = env[k]
      env[k] = v
   end

   table.insert(stack, function(put, stack, env)
                  for k in pairs(self.vars) do env[k] = saved[k] end
   end)
   return self.child:visit(put, stack, env)
end
```

Note that it's not safe to do `for k, v in pairs(saved)` because the saved value may have been a `nil`, in which case Lua would skip it in the iteration!

Variable nodes just delegate to their value (which, remember, was coerced to a `macrope` when the environment node was created), assuming the value exists:

```
varmeta = meta()
function varmeta.__index.visit(self, put, stack, env)
   local val = env[self.name]
   if val == nil then error("name not found: " .. self.name) end
   return val:visit(put, stack, env)
end
```

We need a function to export from the module to instantiate variables:

```
local function var(name)
   return setmetatable({name=name}, varmeta)
end
```

Constant nodes, the `Leaf` of the OCaml implementation above, simply invoke `put` to append their contents to the growing output buffer:

```
constmeta = meta()
function constmeta.__index.visit(self, put, stack, env)
   put(self.value)
end
```

There is a `const` constructor which could perhaps be inlined into the `macrope` coercion function:

```
const = function(value)
   return setmetatable({value=value}, constmeta)
end
```

Finally, the main entry point to the module does this type-testing DWIM magic:

```
macrope = function(thing)
   if type(thing) == 'number' then thing = tostring(thing) end
   if type(thing) == 'string' then return const(thing) end
   if thing.is_macrope then return thing end
   -- XXX maybe try to invoke tostring on it?
   error("not a macrope or string: " .. thing)
end
```

And the module exports:

```
return { macrope = macrope, var = var }
```

How worst-case ropes arise

One worst case is building up an extremely imbalanced tree of

single bytes:

```
macrope = require 'macrope'
N = 100000
s = macrope.macrope ''
for i = 1, N do s = s .. 'x' end
s = tostring(s)
```

As I said above, this takes 250 milliseconds, working out to 400 kilobytes per second. Although at this scale this is 12 times faster than the $O(N^2)$ native-Lua implementation, it's still ridiculously slow, and three times slower than when I wasn't using an explicit stack. For perspective, this takes about the same time, with 1000 times as many iterations:

```
macrope = require 'macrope'
function doit(N)
    s = 0
    for i = 1, N do s = s + i end
    s = tostring(s)
    return s
end
=doit(100*1000*1000)
```

Building the same string this way instead gets a further 5× speedup:

```
macrope = require 'macrope'
N = 10000
s = macrope.macrope ''
for i = 1, N do s = s .. 'xxxxxxxxxx' end
s = tostring(s)
```

Most of this speedup is in the explicit loop there, which took two thirds of the time before and now runs one tenth as many iterations.

This is the linear-search worst case that forced me to use an explicit stack in the `__tostring` function to avoid stack overflows, at the cost of a 2.5× slowdown in the `tostring` call. If you did some kind of tree balancing during the construction of the graph, it probably wouldn't speed it up (doing more work on each iteration would probably slow it down instead, even if the working set shrank) but it could speed the final tree traversal substantially.

A different kind of worst case is the other example above:

```
macrope = require 'macrope'
x = macrope.macrope 'x'
for i = 1, 25 do x = x .. x end
tostring(x)
```

This doesn't take long to construct, because there are only 26 nodes in the DAG, but in the `tostring` call the single leafnode is visited 33 million times; that's why it takes 30–40 seconds. Constructing the same string as follows instead takes 1.3 seconds, about 30 times faster:

```
macrope = require 'macrope'
```

```
x = macrope.macrope 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
for i = 1, 20 do x = x .. x end
#tostring(x)
```

So it wouldn't take a lot of tree optimization to speed things up by quite a lot in this case.

Leafnode coalescence, and when it isn't enough

The simplest measure would be to have a special case for concatenating short const leafnodes: if the total length of the result is under some threshold, somewhere in the range of 16–128 bytes, it's better to just copy all the bytes into a new const node instead of making a concatenation node. This would help a lot with the million-laughs DAG above but would only slightly worsen the problem of successive concatenation at the end.

You could be a little more sophisticated and get a linear improvement by having your concatenation operator coalesce with non-root leafnodes, something that's dramatically easier to expressin a language with pattern-matching (here ^ is OCaml's string concatenation operator):

```
let cat2 a b = match (a, b) with
| (Leaf(n1, s1), Leaf(n2, s2)) when n1+n2 < 128 ->
    leaf(s1 ^ s2)
| (Cat(_, x, Leaf(n1, s1)), Leaf(n2, s2)) when n1+n2 < 128 ->
    cat x (leaf(s1 ^ s2))
| (Leaf(n1, s1), Cat(_, Leaf(n2, s2), x)) when n1+n2 < 128 ->
    cat (leaf(s1 ^ s2)) x
| (_, _) ->
    cat a b
```

This successfully reduces the tree depth by a linear factor — 128 in this case — in the simple scenarios considered above. It might speed up or slow down the tree construction, though if it does slow it down, that's probably just because 128 is a bit too big. However, it doesn't help in all cases — consider the case of alternately adding to the beginning and the end of the string:

```
cat2 (cat2 (leaf "x") (cat2 (cat2 (leaf "x")
                             (leaf (String.make 128 'h')))
                      (leaf "x")))
     (leaf "x")
```

The initial `String.make 128 'h'` produces a large leafnode, and the following operations of appending or prepending a single character are then blocked from coalescing.

Evidently it would be useful to have a tree structure with rigorous guarantees on worst-case behavior.

B-tree ropes

B-trees are great for worst-case performance. The tree has a uniform depth on every path from the root to the branches, and the high branching factor minimizes the number of internal nodes on

which we must waste storage space and the amount of memory needed for tree traversal. And, at least in principle, they're simpler than other popular self-balancing trees such as red-black trees, AVL trees, and treaps. (They also tend to be much faster, especially on modern deep memory hierarchies.)

But can we use B-trees for ropes like the above? I started an OCaml implementation in 2015 and never finished it but I think that in principle it's straightforward. To concatenate, you may need to add tree levels to the smaller rope, and then you can merge (by concatenating) newly-adjacent nodes moving down from the root until you encounter a place where merging would make the new node too big; then you stop.

I still need to read Okasaki's masterwork and the follow-on work in the decades since, but there's a trap in amortized analysis of FP-persistent data structures — typically, amortized complexity analysis assumes that once you've done some big messy reorganization, like rehashing a hash table into a larger array of buckets, you can be sure that you won't need to do it again anytime soon. But with FP-persistent data structures (like ropes!) the state of the data structure immediately prior to the reorganization may still be accessible, and so it may be possible to provoke the reorganization over and over again by deriving new states from it.

This suggests that to get good amortized performance from FP-persistent data structures, either you need mutability behind the curtain or you need good *worst-case* performance per update operation. This is a connection I hadn't previously suspected between the world of FP-persistent algorithms and the world of bounded-time algorithms, which are usually on opposite ends of the universe.

B-trees in particular are relatively friendly to this. Suppose you decide on nodes of about 128 bytes: 64–256 bytes of text for leafnodes, 8–32 pointers for internal nodes†. The worst-case B-tree for a 4-gibibyte rope is $2^{26} = 67108864$ leaf nodes, which is at worst 9 levels of internal nodes. So, to concatenate it with another such rope, at worst you'd have to merge together 9 pairs of nodes, about 2 KiB of memory traffic. This is definitely worse than the 32 bytes or so of memory traffic used by `cat` or `__concat` above, by about a factor of 64, but it's also fairly closely bounded. Note that with a minimal branching factor of 8, the internal nodes are guaranteed to use no more than ⅐ of the leafnode memory.

For smaller strings the cost is smaller — with those parameters, everything up to 512 bytes is guaranteed to fit into a single level of B-tree.

For perspective, this suggests that the process of inserting a character (or arbitrary string) into the middle of an FP-persistent 4-gibibyte rope will require on the order of a microsecond and ten kilobytes of allocation:

- 18 new nodes, totaling 4 kilobytes in cache to break the tree into two slices at the insertion point;
- 9 new nodes, totaling 2 kilobytes in cache, to create the tree for the new byte;
- 9 new nodes, totaling 2 kilobytes in cache, to concatenate the new byte to the left tree fragment;
- 9 new nodes, totaling 2 kilobytes in cache, to concatenate the right

tree fragment onto that.

Filling up these 10 newly allocated kilobytes of memory is going to take a few thousand instructions, which takes about a microsecond on modern CPUs. You could probably reduce this cost in the average case with a simplified "buffer gap" approach in which you maintain separate left and right trees, so that you normally only pay the cost of creating the new byte's tree and concatenating it onto the left tree.

I feel like there may still be aspects of B-tree rebalancing I'm not appreciating, even without slicing.

† CLRS claims that allowing nodes to be less than ½ full, as in the factor of ¼ in this example configuration, makes it no longer really a B-tree, and if we don't allow nodes to be less than ⅔ full it becomes a "B*-tree". However, CLRS gets terminology wrong pretty often, so this might not be right. My rationale for the extra factor of 2 slack, which probably doesn't really apply in an FP-persistent context (at least not without more work), is to prevent pathological modification sequences from thrashing between splitting and joining the same node.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Python (p. 3671) (27 notes)
- OCaml (p. 3602) (8 notes)
- Lua (p. 3556) (5 notes)
- B trees

Interactive calculator

Kragen Javier Sitaker, 2018-04-26 (16 minutes)

I've written a keyboard-driven interactive calculator at http://canonical.org/~kragen/sw/dev3/rpn-edit, but it's really more of a prototype than anything else, and using it on a cellphone is painfully clumsy. But it's already pretty useful.

It interprets an RPN program to produce a sequence of formulas and their corresponding numerical results, displaying both the formulas and the results; the values can be vectors, and vectors are plotted as sequences of numbers.

So I've been thinking about how to take advantage of multitouch and small screens, and I've come up with some ideas that I think will be substantially easier than what I have now, even without a keyboard.

One of the problems with the current approach I'm taking is that what you're editing is really the RPN representation of the set of formulas. This is particularly tricky when you're rearranging a formula, because it's hard to tell if you're going in the direction you want; you have to go through a lot of disorienting intermediate states first.

Another problem is that it's pretty time-consuming to select existing sub-terms. A third is that DAGs aren't really supported, because the stack machine doesn't have variables or operations like DUP, OVER, or PICK.

So here are a few ideas that are generic to multitouch in general which I haven't seen tried much or at all:

• Quasimode buttons which, while held down with one finger (and possibly pushed in some direction), cause touches with the other finger to have some unusual effect. For example, a trashcan button which, while held, sends everything you touch into the trash; a break-apart button which, while held, splits subnodes out of formulas and turns them into formulas themselves at the top level; a push-up button which, while held, factors subexpressions out of function definitions, copying them into all callers; etc. In a photo manager, you might have a rotate-90° quasimode, or a quasimode to apply a given filter. Quasimode buttons have the advantage over pie menus (mentioned later) that you can drag your finger around the display until you have the right thing selected, taking advantage of visual feedback to allow you to accurately select deeply nested subexpressions. Quasimode buttons should visually highlight the objects they are applicable to while they are pressed; they can also display a transparent, non-interactible message that explains what they do. And of course some of the buttons can be non-displayed at any given time, using scrolling or tabs to bring the desired buttons into view. Foucault et al.'s "SPad" prototype uses this approach. In Surale, Matulic, and Vogel's 2017 paper, it's called "non-dominant hand" or "non-preferred hand", a term they took from Li et al.'s 2005 pen-based paper; they found it was the fastest of all mode-switching alternatives tested except for two-finger strokes, although it had a higher error rate when the user was standing; users

rated it as the easiest to learn and most accurate when sitting, though it suffered somewhat on some other measures.

• A single primary selection or focus, as with Macintosh-style GUIs, which can be set by tapping an object; but unlike Macintosh-style GUIs, the selection of an object results in action buttons popping out of the sides of other objects for binary operations that combine the two objects. For example, if you have an expression "5" sitting around and another expression "[11, 12]" gets selected, the "5" should grow buttons for +, -, ÷, ×, ↑, and maybe some other things. Originally I was thinking that you should just use two fingers to select two objects, and that might be worth trying, but then I realized that in cases like these, most of the time you want to select newly created objects to do more things with them, and it's probably easier if that happens automatically without having to locate the new object and move your finger to it.

• Another kind of quasimode that turns the whole area of the display into a two-dimensional input for the currently selected item until the quasimode button (or perhaps just the drag) is released. For example, this could allow using the whole display to spin through enumerated alternatives, rather than the half of the screen that iOS Safari uses for `<select>` elements. This technique applies also to the sliders and dials mentioned later. Some FPS games may use this for aiming; Bret Victor's animation demos use it.

• To take advantage of multitouch's higher precision in motion direction than initial touch position, like the typical lower-left-hand corner movement pad in video games which centers on your initial tap, we can use pie menus — while your finger is on an object (perhaps with no quasimode active, or perhaps with a special menu quasimode active), a pie menu pops up around it to offer applicable unary operations. In the calculator, this is probably better for top-level formulas than for subexpressions, because selecting a subexpression could be finicky. The Banovic et al. multitouch variation of pie menus uses a second finger to select the menu item, which seems like it should work a lot better actually, since it's the distance and angle between the two fingers that determines the command. Gupta & McGuffin's 2016 pie menu paper is similar to the pin-and-cross method mentioned in #10, but by using two separate fingers to open the menu (selecting the object) and to select the menu items, it has two touchpoints to use to control the resulting operation once it's begun, allowing simultaneous rotation, scaling, and translation. Bitwig Studio's "radial gesture menu" uses motion to select one of a small number of "inner ring" menu items or a second finger touch to select items from an "outer ring".

• Operation previews, providing one step of lookahead: the arithmetic operation buttons sticking out of the "5" in the above example should show the results — "+ [16, 17]", "↑ [48828125, 244140625]", "× [55, 60]", and so on. Once the operation is selected, a new object is created, and it becomes the selected object. In the calculator, though maybe not in other contexts, the objects that went into it disappear (you can break it back apart if you want).

• In addition to the usual keyboard entry of numbers, entering univariate functions by drawing graphs of them with your finger is a very useful feature. You probably want some kind of weighted average of recent and nearby finger strokes in order to not introduce

discontinuities unintentionally and in order to allow changes that are small compared to your finger positioning precision. For discrete sequences of numbers, lollipop charts may be a reasonable alternative.

• Each object has multiple possible views: in the case of the calculator, for example, as a formula, as a sequence of computed numeric values, as a graph, and so on. Some past experimental user interfaces have used "lenses" that could be dragged over objects in order to provide these different views, but I think that's probably kind of gimmicky and clumsy, except for cases where you actually want to scrub a boundary back and forth over small parts of an image; it's better to just display all the views for the currently selected object, when there is one, and allow the user to "pin" views they want to keep visible. Video games actually do provide multiple views of the same data fairly often, but they usually aren't all interactive.
• A transparent overlay keyboard. An overlay keyboard for typing is an honest-to-goodness mode, rather than a quasimode, but it's probably necessary on current cellphones due to their small screens. But there's no reason it has to obscure your view of the workspace. (GTA 3 has a slightly transparent keyboard.) For the calculator application, you probably want to be able to enter several numbers in a row without stopping. Given sufficient scrollability, the always-visible quasimode buttons could be transparent as well, like most buttons and most status displays in most 3-D cellphone games, so that they effectively take up less screen real estate.
• Sliders and dials: in addition to entering numbers with the keyboard, we want to be able to adjust them interactively with a movement, as in Bret Victor's work. So two of the views available on any number literal are as a linear slider and as a logarithmic dial with one decade per rotation, plus a button in the middle for negation. These two views are quasimodal in the sense that while you press on them, the entire screen can be used for the interaction. The logarithmic dial in particular allows access to a large range of values without having to define the range ahead of time. Making these quasimodal allows us to use multiple strokes with the adjusting finger across the display. On the other hand, non-quasimodal sliders would allow us to use several fingers at once to control several different quantities.
• The fidget-spinner-like pin-and-cross interaction technique presented by Yuexing Luo and Daniel Vogel is maybe a better alternative to pie menus; it preserves the high spatial resolution of being able to drag your finger around until you find the right thing with the screen real-estate advantages of pie menus. The disadvantage is that (it looks to me like) really only angles almost directly to the left and right of the target object are really usable, giving you four commands, although they went a bit further and used a few different angles on each side, including one at 45° above the X-axis which I have to think would be super uncomfortable if you were left-handed. Distinguishing between different distances seems like it would multiply the number of alternatives. Gupta & McGuffin's approach mentioned above might allow further degrees of freedom, and could be used one-handed within limited angles.
• Focused-item zooming: the currently-selected item should be displayed larger, maybe by an areal factor of 2–4, so that you can

normally display things at a slightly uncomfortably small size, and then make them amply large when they're selected. Some games do do this.
• Indirect input, like Pfeuffer et al.'s "gaze-touch" and "gaze shifting", either through a draggable lens (since gaze-tracking on cellphones is an unsolved problem), or through something like Käser, Agrawala, and Pauly's 2011 FingerGlass, which defines the area of interest using a pair of nearby fingers. These approaches dramatically improve effective display resolution and reduce the occlusion problem for drawing and selection precision.

Upon observing some people in the subway using their phones, I came up with some observations:

• By far the most frequent interaction types were vertical scrolling without taking advantage of momentum and typing a letter on the onscreen keyboard. These clustered; it was common to see 20 interactions in a row of the same type.
• The third most common interaction type was tapping buttons, list items, or menu items.
• The fourth most common interaction type was quasimodally recording an audio message in WhatsApp.
• Occasionally I saw horizontal scrolling. Once I saw a double-click to zoom. I saw no photo-taking, drawing, or pinch-zooming.
• Around 95% of the people were using their phones in portrait mode.

• Most people typed using two thumbs on the onscreen keyboard, even though the phone was in portrait mode; sometimes they even let go of the subway straps while standing to do this. They just bought phones big enough that this was comfortable for their hand size. The only person I saw using their phone in landscape mode was playing an immersive video game.
• Social interaction through cellphones, such as WhatsApp, Instagram, and Fecebutt, was much more popular than solo video gaming, such as Candy Crush.

I tentatively conclude that a new interface needs to be operable with two thumbs in portrait mode and not break single-finger vertical scrolling in order to be avoid large usability problems.

Some ideas specific to the calculator application

In a good calculator, there is some way to use a computed value more than once; the dataflow forms a DAG, not a tree. In the language of formulas, we do this with variables. I think there should be a quasimode button that allows you to factor out a subexpression into a variable so you can use it again. At one point, I thought that probably it would be best to do this implicitly: if you selected a subexpression as an operand, then that would pull out the subexpression into a variable and use it twice. But then I realized that it would be more intuitive to use that approach to edit the existing formula — same interaction, but sucking the standalone operand into the existing formula, replacing the selected subexpression with the newly created subexpression. This is similar to how function composition worked in early versions of Subtext.

Aside from adding complexity to existing subexpressions, we often want to simplify them or replace them altogether. I think simple replacement is likely to be frequent enough to warrant the use of drag-and-drop of a formula for it — which, if we make it mean "swap", can also function to swap arguments around and rearrange parts of an expression. Simplification can be done with the inverse of the above-mentioned combining operation — a user-interface action that removes an operator and expels one of its arguments to be an independent formula.

Given the ability to define variables like x = 2+3×4, a very incremental way to define functions is to select a subexpression and convert it into an argument; for example, if the 3, we would transform the definition into x(y) = 2+y×4, replacing all the existing references to x with x(3). This refactoring pulls the argument into the caller (every caller); if the whole definition of x is selected, it reduces x to the identity function, and we could as a special case remove x entirely, thus allowing this single extract-argument refactoring to also serve as inline-function.

The inverse refactoring is not always possible, since different arguments may be passed at different locations.

derivatives, integrals
optimization
vertical layout
reduction (?)
different views

• Temporal univariate function input and output: in addition to a static two-dimensional display of a sequence or continuous function, we can cycle through the values of the independent variable with time, as, for example, we do when it's an audio waveform we're playing back. But we can use this to choose a single numerical value to display at any given time, stepping a cursor visibly through the values; for an input variable, if augmented with a slider, we can use the slider to "push" the value up or down in a given region, or just to set it.

Topics

• Human–computer interaction (p. 3493) (76 notes)
• Multitouch (p. 3591) (12 notes)
• Calculators (p. 3362) (11 notes)
• Programming by example (p. 3655) (4 notes)
• Quasimodes (p. 3675) (2 notes)

Cheap shit ultrawideband

Kragen Javier Sitaker, 2013-05-17 (10 minutes)

So, pulse radio, huh?

If you want to do short-range wireless communication among really cheap circuits, maybe made out of garbage and running off batteries, you could in theory use something like CW oscillations at some arbitrary carrier frequency, like 101.3 MHz. A Q=100 fixed-frequency resonator circuit is relatively easy to build (three or four components), will accumulate RF energy over some 100 oscillations, and will allow you a modulation bandwidth of about 100 times less than the carrier frequency, in this example 1MHz. Then you could transmit pulses using, say, Manchester encoding, and do clock and data recovery in software. Your filter circuit gives you some 20dB of rejection of wideband noise like lightning and sparks and stuff.

This has a couple of big problems:

- 20dB isn't very much, so you have to use a lot of power.
- If you're using a single frequency, your signal will be easily picked up by accident by e.g. standard FM radios for listening to music. This will provoke complaints if you happen to be close to a standard radio station.
- By the same token, if you're near a strong source of RF noise at a given frequency, such as a radio station, it will make it impossible to receive data.
- Even though it's easy to build a fixed-frequency resonator with Q=100, it's hard to tune its frequency to 1% precision, for thermal and fabrication-error reasons. That means you'll have a hard time getting two such devices to communicate with each other without some kind of tuning component, like a varactor, unless you use something like a quartz crystal or a mechanical resonator for your filter.

Instead, you could build a low-Q resonator and transmit occasional pulses spread across a wide band. Here in ITU region 2, which includes the Americas, we have a 902–928 MHz ISM band (Q=35) commonly used for cordless phones, plus the 2.4–2.5 GHz ISM band (Q=25) also used by Wi-Fi and Bluetooth, both of which have reasonable penetration of building materials.

The lower your pulse rate, the less interference you'll cause with other radio applications. But now, instead of high-precision resonators, you need high-precision clocks. A common quartz clock crystal oscillator has a precision of below 300ppm (I think 100ppm is actually typical) but most of this error is due to temperature and voltage variation. If you can measure and compensate for the temperature and voltage, producing a "temperature-compensated crystal oscillator" ("TCXO") you can straightforwardly get precisions close to 1ppm. If you're counting cycles with a microcontroller, increasing or decreasing the cycle count is a simple way to compensate.

(You don't really need the resonator to make the communication work, but it might help to prevent your generated RF interference from straying outside the ISM band.)

If your average time between pulses is, say, 262144 clock cycles, then you can transmit up to some 18 bits per pulse. If your clock rate is 20MHz, easily achievable with off-the-shelf microcontrollers, you'll be transmitting an average of 80 pulses per second, working out to a maximum of 1.44 kilobits per second. The 262144 is limited by the precision of your clock speed; you need an oscillator with a precision of 4ppm, which will require temperature compensation, to get such a sparse pulse rate.

I think this is also the limit on your receiver selectivity --- those 18 bits have to be allocated between rejecting interference and transmitting information. You could get 50dB of receiver selectivity at the cost of only transmitting one bit per pulse; or you could get down to 3dB of receiver selectivity with the benefit of transmitting 17 bits per pulse.

I think it's reasonable to assume temperature compensation for the duration of a communication; the transmitter can transmit a series of pulses that allow the receiver to estimate the difference between their two clocks. A reasonable initiating pulse pattern for this might include a series of exponentially-increasing intervals: first 2 clock cycles, then 3, then 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, and so on, each pulse allowing the receiver to more precisely calibrate their expectations for the timing of the next pulse. You don't even need to start at 2 cycles. You could start at whatever you think the worst-case clock-rate skew is likely to be. Suppose you're reasonably sure your clocks are accurate to within 1000ppm, for example; then you could start with 512 cycles between pulses and go up from there.

Because of the need for coding-gain receiver selectivity, I think you probably need to transmit many more than 80 pulses per second to get a reasonable data rate. For example, if you transmit an average of 1024 pulses per second, a 16.777MHz clock gives you some 14 bits per pulse. If you allocate 11 of these bits to receiver selectivity and use the other 3 to encode information, you get three kilobits per second and 33dB receiver selectivity. Your signal could in theory also be rejected by some 30dB by someone using the same frequency band with frequency-division multiplexing, but this will be limited by their receiver Q. If you're talking about a regular FM radio, I don't think you're going to hit 30dB.

In a two-way communication, you could negotiate this allocation dynamically, so that you get higher data rates when there's less interference, in this case from fractional kilobits up to tens of kilobits.

To improve rejection of narrowband interference, your receiver (but not your transmitter) could be several concurrent channels handled by different analog filters with slightly higher Q. For example, you could divide the 902–928MHz band (Q=35) into four bands of some 6MHz, each handled by a separate Q=140 analog resonator. Any single source of narrowband interference will be limited to only one or two of these four bands, and you can simply ignore that band when decoding.

You may even be able to get some of your frequency selectivity from your antenna. A 900MHz quarter-wave antenna is some 8.3 cm in length. Unfortunately, I don't know how much selectivity you can get out of an antenna. Q=2, maybe?

Such cheap-shit electronics might often need to maintain very low

energy usage with extremely intermittent communication, for example in mobile-sensor-network applications. Achieving this probably requires maintaining a very low duty cycle for the receiver circuit, which effectively means you need some kind of time-domain multiplexing, with each device only listening for transmissions to it inside of a narrow time window.

Complicating this picture is the problem that the individual devices may be radio-isolated from each other for long periods of time and have clocks of limited accuracy, or even occasionally run out of power. If you want them to still be able to communicate when they come in contact, one of them needs to happen to transmit at a time when the other happens to have its receiver turned on.

A reasonable duty cycle for the receiver might be 1/1000 or so, if the receiver needs 1000 times the idle power of the rest of the device.

If you can depend on clocks continuing to operate and on occasional contact, then you can just use large timeslots. For example, if your worst-case clock drift is 15 seconds in a month, which is around what you'd get from a watch crystal, and your worst-case delay between resynchronization is a month, then each device can listen in a designated 30-second interval. With a duty cycle of 1/1000, the total cycle will be about 8 hours. So evidently this approach (which Elaine Chao told me about; I don't know if she invented it) works fine for wireless sensor networks that are relatively static, but it won't work well in the dynamic-topology case, unless 8 hours is "dynamic".

Elaine's solution was that when a new device is joining an existing network, it keeps its receiver turned on all the time until it receives a beacon from an existing device, which lets it synchronize with the network's timeslots. This could work well if you have mobile nodes with much higher available energy: they can listen constantly for beacons from fixed low-power nodes announcing their listening timeslots.

A different approach, suitable for a more symmetric system, is to transmit beacons and listen at random. Suppose we want a 10-second average-case synchronization lag; then we need to arrange for the transmitting node to transmit at the same time that the receiving node is receiving, on average, every 10 seconds. Suppose that the fastest that you could notice that you might have received a beacon, and therefore ought to keep listening to see if it's a new node you can talk to, is 1 microsecond. Now you have to coincidentally synchronize with a probability of about 1 microsecond in 10 million microseconds. If you listen a random 1/3000 of the time and the other node transmits a beacon another random 1/3000 of the time, then once every 9 million microseconds, you will happen to synchronize.

This implies, however, that you're transmitting 3333 beacons per second, which is a lot, since the entire beacon frame will take a lot more than a microsecond. If you're willing to accept a longer synchronization time, like 1000 seconds, then you can transmit only 333 beacons per second and listen 333 times per second.

Topics

Ostinatto

Kragen Javier Sitaker, 2014-04-24 (4 minutes)

I'm in the lobby of Ostinatto hostel in Buenos Aires, where Stace has come to see if she can get a job. Eminem from 2000 is blasting on the stereo, the fridge is full of Quilmes and Corona beer and Speed Unlimited energy drink. Blue-LED Christmas lights festoon the railings.

I just took Stace to see her first milonga. She'd never seen people dancing tango in real life before. It was under the summer night sky and full moon in Plaza Dorrego, also known as Plaza Bethlem. Across the Plaza, Candombe Monserrat was finishing up their weekly ambulatory performance of candombe drum music, the blessing and curse of living in San Telmo, but we could still clearly hear the golden-age tango recordings booming from the loudspeakers. I'd been suggesting she take tango lessons; within about ten seconds of arriving at the milonga, she had decided: this is a thing she must learn.

The lobby of Ostinatto is some seven stories tall, with wire staircases crisscrossing in the air under the skylight, which is dark with the night. Stace is hoping to get a job here.

Later we cross over to Tanguera Hostel, where Beatrice and I lived for some weeks, one of a dozen different places in Buenos Aires I've called home. It's beautiful with its marble and tile floors and its elaborate wall carvings, but we stopped recommending that friends stay there when they had a problem a few years back with bedbugs.

I show Stace the fiberglass statues of cartoon characters that decorate San Telmo, and tell her the little I know of each one; but my Argentine comic strip knowledge is pretty limited, and even if it weren't, it's hard to evoke the spirit of a comic strip in words. The statues speak more eloquently than I manage to.

On the bus home, a group of Boca fans are singing and whistling about how River fans are faggots lacking a testicle, and how they are going to kill them. Stace is enjoying it, since that's the kind of thing she goes in for, in fact the kind of thing she likes to organize, but I'm not --- I'm reminded of the last time a group of people called me a faggot and threatened to kill me, which was the crackheads who robbed me on the train in November. And the whistling hurts my ears, but I'm afraid that putting in earplugs will single me out as a target. After about 10 minutes, I get off early and take a separate bus home, angering her, because she feels I am abandoning her; but at the same time, she was complaining I was harshing her buzz from the bouncy football hooligan song. The adrenaline has mostly gone down by the time the other bus arrives.

Earler today, I spent some time with my coworker trying to qualify some off-the-shelf software for a $work task, hoping it can help me to avoid reimplementing its supposed functionality from scratch. So far it's something like fifty times slower than the software we've written ourselves, although our software doesn't yet do as much. I'm not sure if I'm doing something stupidly wrong or if it's really that slow.

I spent most of Saturday sitting in Starbucks reading *The Grammar*

Of Graphics, a book about data graphics, an influential book highly recommended by a couple of different widely-used pieces of data-graphics software. I'm finding it slow going, in part because the abstractions they define are all slightly different from the related abstractions I'm used to. Time alone in the café — all afternoon until it closed at midnight — was helpful in keeping my focus on the book.

Tomorrow I have more off-the-shelf software to try out for the $work task. Or I could try spending some more time with this software.

Topics

Heckballs: a laser-cuttable MDF set of building blocks

Kragen Javier Sitaker, 2016-08-17 (updated 2016-08-30) (24 minutes)

In part since I'm living with a seven-year-old, I've been thinking about laser-cutting a construction set based on a design Matt Heck showed me at TechShop something like a decade ago. If I recall, he had some things laser-cut from Masonite. I can't find anything he's put online about it.

Although the original was laser-cut from I think high-density fiberboard, you could realize this design with any sheet-cutting process, including scrollsaw cutting, plasma cutting, oxy-fuel cutting, laser-cutting of plastic or metal, waterjet cutting, hot-wire cutting, wire EDM, or sheet-metal shearing.

The basic Heckballs unit is a square with rectangular slots cut halfway from its edge to its center, with the width of the slot being the same as the thickness of the square. By slotting squares into each other's slots (edge-lap joints), so that their edges match up with each other's centers, you can easily assemble a somewhat free-form 3-D lattice. 80 mm is a reasonable size for the squares; 1.5 mm is a reasonable thickness.

To be somewhat concrete about costs, in the most basic Heckballs form, you can cut an 810 mm × 450 mm sheet of 1.5 mm MDF into 10×5 = 50 squares using 10×5 cuts totaling 10·400 mm + 5·800 mm = 8 m of cutting; each slot is 20 mm + 1.5 mm + 20 mm = 41.5 mm, and there are 200 of them, for another 8.3 m of slot cutting, a total of 16.3 m. At 30 mm/s, this is 543 s ≈ 1 hour ÷ 6.6, which is probably about US$6 of laser cutting time, or about 12¢ per Heckballs square.

MDF costs AR$179 ≈ US$12 for 1830 mm × 2600 mm of 3 mm-thick MDF, which works out to about US$2.50/m², or AR$40 for 300 mm × 600 mm of 1.5 mm thick MDF, which works out to about US$14/m². So the cutting is probably the bulk of the expense, which suggests it might be worth trying to cut stacked MDF.

(Steel is supposedly around US$300 per tonne right now, which would be US$7.20/m² at 3 mm, which rather surprisingly suggests that ASTM A36 steel only costs about twice what MDF does per unit volume, despite having 210 GPa of Young's modulus, which is 50× MDF's stiffness, and 290 MPa yield stress, which is 16× MDF's strength.)

In this form, this has a few drawbacks:

- The lattice is entirely made of perpendicular and parallel planes, which is the worst possible configuration for rigidity. More liberty of form would allow much more efficient material use.
- There's no way to make a flat surface without stuff sticking out of it, like for a shelf, chair seat, table top, or foot.
- The slot width is unforgiving of imprecision; if the material is slightly thicker or thinner than spec, or if the slot is cut slightly narrower or wider, it's easy to get parts that won't fit together or don't stay together reliably.
- If you fill all the slots of a square, the other squares collide in the

middle.
• The tensile strength of the joint is low because it is entirely dependent on friction.
• The square corners of the slot create stress risers in the material, resulting in much lower resistance to cracking than is necessary.
• Covering a long distance demands a lot of material, because you can't make a "beam" of joined squares that's less than 80 mm wide in both dimensions.
• Covering a long distance creates a lot of angular slop, because there are a lot of serial degrees of freedom in all the slot joints.
• There's no way to make joints that are free to turn or slide, only rigid connections.

To provide tensile strength, the joints can be provided with tapered clips (i.e. snap joints). To eliminate stress risers, the internal corners of the slots can be replaced with rounded divots, or better for laser cutting, chamfered divots. The tapered clips should also allow making the slots a bit wider, eliminating the sensitivity to precision, but curving the slots slightly might also solve the problem without introducing more play. Curving the tapered clips could make it easier to fit a fingernail or whatever between them and what they're clipping onto so that you can unclip them.

To prevent the collisions at the centers, it probably makes sense to chop off the edges of the squares, so that the distance between the centers of two edge-lapped squares is still 40 mm, but the squares are only 77 mm wide.

To permit covering a longer distance, I think probably the right solution is to have more than one kind of unit. For example, a long beam with 20 mm slots only at its ends, could hold the centers of the two squares at its ends 320 mm apart instead of 80 mm apart. This gives it a total length of 320 mm (or 317 mm to avoid collisions).

If the beams are 8 mm wide, we can cut an 80 mm × 320 mm rectangle into either 4 squares (roughly) or 10 beams. We probably want to have more beams than squares, because even in a 2D square grid, you have twice as many beams as vertices. The 4 squares require 320 mm + 4·80 mm + 4·4·21.5 mm = 984 mm of cut (although adding clips might add another 300 mm or more to that). The 10 beams require 10·320 mm + 80 mm + 2·4·21.5 mm = 3452 mm of cutting, which suggests a certain cost disequilibrium, pressuring the scale to increase to avoid wasting expensive laser to conserve cheap MDF.

Solving the rigidity problem involves using a different lattice. The incremental change is to add diagonals: convert the squares into octagons with eight slots and add beams of $320\sqrt{2}$ mm ≈ 452.5 mm. Alternatively, you could try to switch to a sphere-close-packing lattice.

Once you have 8-mm-wide beams, you could enable rotation by cutting circular holes through the centers of some octagons or squares. Curves are much slower to cut on typical laser cutters, so you probably don't want to do this for all the octagons, just some of them. The diameter of the hole should probably be about 8.2 mm.

Making flat panels that can fit snugly onto the coplanar edges of some octagons shouldn't be very difficult, although they won't be watertight. You just cut slots for the edges of the octagons to peek through.

These beams will be somewhat prone to buckling, since their slenderness ratio 320:1.5 is over 200; although the column effective length factor K is probably 0.5, this is still plenty long enough to enable buckling. If you could strengthen the beam into a T-beam with another 8 mm beam, the problem would probably be solved.

The aforementioned pressure to increase the scale adds more pressure to do the T-beam thing, since at a larger linear scale, buckling becomes a bigger problem.

What would the face-centered-cubic close packing solution look like? The unit cell is a cuboctahedron; in different planes, the number of lines coming out of a vertex are 0, 2, 4, or 6, all evenly spaced. If you think of the face-centered cubic arrangement as being a stacking of square layers, the connections within a given layer are at 0°, 90°, 180°, and 270°, while the connections to the layer above or below are at 45°, 135°, 225°, and 315°. Unfortunately, both the layer above and the layer below are at exactly the same angle, so they can't both connect. Still, you could enable a double-layer close-packed truss with this arrangement by simply having octagon joints and extra unit-length beams whose end slots are angled at 45°. For the moment I think I'll give up on fcc and hcp.

So, what does the most basic Heckballs kit look like? Let's say I want to be able to make a cube with diagonals on most faces. Four vertices, four regular beams, and one long beam make a square (320mm side plus an extra 40mm stickout). But then there's no immediate way to extend the lattice in a perpendicular direction. By adding more joints onto the edge, we could get a perpendicular direction and thus a sort of cube, but now some of the diagonals are the wrong length.

So here's the part where Heckballs actually become balls. We eliminate rotational symmetry from the octagon; one of its slots now extends to its center, and the opposite one is eliminated. Two such octagons can be edge-lapped together to form something approximating a sphere with 12 available valences, 8 of which are among the 12 directions needed for the fcc crystalline structure. To make the cube with some diagonals, we can use the three valences along one edge of one octagon to make the two sides and diagonal of one square, and the valence in the center of the side of the other octagon to make the side of a perpendicular square. Two corners of the top square must be made in this way, with their axes coplanar with and at 45° to the square's sides. The other two corners could be made with their axes coplanar to the perpendicular square and perpendicular to *its* diagonal, thus enabling four of the six cube squares to have diagonals. The other two faces are not thus diagonalizable.

If it is our priority to have three *adjacent* square faces diagonalized, neither this structure nor any similar structure can solve that problem, because connecting to the ends of the axis is impossible. A possible solution is to make some beams integral with joints, enabling them to connect axially.

Another possibility is to avoid connecting *at* vertices and instead connect *around* vertices, forming a polygon or polyhedron enclosing the vertex. For example, beams could penetrate into one another at some angle near their ends. It's easy for me to imagine how this would work for 90° in a single plane around a vertex, but harder for

me to imagine other angles.

Oh, wait, it's not that hard. You can get an arbitrary angle with the plane of each beam rotated by some amount with respect to the plane of the angle; as long as the beam's plane is neither parallel nor perpendicular to the angle plane, it's workable (though more so if it's not nearly so). Hmm, I have to think about that some more.

Anyway, back to the cube. Here we have eight vertices, each of which needs two half-slitted octagonal joints, sixteen octagons in all, plus 12 short beams and 4 long beams. Let's suppose we scale up a little bit to respond to the cost-optimization thing to ⅛m (125 mm) for the octagon nominal width, and add another 4 long beams for extra flexibility, and cut down the beam length to half the octagon width. Now we have:

- 16 122 mm × 122 mm octagons
- 12 250 mm × 31¼ mm short beams
- 8 353.6 mm × 31¼ mm long beams

That all adds up to about 0.42 m², slightly more than a single 810 mm × 450 mm cut at Max58, even before worrying about packing. Maybe to get quicker feedback I should scale down by a factor of √2 to 88.4 mm as the basic unit? This is getting fiddly, I should probably just use 100 mm, and go back to 4× for the beams. Also I don't need to leave a whole 3 mm off at the end; I only need ¾ mm off each end, or 1.5 mm in total:

- 16 98½ mm × 98½ mm octagons
- 12 398½ mm × 25 mm short beams
- 8 564.2 mm × 25 mm long beams

That's 0.39 m², still slightly too big, argh.

All right, let's bite the bullet and go down to a basic unit of 62.5 mm:

- 16 61 mm × 61 mm octagons
- 12 248½ mm × 15 mm short beams
- 8 352.1 mm × 15 mm long beams

That's 0.147 square meters, so I should be able to nearly double it.

It's about 41% octagons, 31% short beams, and 28% long beams. We could probably do with a higher proportion of beams: maybe 24 octagons, 28 short beams, and 20 long beams.

(Actually the octagons could still be squares maybe, although the slot ends should still be radially symmetric.)

A thing to somewhat worry about with these small octagons is that the slot ends will be awfully close together, and especially close to the deepened slot. The distance from the center of the joint to the end of the slot is 31.25 mm, with another 29.75 mm to the edge of the octagon. The clip hole needs to be at least 1.5 mm further in, probably better 3 mm further in, and the clip hole and slot probably need to be 1.7 mm across, and the clip hole probably needs to be 3 mm long, so the innermost part of the clip hole will be (31.25 − 3 − 3) mm = 25.25 mm from the center. But the slots at 45° from the deepened slot will have the center of the innermost end of their clip hole only √½ 25.25 mm = 17.9 mm from the deepened slot, and the 0.85 mm diagonal reduction from the width of the clip hole reduces that by 0.6 mm further down to 17.3 mm. Actually that's the distance the

clip hole corners will be from one another, but the deepened slot will also have its own clip on one side.

What should the clip dimensions look like?

The maximum tensile strength we can expect is the compressive strength of the 1.5 mm × 1.5 mm maximum possible contact area, 10 MPa · (1.5 mm)² = 22.5 N, or 45 N when we take into account the clips on both sides, which is pretty wimpy. Ideally this thing would withstand my harshest pulls without breaking, and with my legs I can pull close to twice my body weight, so about 2000 N. So this is about 44 times weaker than it needs to be.

It's possible to get some improvement there with multiple clips in multiple holes, but there isn't room for 44 sequential holes. At 6 mm per hole, there's really only room for 3 at most. So this is probably unattainable. It does rather strongly suggest that 1.5 mm MDF is suboptimally thin and will result in assemblies that are far too easily broken. Indeed, probably even 3 mm is too thin. The next size up stocked by Max58 is 5.5 mm, giving us about 13× as much contact area.

So what if it's 5.5 mm thick? 10 MPa · (5.5 mm)² · 2 = 600 N, about a third of what we'd like it to withstand. So maybe a double hole in 5.5 mm and call it a day at 1200 N of compressive strength.

But for that double hole, we probably need 22 mm from the center to the inside of the slot, even if the hook holes are absolutely square. So we probably need to make our octagons bigger, more like 150 mm. 15 such octagons would occupy our entire 810 mm × 450 mm sporting area, so we don't have room for anything else on the first pass. Which is fine, because this is already taking me forever.

What about tensile strength? MDF's UTS is only 18 MPa, so to withstand the 600 N of tension on either of the double hooks, we need 6.1 mm of width on the 5.5-mm-thick clip at the first hook to keep it from simply breaking from tension. That's going to make it kind of tough to unclip by hand, I think.

(I have a new design to keep the clip head from tearing off due to a tension concentration on the hook side of the clip; it involves a bar on the other side sliding into a slot, preventing the clip from rotating. But the clip still needs to withstand the tensile stress.)

What about the force and energy needed to unclip the clip? Patrick Fenner tells us that if we don't taper,

> the force to deflect the tip by a set distance is $F = dEta^3/(4l^3)$, where d is deflection, E is the Young's modulus of the material, t is beam thickness and a is the depth of the beam.

In this case, d is at least 5.5 mm, E is 4 GPa, t is 5.5 mm, and a is 6.1 mm. So if l = 75 mm, which is kind of an absolute limit if the diameter of the thing is 150 mm, then we need 16 N, which is doable without tools. For a more reasonable value of 40 mm, we get 107 N, which is going to require a screwdriver or something.

We may be able to improve the situation by tapering the clip down to 3.1 mm at the second hook, which should still be enough to resist the tensile force there, starting from a thicker root.

The clips and clip holes shouldn't extend all the way to the center, to avoid colliding with the deepened slot or excessively weakening the overall joint, nor should they extend all the way to the edge, in order to keep the angle properly controlled. The clip beam can extend deeper than the slot, but not by too much.

I don't know, maybe I'm being too demanding of the tensile strength of a clip joint. Suppose that instead of designing for 1200 N we settle for 360 N, 90 N per hook, which we can get with 3 mm MDF instead of 5.5 mm, so we don't need 11 mm of depth per hook hole, only 6 mm; we can now get 180 N tensile strength out of <3.4 mm of beam. And maybe we go back down to truncated 100 mm squares for our octagons. If the hook holes end 12 mm after the end of the slot, and we extend the clip beam by those same 12 mm, then its total length is 24 mm. So $dEta^3/(4l^3) = 3\text{ mm} \cdot 4\text{ GPa} \cdot 3\text{ mm} \cdot (3.4\text{ mm})^3 / (4\ (24\text{ mm})^3) = 25$ N, which you can operate with your fingers. It should get even better with tapering.

(ETA: actually it's totally reasonable to have arbitrarily low unclipping force with arbitrarily high tensile strength. Among the things you can do are to fold up the spring serpentine-style or to cut lengthwise slits in it to make it less stiff, in the limit becoming like the pages of a book.)

Plasma-cutting clips like these in steel might be feasible, but they'd have to be much thinner, since steel's stiffness is 50× higher; about 3.7 times thinner to retain the same operating force they had in MDF. Plasma has tolerances in the 300-micron range, so if your steel clip's nominal thickness was 1.5 mm, it might come out as 1.2 mm (and have 51% of the desired operating force) or 1.8 mm (and have 173%). So the 920-micron clip width indicated ought to be doable, and its 290 MPa yield stress would give it a tensile strength of 800 N (or as low as 540 N if it came out thin). The steel's 0.13% elongation at yield is only a fourth the MDF's 0.45% or so elongation at break, so steel flexures have to be quite a bit thinner in order to work at all.

Anyway, back in MDF-land, the whole clip/slot perforation extends to 13 mm from the center, which is only 8 mm in the case of the corners of the 45° and 315° diagonal slots next to the deepened slot. I'm getting less worried about that, though.

Bending strength of the joint should be good. If you have octagon A and octagon B edge-lapped and B tries to rotate around its axis, it's pressing against A at almost A's center (like uh 2 1.5 mm $\sqrt{2} = 4.2$ mm from the center I think?) and has one of A's hooks in it at 25 mm + 12 mm = 37 mm out. It's not totally easy to predict how that's going to end up, but let's say we have a 15 mm lever arm from where the fulcrum ends up being and something like 30 mm² effective surface area; then the surface will start to be damaged at about four or five newton-meters. Hmm, that suggests that maybe the double clips are overkill?

Trying to squish the two octagons against each other may break them more easily. The surface only has a 1.5 mm lever arm at most to resist the 50 mm lever arm where you can apply torque.

The 400 mm × 400 mm area devoted to the 16 100 mm octagons in the starter kit leaves a spare 400 mm × 10 mm area along one side and another 450 mm × 410 mm area for beams. No long beams will fit that way! We could fit long beams if we have a 200 × 400 area with 8 octagons, and then 8 more octagons in a 400 × 200 area to the right, leaving a 610 × 250 area above it into which we pack 10 564.2 × 25 long beams, leaving a 45 × 250 area to its right, and a 200 × 200 area below, or something. So then one short beam can fit vertically on the right. Blegh.

Alternatively the second group of octagons could be 600 × 100,

cutting the total octagons to 14, and leaving space for plenty of short beams above it. Actually no none of this makes any sense. I should drag and drop stuff or some shit.

Topics

Erlang musings

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

Learning Erlang for the first time. A few notes --- mostly the language and especially the runtime system are lovely, so to keep these notes short, I haven't listed the many things I like about them.

The syntax for tuples and records is just too bad. Maybe the record syntax was unavoidable, but the tuple syntax is bad.

Selective receive seems like it could lead to deadlock.

It seems like client-server responses ought to be tagged with a request-ID, not the server's PID.

It's kind of bizarre that the ugly "registered process" mechanism, which introduces a mutable global namespace into an Erlang node, is built into the message-sending machinery, but you must build at user-level the machinery to include your PID in a request if you want a response and figuring out which of your pending responses corresponds to which request.

Distributed performance isn't particularly good. Ping-pong time between processes in a single OS process is on the order of 12-15us on my machine, but between processes in different OS processes, it's about 1000-3000us.

It seems like the system tries to hide a lot of the crucial issues related to distribution: fault-tolerance, tolerance for partitions, security. There's a magic "epmd" process ("Erlang portmapper daemon") that gets started on a node by the first guy who starts a distributed Erlang process, and then the other Erlang processes connect to it (on TCP port 4369 --- what, was 4269 taken? 4369 is 0x1111.) and use it to figure out how to talk to each other. Even if another user stars running Erlang processes later on on the same node, they will use this same "epmd" for their nameservice.

I used Wireshark to see what they're saying to each other. None of the conversations are encrypted, but they all use some ugly binary protocol that makes them a bit hard to read. (Not sure why they bother with a binary protocol if it's going to take on the order of a millisecond each way to do a simple RPC containing a timestamp anyway.) The "epmd" conversations are pretty concise, but the "RPC" conversation between the two Erlang processes is a bit more verbose.

Apparently there's some kind of shared-secret authentication involved as well...

It seems like it would be better to just layer on top of some other system for actually making, authenticating, and encrypting the connections in the first place --- ssh, named pipes in the filesystem, HTTPS, or whatever.

There also doesn't seem to be any documentation for the security properties of binary_to_term/1.

Topics

Rubber air conditioner

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

Stretching rubber or other hyperelastic material heats it; allowing it to relax cools it. If you have a set of concentric rubber tubes you can twist, with metal rings embedded at intervals in the tubes to keep them from collapsing, then the torsion should manifest as heat which could be carried away by a fluid passed through them; allowing the torsion to relax should cool the tubes, which can be used to cool a fluid passed through them at that time.

This allows the construction of a refrigerator whose refrigerant is solid and thus very safe. By passing the same air (or other fluid) through the tubes during successive cooling cycles, it should be possible to cool it to lower and lower temperatures, until whatever leakage is inherent in the system cancels it out.

The cooled, stretched rubber has lower tension than the stretched rubber did when warm, so the relaxation cycle returns less energy to the motor than was put into it originally, which is where the ability to reduce the entropy of the air comes from. Still, it's likely to be significant, so it may be desirable to use a flywheel or something in harmonic motion to recover that energy for the next stretching cycle.

Fatigue is potentially a major problem for this device, since the rubber needs to last through at least tens of thousands of stretch-and-release cycles to be useful at all, if not millions; and the degree to which the rubber can be stretched safely determines the achievable ΔT.

Topics

Camera flash extrapolation

Kragen Javier Sitaker, 2019-11-12 (6 minutes)

One hot summer night recently, I guided a tourist through the streets of Palermo, the chichi Buenos Aires neighborhood. She was enthralled by the colorful murals that cover the walls of many of the alleyways (*pasajes*) and stopped to take many photos. (Incredibly, during her entire trip, she didn't get even a single hand computer stolen, just a jacket and some colored pencils.)

Her hand computer camera's wimpy LED flash was not enough to illuminate the murals clearly; unfortunately she wasn't able to visit them during the day. It occurred to me that it should be possible with multiple coregistered camera frames to artificially amplify the brightness of the flash somewhat.

The scheme is as follows. First, we take some frames with the flash and other frames without it, and perhaps frames with different quantities of flash or different exposure times (and variation in "ISO" gain to compensate for the exposure time). Then, we coregister them to find the corresponding pixels. For each pixel, we compute a linear regression of its color in each frame against the amount of flash in that frame; this resolves the various frames into a frame of "biases" representing the color of the pixel under existing lighting, a frame of "slopes" representing the per-spectral-band reflectance of the pixel to the flash, and what we hypothesize is random noise.

This allows us to extrapolate what the image would look like if the flash had been brighter than it was in fact. This will amplify errors and measurement noise to some degree, but if we are amplifying the flash signal by less than an order of magnitude, the effect should be small enough not to overwhelm the quality improvement from the added light.

The flash may cause some pixels to saturate, and saturated or near-saturated pixels should either be excluded from the linear regression or given a very small regression weight. A potentially worse problem is that the extrapolated image may saturate pixels that were not saturated in the input images. This can be handled by doing the extrapolation with extra bits of precision or in floating point to handle the larger dynamic range, then translating these linear high-dynamic-range values to the final image in a way that preserves as much information as possible. As I understand it, ACES recommends using a sigmoid curve per color channel to imitate the gradual-saturation nonlinear behavior of film emulsions, thus avoiding the total loss of information in particularly dark or light areas, providing higher dynamic range in digital images without the spatial filtering we usually see in HDR images.

Pixels spatially closer to the camera will experience more flash illumination, at least if we're using an on-camera flash (typical for photos from hand computers). However, they will also experience more flash illumination if they have higher reflectance, so by itself this does not turn a camera with an on-camera flash into a depth camera; it cannot untangle reflectance from closeness to the camera. If the non-flash illumination were constant everywhere in the scene, like a naïve ambient light source in a raytracer without ambient

occlusion, then we could use the ambient light source to derive the raw reflectance, then divide the flash slope of each pixel by its reflectance to derive its (inverse square) distance. However, this seems unlikely to work in practice.

A different way to take advantage of this phenomenon that might work better is to use even fairly imprecise SfM depth data derived from camera parallax to figure out which parts of the image are unusually close to the camera, then artificially attenuate the flash on them. This give the effect of a flash positioned further from the scene than it really is, thus evening out the flash illumination across the scene somewhat.

If we have this crude SfM data available, we should also be able to determine that certain pixels are at very nearly the same depth — for example, nearby pixels on the surface of a single object — so they should be receiving very nearly the same illumination from the flash. This means that differences in their flash slopes reliably indicate differences in their reflectances; this enables local correction of the "bias" image for reflectance to get an image that is purely a map of non-flash ilumination (plus non-Lambertian phenomena such as specular glints). Since we're talking about pixels that are close together in three-space, the *intensity* of non-flash illumination on them should be the nearly the same, except in shadows; only its *direction* varies. So we can use this to get information about the angle between the surface and the non-flash illumination and thus about its geometry.

Of course, if we have multiple flashes in different positions available, we can use them to illuminate the scene from different angles in different frames, thus giving us lots of great information about surface geometry. This might sound like an expensive setup that is unrealistic in most circumstances, but in fact with the advent of widespread forward-facing cameras on hand computers, it's eminently feasible for small objects: the different "flashes" are just different regions of the screen. (Some researchers also noted that this allows you to get nine or twelve channels of spectral information rather than the usual three, since the spectra of the RGB or RGBW filters on the screen will usually imperfectly match the spectra of the RGB filters on the camera focal plane. Unfortunately, some shithead reporter mischaracterized the research as providing "hyperspectral" capabilities, and so most of the commentary I saw focused on debunking the lie rather than the actual research.)

Topics

Ideas to explore

Kragen Javier Sitaker, 2017-05-29 (updated 2019-09-15) (3 minutes)

• Portals: can I get some kind of usable windowing/hypertext system out of circular rotating scaling portals?
• Laser-cut omnitriangulation: can I use tusk tenons to make light, airy lattices out of cheap, thin laser-cut MDF?
• Reduced affine arithmetic raytracing: can I make a fast ray tracer with it?
• Reduced affine database: can I do a fast query on gobs of time-series data with it? (See An affine-arithmetic database index for rapid historical securities formula queries (p. 2275).)
• Circuit simulation: can I get SPICE working in a reasonable way with a Falstad-like UI?
• Database exploration: what does it look like to load my mail into Kafka/Samza, SQLite, Elasticsearch, LevelDB, Redis, Datomic, or just MySQL?
• Home kiln: can I get a reasonably efficient kiln working?
• Programmable logic: can I whip these 2114s and PALCE16V8s into some kind of useful logic circuit?
• Fabribot: can I get some kind of fabricating bot working? Clay cutting, 3-D printing, etc.
• Calculator rebraining: can I replace a calculator brain with an AVR running my software and get a super-calculator? (See Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717).)
• Self-sustaining personal computer: can I wedge a Nokia cellphone display into a PS/2 keyboard and SD card and get a self-hosting computer out of the deal, one that can reprogram itself?
• Moiré servomechanism: can I use moiré patterns between sparse random screens with webcams to get high-resolution feedback on where a thing is?
• Garbage oscilloscope: can I make a reasonable oscilloscope out of common garbage? (See files VCR oscilloscope (p. 213), TV oscilloscope (p. 1253), Laser printer oscilloscope (p. 449), and Disk oscilloscope (p. 713).)
• Digraph notation: can I come up with a much better way of describing finite state machines, electrical circuit schematics, dataflow graphs, and similar graphs? (See files Graph construction (p. 3226) and Circuit notation (p. 1161).)
• Card-based hypertext: what does it look like to re-envision a text as a set of linked "cards" small enough that several can be on the screen at once, even on a cellphone? Is this Smallest Federated Wiki?
• RPN UI: can I extend rpn-edit to be powerful enough to handle most everyday calculations, maybe better than Excel and Python? (See An RPN CPU instruction set doubling as user interface (p. 177) .)
• Synthgramelodia: can I repackage synthgramelodia into a form that makes it easy to run and shows what it's doing?
• Magic Kazoo: can I get a prototype Magic Kazoo running, maybe as an Android app? (See The Magic Kazoo: a synthesizer you stick in

your mouth (p. 1873).)
• Power supply hacking: can I make an adjustable benchtop power supply that won't catch on fire and displays the output voltage with an AVR on an LED display?
• Life clock: can I make an LED display of how many days someone likely has left to live?
• AVR programmer: can I make a serial programmer to load the Arduino bootloader into the various AVR chips I have lying around? Using ArduinoISP, say?
• Tiny bytecode machine: can I implement a tiny bytecode engine that allows me to do any further programming in a flexible way?
• Logarithm table memorization: can I practice mental logarithms enough to make them a useful calculation shortcut?
• Machine-learning/optimization stuff: what can I do in this area? Maybe I need someone else to suggest exercises. All kinds of physical and information design, plus control systems, seem like appropriate things to try.
• Tile grid systems: can I make a maze out of user-drawn tiles in DHTML?

Topics

Freeze distillation at 1 Hz

Kragen Javier Sitaker, 2016-10-06 (5 minutes)

How can you do process intensification of fractional crystallization cascades?

Freeze distillation or fractional crystallization is normally a slow process. But they still work even if the crystals are very small, so there's no need for them to be slow. You could carry them out at frequencies in the neighborhood of 1 Hz using something like a phase-change regenerator. The "coolant" of the "regenerator" is probably something innocuous like brine, while the phase-change material in the "regenerator" is the material you're actually trying to separate by partial crystallization, and it isn't quite stationary: you pump the liquid phase in a direction parallel to the direction of the coolant, but 90° out of phase with the pumping of the coolant.

That is,

• First, you pump the coolant from the cold reservoir to the hot reservoir, freezing some of the phase-change material. Then,
• you pump the liquid part of the phase-change material (the mother liquor) toward the cold reservoir (that is, in the same direction, but in a separate isolated circuit). Then,
• you pump the coolant from the hot reservoir back to the cold reservoir, melting some of the phase-change material. Then,
• you pump the mother liquor toward the hot reservoir. Then you start again.

This should result in concentrating the lowest-melting solution of the phase-change mixture toward the cold reservoir, and the higher-melting parts toward the warm reservoir, while more or less maintaining the temperature gradient constant. The temperature swing may be more than you would expect, because metastable zone width increases with stirring and cooling rate, and our stirring and cooling rate here is yuuge.

If at some point the phase-change material passages are completely blocked, you can pump warm coolant past it until that's no longer the case.

Separated phase-change material can be removed at the extremes of the apparatus, and unseparated material can be added to replace it in the middle, providing a quasi-continuous process.

This process should be sensitive enough to separate substances by even very slight differences in solubility, including difficult cases like separating erbium bromate from holmium bromate or separating heavy from normal water. It might even be capable of competing with ion-exchange chromatography. The regenerator-like configuration eliminates nearly all of the energy waste associated with traditional fractional crystallization cascades.

It's important for the piping to be narrow enough to prevent diffusion of the liquid phase-change mixture against the concentration gradient from becoming turbulent rather than viscous, and also to prevent the diffusion of heat against the temperature gradient, since narrower passages are longer for constant volume; additionally, narrower passages mean that heat diffuses between the

coolant and the phase-change material more rapidly, allowing higher frequencies. However, narrower passages require more energy applied both to pump the fluids and to restore the temperature difference between the reservoirs.

If both fluids are liquids, it may be desirable to carry out the entire process under high pressure or even to alternate between pressures (in addition to or instead of pumping coolant) to alter the equilibrium phases of the phase-change material. This may allow an escape from pernicious eutectics. Doing this with a gaseous coolant might be feasible but seems like it would be very difficult due to adiabatic heating and possible deformation of the apparatus.

Here are some possible materials, depending on the temperature range in which the material to be separated solidifies:

temperature range	coolant	piping
<-200°	LN_2	copper
		brass
		cryogenic stainless
-200°--100°	ethane (to -182°)	cryogenic stainless
	R-32 (to -136°)	brass
	R-22 (to -175°)	copper
	propane (to -187°)	
-100°--20°	ethanol (to -120°)	silicone
	propylene glycol	copper
	(to -59°)	brass
	R-134a	low-temperature stainless
	SF_6 (to -50°)	
	R-22 (below -40°)	
-20°-0°	brine	polyethylene
	ethanol	polyethylene terephthalate
	propylene glycol	silicone
		copper
		stainless
		steel
		brass
		glass
		aluminum
		PTFE
0°-200°	water	polyethylene
	mineral oil	PET
	propylene glycol	glass
	(to 188°)	silicone
	glycerol	polyimide
	ethanol	copper
	silicone oil	stainless
		steel
		brass
		bronze
		aluminum

		PTFE
200°-500°	molten nitrate salts	borosilicate glass
	fluorocarbons	stainless
	lead-tin eutectic	polyimide
	type metal (Sn/Pb/Sb)	copper
	FLiNaK	brass
	FLiBe	bronze
	NaK, Na, PbSb	
	tin	aluminum
	air	steel
	CO_2	
	glycerol (to 290°)	
	steam	
	silicone oil	
500°-1000°	molten nitrate salts	fused quartz
	Sn/Pb, Sn/Pb/Sb, Sn	stainless
	FLiNaK, FLiBe	superalloys
	air	fluorination may be desirable
	noble gases	noble metals
	CO_2	
	nitrogen	
1000°-1200°	Al, Pb, Li	stainless
	CO_2	superalloys
	nitrogen	noble metals
	noble gases	fused quartz

(All of the above is kind of a guess, not deep materials knowledge.)

Presumably the only temperature limit on the applicability of the process is being able to find piping materials that melt hotter than the materials you're trying to separate and that won't significantly dissolve in or react with the materials you're separating at their melting point. (You don't want total nonreactivity, though, because you need the crystals to nucleate, ideally on the walls.) I just don't know what to propose above 1200°.

Topics

- Materials (p. 3560) (112 notes)
- Chemistry (p. 3373) (20 notes)
- Process intensification (p. 3653) (6 notes)
- Regenerators (p. 3679) (4 notes)

Laser ablation of zinc or pewter for printed circuit boards

Kragen Javier Sitaker, 2016-09-19 (4 minutes)

The typical way to make printed circuit boards is by buying copper-clad glass-reinforced epoxy circuit boards and etching away part of the copper — either with ferric chloride, with air-regenerated cupric chloride, or with electrolytic etching. Occasionally people will instead use end-mills in milling machines instead, which inevitably cuts into the highly-abrasive GRP board and spreads glass dust. Typically the copper is around 25 microns thick. But what if we could laser-cut the metal instead?

Well, copper is hard to laser-cut. It heats at 24.4 J/mol/°, boils at 2562°, sucks up 300kJ/mol, and from yellow well into the infrared, it reflects more than 90% of incoming light. (It only reflects about 60% of blue light, giving it its characteristic red or orange color.) But all the kinds of blue and ultraviolet lasers I know about are a real pain in the ass. So you're just about stuck with an extra factor of ten or so in the already-gigantic energy. (Maybe you could avoid the extra factor of ten by pre-oxidizing the surface.)

(Calculating: 2540° · 24.4 J/mol/° + 300kJ/mol ≈ 360kJ/mol; multiply by another factor of ten or so and you need 3.6MJ/mol for copper. At an atomic weight of 63.5 that's 5.7kJ/g; at a density of 9.0g/cc it's 51.3 J/cc. I'm leaving out heat of fusion because it's quite small.)

Zinc might seem like a more reasonable metal to laser-cut. It weighs 65.4 g/mol, boils at 907°, heats at 25.5 J/mol/°, weighs 7.1 g/cc, and sucks up 115 kJ/mol to boil, and typically has about 80% reflectance across the VNIR spectrum, with an inconvenient peak in the red, but a very convenient dip well below 70% in the 1μ range. Its conductivity is about a fourth of copper's, at 59 nΩm to copper's 17 nΩm, which you'd have to compensate for by making it four times as thick. So you need 885° · 25.5 J/mol/° + 115 kJ/mol ≈ 138 kJ/mol to ablate it, and if you get ⅓ laser efficiency, you need 0.4MJ/mol, an order of magnitude less ablation energy than copper. But then you lose most of that again with the factor of four increased thickness, making laser energy ablate only about two or three times as much circuit board area per joule.

(Or, if your laser pulses are longer time scales, maybe more, because copper conducts heat at 400 W/m/° while zinc only conducts it at 120 W/m/°. But I'm assuming the laser pulses are short enough that this isn't a consideration.)

Zinc has a major disadvantage for circuit boards: it tends to form zinc whiskers. It may be possible to fix this by alloying it with some other metal, such as copper (forming brass) or tin (forming a sort of pewter).

I don't know if it's possible to reduce the boiling point of metals by forming positive-azeotrope alloys from them.

Other candidate metals might include magnesium (1090°), bismuth (1560°), manganese (1962°), indium (2000°), and tin (2270°). Ytterbium (1466°) and thallium (1473°) are too unstable and toxic,

thulium is too expensive, and aluminum doesn't boil until 2470° and is particularly highly reflective.

Other candidate methods for thermal ablation of metal coatings include arc heating, electron-beam heating, and ion-beam heating.

Topics

Affine arithmetic optimization

Kragen Javier Sitaker, 2017-07-19 (updated 2019-09-15) (3 minutes)

Reduced affine arithmetic is an extension of interval arithmetic; it is a nonstandard semantics method for evaluating a computable function in the form $f(x_0, x_1, x_2 \ldots x_n)$ over some multidimensional interval $x_0 \in (x_{00}, x_{01})$, $x_1 \in (x_{10}, x_{11})$, … $x_n \in (x_{n0}, x_{n1})$ to get a conservative affine approximation $k_0x_0 + k_1x_1 + \ldots k_nx_n \pm k_{n+1}$ within which f is guaranteed to be contained; for arithmetic expressions, it normally takes a factor of n longer than simply evaluating the expression at a point, and for regular functions, in the limit, the error term k_{n+1} diminishes quadratically with the interval size.

(Can we apply RAA recursively once to get a reduced affine approximation of this error term, thus telling us which independent variables are not contributing any significant approximation error, and perhaps also getting a cheaper second-order approximation than the full $O(N^2)$ representation?)

Mathematical optimization is the problem of finding the minimum of a "cost function" of some "design variables" within some given "feasible region". Usually this can be restricted to the problem of finding the function's global minimum, because we can modify the function to guarantee its value outside the feasible region will be very large.

For perfectly linear functions, reduced affine arithmetic computes a perfect approximation; it is only nonlinear operations such as multiplication, division, or conditionals that contribute error.

(Can you apply RAA in other fields, such as $GF(2)^n$ with XOR, NOT, and either AND or OR? XXX $GF(2)^n$ isn't a field, dude)

Here is an algorithm for using reduced affine arithmetic to find the minimum of a function over some domain.

Begin with a single interval covering the entire feasible region, perhaps $(-\infty, +\infty)$ on every independent variable. Compute the function using reduced affine arithmetic over that interval. Store it in a min-heap of (interval, result) pairs indexed by least lower bound (i.e. the lowest value the function can possibly achieve on that interval).

At each step, select the interval with the smallest least lower bound. Remove it from the min-heap and subdivide it in some way, for example into three subintervals along a randomly selected axis. Compute the function for each subinterval, and insert the new subintervals into the min-heap.

At any step, (one end of) the best interval on the heap is in some sense our best guess at the true global minimum of the function, but any other interval whose least lower bound overlaps its least upper bound may actually be better. However, once the difference between the upper and lower bounds is within our desired tolerance, any one of those intervals is an acceptable answer.

Note that, although it is not a gradient-descent optimization algorithm, it is still an anytime algorithm.

Topics

Internal determinism

Kragen Javier Sitaker, 2016-08-17 (2 minutes)

Guy Blelloch and others wrote a paper in 2012, "Internally Deterministic Parallel Algorithms Can Be Fast".

Their basic theory is that fork-join nested parallelism with no communication between concurrently running threads allows you to program efficient parallel algorithms relatively easily without introducing nondeterminism.

Or wait, maybe their threads have "shared state" and consequently depend on "non-trivial commutative operations". If that's true I don't see how the stated properties of the dependency graph can be correct. I guess those commutative operations on shared state don't enter into the dependency graph except as a set.

So for example they have an AtomicAdd operation to add a value to a shared variable, but it can't return a value without violating internal determinism.

They support four different kinds of memory objects supporting commuting operations:

• memory cells, supporting the "priority write" operation x.pwrite(v), which updates x to have the maximum of its old value and v, and a read operation that returns its current value.
• reservables, supporting the operations x.reserve(p), x.check(p), and x.checkR(p) check-and-release. Check and check-and-release commute, I think? Not entirely sure. checkR sets the priority of the reservable back to ⊥ and returns TRUE if it was currently reserved with the specified priority. None of the checks commute with reserve(), but I think reserve() commutes with itself.
• dicts, supporting d.insert(x) and d.elements(), which returns an element for each key that has been inserted, eliminating duplicate keys. Both reads and writes commute among themselves but not with each other. Elements contain their keys within themselves. A user-specified priority resolves duplicates, and linear probing evicts elements with lower-priority keys and moves them further down to ensure that the final iteration order is not dependent on insertion order.
• disjoint sets, for spanning forests; f.find(x) returns the set identifier containing x, and f.link(s, x) merges the sets of s and x. The criteria for commutativity are somewhat tricky.

Topics

Thermodynamic systems in housing

Kragen Javier Sitaker, 2016-06-28 (24 minutes)

Much of the business of a house comes down to controlling the flow of a few crucial commodities:

- Heat;
- Light;
- Air;
- Water, whether as humidity or liquid;
- Noise.

(This is also true, incidentally, of gardening, of which more later.)

This is a Fullerian view of a house as a dwelling-machine, rather than a structural engineering effort to resist its own weight. It's also a sort of instrumentalist worldview: the house is designed for CONTROL, so that its owners can use that control to experience comfort.

In its crudest form, a house is just an enclosure to restrict the flow of all four of these commodities. The roof and walls stop the rain from soaking the space beneath, the sunlight from heating it, the wind from casting it into disarray, the cold night from chilling it, and the inhabitants from scaring the animals with their gasps and ululations. In nature, all five of these elements generally arrive together and depart together, and to exclude one is to exclude another.

But too much restriction is usually bad, even fatal. If heat cannot escape, an inhabited dwelling will eventually cook its inhabitants until they cease to heat it with their body heat; if light cannot enter, the inhabitants are blinded; if air cannot enter, they suffocate; if water cannot exit, everything becomes waterlogged with their breath; and if noise cannot enter, well, perhaps nothing bad happens.

More elaborate contrivances allow us to separate these five elements to some extent, reducing the necessity to compromise our needs between them:

- Solid windows, whether of mica, of glass, or of polyethylene film, damp noise but permit light to pass without bringing along air and water, or letting quite so much heat pass out the other way when it's cold out.
- Shutters allow us to further restrict the air, light, and noise that pass through windows.
- Curtains, too, are mostly used to stop the flow of light through windows; they can also be used for privacy, to stop the flow of heat through walls, and to absorb noise.
- Thermal mass, like adobe walls, damps thermal fluctuations like the big capacitors in your power supply. That same mass often serves to absorb noise and damp fluctuations in humidity.
- Furnaces, heaters, stoves, and microwaves produce heat with little light, but the flame-powered kinds require access to vent their exhaust.
- Luminaires provide a small amount of light upon demand, even at

night, along with correspondingly little heat.
• Air conditioners and other kinds of heat pumps move heat in the direction of our choosing without bringing air and light along for the ride, though often at the cost of noise.
• Plumbing provides water on demand, often leavened with a precise degree of heat for luxury; like heat pumps, it, too, can be used to control the flow of heat, which will probably be its primary function within a few decades, as machines' energy usage explodes.
• Rugs, historically, served largely to insulate us from chilly floors and absorb sound. (Since the 1930s, they have a secondary use of reducing the cost of housing by hiding plywood flooring which would otherwise have to be covered with more expensive hardwood.)
• Construction materials are often chosen to be porous to prevent condensation inside walls: if humidity can get into the wall from the warm side, the thermal gradient through the wall must be matched by a humidity gradient, or when it's cold outside, the thermal gradient will cross the dewpoint.
• Insulation in walls stops heat transfer without affecting airflow significantly.
• Weatherstripping around doors and windows stops heat transfer, noise, and airflow.

But there are a variety of other techniques, less widely used, some even entirely speculative, which promise to offer better tradeoffs. Some of these are proven but not widely known.

• Solar overhangs over windows permit sunlight to enter the windows during the winter but not the summer, passively reducing the temperature variation throughout the year.
• Electrically controlled shutters or mirrors could provide the same service, but controlled by a PID or better negative feedback system, rather than crudely by the season.
• Countercurrent air heat exchangers allow air to enter and leave a house without taking heat and humidity with it. These are necessary for Passivhaus structures, which are designed to limit heat flow to a much lower level than traditional, in the interest of reducing marketed energy consumption.
• Speculatively, regenerative heat exchangers are another way to achieve the same goal in less space, and possibly with less expensive materials. While a countercurrent heat exchanger runs currents in both directions at once, separated by a high-thermal-conductivity barrier (for example, warm air exiting the house in winter might be used to thus warm up cool air being brought into the house at the same time, by passing the air currents through parallel pipes separated by an aluminum wall with lots of fins), a regenerator instead alternates between the two directions. So a regenerator could consist of, for example, an insulated tube full of sand, gravel, or a microencapsulated phase-change material (see below). This, however, requires the air volume inside the house to vary, which is potentially difficult to achieve. With a little water, it's possible that the regenerator could simultaneously function as a filter, although of course water evaporation would also humidify and cool the air.
• Geomembranes could permit subterranean construction with much less compromise. One great difficulty of subterranean construction is that all traditional construction materials are porous, so below the

water table, a constant trickle of water enters in unpredictable places. It must be pumped out to avoid flooding, but it still compromises humidity control; I have known subterranean offices with deadly Stachybotrys infestations. Geomembranes are sheets of tough plastic welded together, developed for keeping landfills from leaking; they could conceivably solve this problem, although I'm not familiar with examples of their use in building construction.

• Speculatively, halite (sodium chloride, table salt) is a mineral that flows under pressure that is relatively light in geological terms. Perhaps halite could serve as a sort of self-healing geomembrane to seal out water, although over long periods of time it will eventually diffuse away in the groundwater, and it would rule out the use of steel-reinforced concrete construction.

• Light pipes (or "light tubes") are waveguides for transporting light long distances, up to tens of meters, through small apertures into a building. In essence, they are fat fiber optics. They can exclude the infrared wavelengths that carry a substantial fraction of sunlight's heat, and they can relieve the murky dimness that often plagues subterranean construction.

• Subterranean construction is superb for noise control, thermal mass, insulation, and security; military buildings have often been constructed underground for centuries for these reasons.

• Alkali scrubbers, like those used in submarines and space stations, remove carbon dioxide from air, reducing the amount of airflow needed to keep it breathable. (When you suffocate, the level of carbon dioxide reaches fatal levels long before the level of oxygen falls dangerously.) In their simplest form, these are just curtains impregnated with lithium hydroxide or a similar alkali, which convert to carbonates upon absorbing carbon dioxide. Lithium hydroxide, though preferred in submarines and space applications for its light weight, requires heating the carbonate to 1300° to regenerate the hydroxide; the carbonates of sodium, calcium, and magnesium are more manageable, at 851°, 550° to 825°, and 350°, respectively, and they are much less caustic as well.

• Phase-change materials like ice, Glauber's salt, or paraffin, provide effective thermal masses that are many times larger than their physical mass. A single ton of ice, with its enthalpy of fusion of 333 kJ/kg, can absorb the same amount of heat in its melting as 44 tons of feldspar rock, with its specific heat of 0.75 kJ/kg/K, over the range 20° to 30°; ice has an inconveniently low melting point, but you can use ice, rather than air, as a reservoir for a heat pump, getting a major efficiency boost, and other phase-change materials are capable of honorable performance at higher temperatures. Paraffins can be fractionated to have precisely calibrated melting points at any desired temperature, and Glauber's salt melts at 32°, just slightly too warm for comfort, absorbing 252 kJ/kg; at 1.46 g/cc, that's 368 kJ/ℓ. Wikipedia says, "For cooling applications, a mixture with common sodium chloride salt (NaCl) lowers the melting point to 18°C (64°F). The heat of fusion of NaCl·Na2SO4·10H2O, is actually increased slightly to 286 kJ/kg." As a useful ballpark, a square meter of sunlight inside your house during 12 hours deposits about 43 MJ of heat, enough to melt about 130 kg of ice or 151 kg (103 liters) of the Glauber's salt mix; a person during 24 hours burns perhaps 2500 kcal, enough to melt 31 kg of ice or 37 kg of the Glauber's salt mix (25

liters). (Other low-melting-point materials exist, but most are not affordable at the kilogram scale; eutectic sodium/potassium nitrate is, but melts at 260°; hydrated sodium silicate melts at 72°. A variety of other phase-change materials are commercially available.)

• Seasonal thermal energy storage are large thermal masses (whether phase-change or otherwise) intended to store enough heat to keep you warm all winter or cool all summer. Ballparking, a kilowatt per person over six months is 15.8 GJ per person, which is 47 tons of ice or 55 tons of the Glauber's salt per person, about 38 cubic meters (38000 liters); or 2100 tonnes of feldspar (820 m^3 at 2.56 g/cc) with a 10° temperature swing. This may sound like an impractically large amount of material, especially the feldspar, but even in that case it fits into a 12-meter-diameter sphere; it's house-sized, not city-sized. You don't have to move all that material, but you do have to somehow control fluid flow through or near it, with a typical technique being to perforate a field with boreholes 3–8 meters apart.

• Double- and triple-paned windows are hermetically sealed, typically with argon in between the panes, which dramatically drops heat loss and noise transmission through the window without noticeably affecting light transmission.

• Vacuum-insulated glazing is a more advanced version of double-paned windows with vacuum between the panes, reaching R-values as high as 12.5 K m^2/W.

• Evaporative coolers come in many forms, such as spray mist nozzles, box swamp coolers with coarse vegetable fiber such as wood wool, and open water pools; it trades an increase in humidity (and a loss of water) for a decrease in temperature, and in some cases the downdraft of the cooler air can also be harnessed. If you can arrange for the water to evaporate outside your house rather than inside, you avoid the humidity increase, but you also lose most of the cool.

• Flat-plate solar thermal collectors are very-low-cost ways of harvesting the sun's heat; they can be a simple aluminum sheet painted black with a thin Styrofoam backing with water pipes welded to it, or even black-painted plastic with channels running through it. (In cold climates, you need some antifreeze in the water, too.) Typical efficiencies are in the 40% – 60% range, several times higher than photovoltaic. With transparent plastic or glass over the top, the water can heat up efficiently to over 50° — not enough to drive a heat engine efficiently, but plenty for climate control or a hot tub, at a very low equipment cost per watt. The Drake Landing Solar Community, at the chilly latitudes outside Calgary, gets 97% of the energy it uses for climate control with this technique, storing it in a borehole field as described above.

• More elaborate kinds of solar thermal collectors, capable of higher temperatures and higher efficiencies, include evacuated-tube types, concentrating types, and types with wavelength-selective paints. Higher-temperature collectors may have to be made of more expensive materials (copper rather than plastic) and use more exotic fluids (oils or molten salts rather than water).

• Thermosiphons are an arrangement of the elements of a solar thermal collecting system such that no extra pump is needed: the heat sink or heat store is placed at a higher elevation than the solar thermal collectors, so that the warmer water from the collectors will rise into the sink or store, replaced by denser cool water. Often a backflow

prevention check valve prevents backward flow at night.
• Thermal radiators are simply solar thermal collectors used at night to radiate unwanted heat as infrared light into space. Wavelength-selective paint is counterproductive for this use. In some systems, the radiator is simply an open pan or floor flooded with water, which is free to shed heat both by radiating and by evaporating. Without the evaporation, the cooling rate is about 75 W/m² at normal temperatures.
• Short-term thermal storage tanks can store heat or cool (thermal absorption capacity) in thermal mass (probably of water) or a phase-change substance. In extreme cases, this could be done in a stainless steel Dewar flask, but less extreme temperatures permit the use of inexpensive plastics with no insulation or inexpensive insulation.
• Solar air heaters are solar thermal collectors that heat air directly rather than water. This reduces the need for waterproof materials, but because of air's much lower specific heat, requires larger systems and more flow. The Trombe wall is one well-known version of this system.
• Venturis and other fluidic pumps permit the use of one fluid flow to produce another, without any moving parts. This could be useful to drive ventilation air currents from convection currents produced by a solar air heater.
• Heat pipes are a lighter, faster way of moving heat than water-filled plumbing, and they also don't break if they freeze. They're sealed and filled with low-pressure water vapor, which rapidly flows to cool parts of the pipe and condenses, then flows along the inner walls until they reach a warm part, where it evaporates again.
• Non-imaging optics permit concentrating sunlight to a very high brightness in a compact space without having to track the sun (much); this is useful for feeding light to light pipes and also for heating fluids.
• Desiccant dehumidifiers are an alternative to refrigerative (or "compressor") dehumidification when humidity is too high. Instead of chilling the air to condense the humidity, the air is run through a desiccant (silica gel or synthetic zeolite) at ambient temperature, and the desiccant, usually mounted in a rotor, is then heated to release the humidity and recycle it for further dehumidification. These have several advantages over refrigerative dehumidifiers: they operate from readily available heat rather than expensive mechanical energy; they can typically dehumidify down to lower temperatures and lower humidities; the extracted humidity is in a more manageable form of vapor rather than liquid; the machinery is much simpler. They also cool the dehumidified air, so they are an alternative to absorption chillers (see below) for solar air conditioning.
• Hypocausts, or underfloor heating, are a great seven-thousand-year-old luxury, universal in Korea, in which the house is heated through its floor rather than by directly heating its air; https://www.youtube.com/watch?v=P73REgj-3UE shows the construction of one from stone and mud. This is more comfortable than the air-heating approach, because people are more comfortable when the radiant-heat temperature is higher than the air temperature. If energy is abundant, it may actually be worthwhile to use it to cool the air while heating the floor. The heat can be carried through the floor in channels for air or water or provided by electric heating

elements; the standard approach currently is to use cross-linked polyethylene (PEX) pipe (good up to 85°) with water.
• Heated counters and heated toilet seats are other similar luxuries that can be provided in like manner. Defrosting plates made of thick aluminum could be useful for many purposes.
• In snowy climates, such a heat reservoir could be used to melt snow covering driveways or roads. Experiments have been done of collecting heat using cross-linked polyethylene (PEX) pipes embedded in asphalt.
• Chilled slabs are the same system, but used to provide radiant cooling rather than radiant heating. I haven't had the opportunity to experience these, but they are in use in places.
• Absorption chillers are refrigeration systems which can be operated entirely from heat, even including low-temperature heat such as that provided by flat-plate solar collectors. They are in common use in camping refrigerators powered by propane flames, and in large-scale commercial use, but they could be used as heat pumps in a variety of climate-control uses, pumping massive amounts of heat entirely by solar thermal power. "Solair" was a 2009 EU research project aimed at commercializing small-scale air conditioning by this technique, achieving a coefficient of performance of 0.7 powered by a 65° heat source. The application to camping refrigerators requires ammonia, which is dangerous (corrosive, volatile, and inflammable, producing toxic fumes), but much safer aqueous lithium bromide is adequate at air-conditioning temperatures. This refrigerant appears to require some moving parts, even though the ammonia version does not.
• Reflective-wall booths, like those used for indoor marijuana growing, and directional LED task lighting can improve light usage efficiency. This should allow people to achieve greater well-being with relatively small amounts of light. Full daylight is on the order of 50 kilolux, which is to say 50 kilolumens per square meter. If you are sitting in a reflective-wall booth where your skin and some reading materials absorb the majority of the light, you might have 4 m² of absorbing area, thus needing 200 kilolumens. A normal GE Polylux 70-watt T8 fluorescent tube produces 6.3 kilolumens, so you would need 32 of them, a total of 2200 watts. Even at an eighth of that, though — 4 tubes, 6.3 kilolux, and 200 watts — you're still twelve times brighter than a normal office. The GE high-efficiency electronic ballast for this setup costs US$21 at retail.
• High-efficiency lighting systems such as sodium vapor high-intensity discharge lighting (favored by marijuana growers) have even higher luminous efficiency than fluorescent lights. While a candle is has a luminous efficiency of 0.04%, quartz halogen bulbs are around 3%, and the fluorescent tube is 12–15%, high-pressure sodium lamps can reach 22%.
• Speculatively, instant hot-water heating can be carried out by running the water through a heat exchanger with a much hotter substance; for example, a small amount of molten salts could be kept around 300°, or a larger amount of oil at above 100°, and water circulated through a heat exchanger with it would flash into steam. This is primarily useful for direct steam applications like foaming milk for cappuccinos or melting cheese, since simply keeping the heat in a larger quantity of water at a lower temperature would avoid any explosion dangers and would have lower heat losses.

• Speculatively, you could heat air for cooking a convection oven in the same way. Perhaps a gravel bed or large rock riddled with air channels would be a better heat reservoir for this than liquids.
• Dehydration of food, damp laundry, and similar things can be carried out with solar-heated air, either inside or outside the house.
• Speculatively, you could use a wet scrubber to remove particulate matter from the house air. This could be helpful if you live in an area with significant particulate pollution, both for health and for keeping the house clean. Dust particles larger than smoke, which are less of a health problem but necessitate cleaning, are probably better removed with a cyclonic separator, electrostatic separator, or paper or fabric filter. A house of 100m² with 4 m ceilings contains 400m³ of air; removing 90 to 99% of the particulates from that air with a venturi scrubber or indoor spray tower, at 0.5–3 ℓ/m³, would require some 200 to 1200 ℓ of water, which is a thing you would might want to do for all the air your bring in from outside. At 1500 kPa of water pressure in the nozzles of a spray tower, cleaning the whole house's air with 800 ℓ of water would use 1200 kJ (⅓ kWh).

Ideally, many or all of these systems would be available to the house-dweller to deploy as they saw fit, rather than hooked up in a fixed topology at build time; heat and cold reservoirs at various temperatures would be continuously replenished when possible, thermostats and humidistats would be programmed to provide a healthy diurnal variation in the living space, and when excess energy was available, it could be spent on greater illumination or radiant heating.

While such control of the indoor climate is pleasant for humans, we can after all put on a coat or take a cool bath if we're uncomfortable. For gardening, however, the differences in productivity from even small changes in temperature can be immense. More factors begin to matter — you care not only about the soil's temperature and humidity, but also its pH and its contents of nitrogen, phosphorus, potassium, and sulfur; and the carbon dioxide content of the air has a significant effect on plant growth. (You could obtain carbon dioxide by calcining calcium or magnesium carbonate, which you then deploy again as air scrubbers.)

In most climates, this level of climate control ought to enable immense gains in agricultural productivity: you should be able to grow sugarcane, bamboo, corn, squash, or rice even at periarctic latitudes, and with CO_2 supplementation, they should grow even faster than in their naturally optimal environments.

Plumbing with crossbars

Passing air through pipes or ducts is an efficient way to move two or three of our five crucial commodities: heat, air, and water in the form of humidity. Even if we want to heat or cool liquid water, for example for washing dishes or laundry or for a shower, it's probably a good idea to use air to transfer the heat or cool from the relevant reservoir to a heat exchanger for the water.

If we have the desire to

Topics

Computation with strain

Kragen Javier Sitaker, 2019-06-13 (17 minutes)

Traditional mechanical analog "computers" represented quantities only as displacements (sometimes linear, sometimes angular), but perhaps by using other quantities such as strain (tensile and shear) and velocity, we could construct simpler flexure computing devices that compute faster.

Needless to say, such devices ought to be as small as possible given the precision that is required; no advantage accrues to a calculating device from requiring lots of power and weighing a lot.

It's unfortunate that we're stuck with the misleading term "computer" for such analog devices, because in the digital realm, we do not call an adder, multiplier, or decoder a "computer"; we reserve that term for essentially Turing-complete digital systems. But it is not even clear what the analog equivalent of Turing-completeness would even be; an "analog computer" is a set of building blocks that can be reconfigured and connected together in different ways to perform different calculations.

Linear and nonlinear springs

The equilibrium or steady-state configuration of a damped sprung mass with a given force applied is a displacement that depends on the force, but not the mass. In some nonlinear cases, like Euler columns, there may be more than one possible equilibrium configuration, even without inelastic deformation. (Merkle's buckling-spring logic depends on this for its memory; see mechanical computation: with Merkle gates, height fields, and thread (p. 2494) for depth on such systems, and Elastic metamaterials (p. 719) and Snap logic (p. 2580) for some more related stuff.)

Hookean springs make this equilibrium displacement a linear function of the applied force, or to look at it the other way, they make the force a linear function of the displacement. This means that if you have a few different linear springs pushing on a single lumped mass, its equilibrium displacement is going to be a weighted sum of the displacements of the other ends of those springs.

Once any nonlinearity enters the picture, which can happen through rotation (as in Euler columns), contact (as a plucked ruler hanging off the edge of a table rattles against the tabletop), or maybe even fluid dynamics, the equilibrium displacement–force relation becomes not only nonlinear, but also potentially nonmonotonic, discontinuous, and multivalued (i.e., not a function).

Considering continuous cases, though, let's think about a cantilevered beam of constant width tangent to a cylinder. If you press it toward the cylinder, it starts to wrap around the cylinder, shortening the lever arm and increasing its Hooke constant. By using a varying curve, the increase of the Hooke constant can be tailored to the integral of some arbitrary nonegative function. You can thus get a nearly arbitrary continuous displacement difference between two parallel rulers wrapping around different-shaped curves under the influence of two parallel coil springs from some remote source.

Additionally, Euler-column-style rotations can provide

nonmonotonic force–displacement relationships, though even without stiction, this can result in memory.

Linearly converting a force to a displacement or vice versa can be done with linear springs; computing an arbitrary nonlinear function of a force as a displacement, or vice versa, can be done by the methods described above. Adding forces can be done by connecting multiple springs to the same object. In particular, it should be possible to use two cylinder-wrapped rulers to compute the logarithms of two input forces as positions, convert these positions back to forces through much lighter linear springs which push on another object, and convert its position to an exponentially increasing force by attaching it to a third even lighter cylinder-wrapped ruler, which thus at equilibrium computes the product of the input forces.

This suggests the ability to compute arbitrarily complex continuous numerical functions, although clearly some kind of energy amplification is necessary to prevent output "circuits" from unduly loading the inputs and to permit chains of more than three or four levels of depth.

These spring systems, like flexures in general (see Flexures (p. 2211)), have no backlash as long as the material is perfectly elastic, eliminating one of the major sources of error in mechanical analog computation.

Integrators and non-equilibrium systems

In general, though, sprung-mass behavior does in fact depend on the mass. The net force on the mass, due to the curves of the various springs acting on it and its current displacement with respect to those springs, produces an acceleration inversely proportional to the mass; the mass's displacement is the integral of its velocity from its initial position, and its velocity is the integral of its acceleration from its initial velocity.

This suggests a much more appealing way of building a time-domain integrator than the disc-on-plate and ball-on-plate devices used by Bush's differential analyzer: represent the time-domain quantity you want to integrate as a force, and then the velocity of some mass gives you its integral!

This has a couple of major disadvantages, though. How do you convert the velocity back into a force? The usual mechanisms for this involve either viscous fluid friction (which is notoriously tricky due to the laminar–turbulent transition), or electrical generation, like old car speedometers or eddy-current magnet braking. And how do you exert a controllable force on an object whose position is not an input to your computation?

I propose that we represent the integral not merely as a velocity but as a *harmonic oscillation amplitude*. A tapered, cantilevered bar will vibrate at some natural frequency, and its current oscillation amplitude is roughly the integral of the oscillatory forces that have been applied to it in the past. The oscillations decay over time in accordance with its Q factor, but Q>100 routinely happens by accident in oscillatory mechanical systems; I suspect this may provide sufficient time to perform a useful calculation to the precision afforded by the rest of the system.

If the cantilever is vibrating only in a single dimension, there are times during its motion that all of its energy is elastic, and other times

when it's all kinetic. This seems like it could complicate the process of continuously feeding energy into it from the quantity you want to integrate; you need to feed in that energy in different frms at different times.

A possible solution to this is to make it vibrate circularly in two dimensions around its natural position. With this approach, you can always apply a displacement or force to it to add or subtract energy; only the necessary direction of the displacement or force varies. (And if you're adding force, you need to add it in a direction that is 90° from the direction you'd need to add a displacement in.)

This has the possibility that you'll end up with a non-circular oscillation, because the signal you were adding had a periodic or near-periodic component at or near the frequency you chose for the oscillation. To remove that effect, you can add the squares of the X and Y component amplitudes.

To convert this oscillating integral value to a single-ended displacement, you probably need to rectify it, ideally a tiny fraction of it so as not to spoil the Q too much. Rectification can be achieved by the earlier-described kinds of nonlinear elastic systems or more simply by banging shit together or trying to push on a rope.

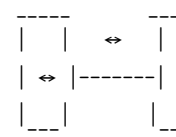
Speaking of pushing on ropes, that may be a way to get the necessary amplification.

Amplification through Euler columns and loose string

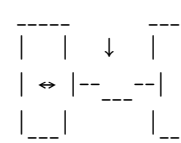
The crucial invention of Bush that made the differential analyzer practical was not the integrator itself — that dated back decades — but the "torque amplifier", a negative-feedback friction device that strongly drove an output shaft to the same position as a weakly-driven input shaft. How can we do something similar without friction, backlash, wear, lubrication, and the other headaches of sliding-contact machinery?

Euler columns

If you have a power-supply block that is being driven back and forth, by your power supply, with some fixed displacement, and it is connected to another block through a straight slender rod, it will tend to drive that other block back and forth, unless the force needed to drive the other block is so large that the rod approaches Euler-column instability:



However, a relatively small force pushing on the side of the rod can get it to buckle when it otherwise wouldn't, greatly reducing the force transmitted to the driven block:



The transverse displacement is relatively large (much larger than the displacement that would have been transmitted to the driven block) and oscillating, but the transverse force need do no net work over time. Consequently you want to apply this displacement through a spring with high compliance.

The column is shortened by this process, so I think the driven block must also have relatively large compliance, at least at DC; if it's stiffly, I think it won't work.

This provides a locally-continuous mechanism both for controlling a large force with a small force and for controlling an arbitrarily large amount of power with an arbitrarily small amount, like a MOSFET.

Making straight string crooked and thus "loose"

Possibly this can be generalized to adding slack to string by pulling it sideways. While the vertical "control string" is slack, as long as the "drive string" is under some DC tension, it faithfully transmits AC movements with high stiffness:

But if a little DC tension is applied to the control string, adding some slack to the drive string, the effective stiffness of the string drops greatly, and much less of the AC signal is transmitted:

(For frequencies sufficiently high to require a transmission-line model of acoustic propagation in the string, you also have to account for reflection from the knot and the conversion of longitudinal waves in the drive string into transverse waves in the control string.)

I think this may be the thread-based amplification construct I was looking for when I wrote about computing with thread in mechanical computation: with Merkle gates, height fields, and thread (p. 2494). What I had come up with at the time was the idea of frictionally clamping a thread running lengthwise down a dowel by tightening another thread that ran around the dowel several times, but I'm not totally convinced that's a workable mechanism. I'm more optimistic about this new design.

I've rigged this up on the back of a chair with a couple of meters of my knitting yarn, and pulling the control string certainly does dramatically attenuate the AC *displacement* that gets transmitted down the yarn. However, it also adds tension to the yarn, so it's hard to tell whether it results in less energy being transmitted than before; I don't have a quantitative measurement of the force. It also has the effect of making my bedroom look slightly more like *A Beautiful Mind*, and not the positive achievement parts.

Since this really only allows a DC signal to control an AC signal, you need a rectification step to produce a fully composable analog computation system made out of thread. Thread is fantastic at rectifying signals — as I said above, it's a cliché that you can't push on

a rope — but it's even worse than electricity at having memory, so you're going to have a lot of ripple in your rectification output. For digital computation, you can probably deal with ripple in a brute-force way if it comes to that — for example, separately switch two phases of a quadrature signal and full-wave rectify both of them into a single junction — but for analog "computation" it seems like a big problem.

Making loose string tight

Much larger and clearer amplification is available easily with string, though. Although the above contrivance can possibly be made to work, it has the major difficulty that it is trying to "loosen" string by adding tension to it, so as to make it no longer straight and thus less "stiff" to a power supply that's trying to drive it. Going at the problem backwards, there's a much simpler solution: try transmitting a displacement signal by moving one end of a loose string, and you will find that it doesn't reach the other end, unless the signal is so large as to pull all of the slack out; here # is used to indicate fixed, solid material:

```
#######
 |   |
_|   |_
↔|\ /|
 | v |
 |   |
#######
```

The slack string can attenuate an "ac power supply" by at least several orders of magnitude. (Here the vertical strings are "leaf springs" whose tension provides a restoring force for the two knots implicit in the diagram.)

But if something takes up the slack in the string, for example by pulling it at right angles, the string suddenly begins to transmit movement, although imperfectly:

```
#######
 |   |
_|   |_
↔|\ /|↔
 | v |
 | |↓|
###|###
```

Some of the energy from the power supply is also transmitted to the "gate string", but I think it is possible to choose the angles and compliances involved such that that energy is very small. In particular, if the slack "channel" string is nearly straight in its slack state, the force applied to the "gate" string will be much smaller than the force transmitted from the "drain" to the "source" through the channel — say, ten times less — so if the gate and source are of similar compliance, the interference energy backfed to the gate could be around 1% of the energy successfully transmitted to the source. I am not sure about the energy needed to pull on the gate string.

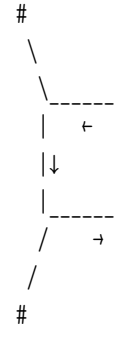
An even simpler nonlinear string mechanism is an "OR gate"

consisting of two input strings that can pull on a common knot, pulling an output string if either of the two input strings is pulled:

Direction change for a limited range of displacements is easily achieved with a knot that can swing in a circular arc because it is anchored to a fixed point:

Different parts of the arc provide different mechanical advantages between the string chosen as input and that chosen as output; if the arc is large compared to the expected displacements, this mechanical advantage will be relatively constant.

"Negation" of displacements, when necessary, is available through a sort of pulley mechanism made from two or three such direction-change arrangements:



But I think that, although these "OR" and "NOT" arrangements might seem logically universal, they might lack some necessary kind of amplification for full combinational logic, petering out somehow after a few stages. I think the right-angled-slack-string approach described above, coupled with a power supply, should provide all the amplification you could need.

Stick-slip violin amplification

Bowing a violin string tends to amplify existing vibrations in it; when a transverse wave movement reaches the bow, it rips the string free of the bow hair, causing it to slip, and when a peak or valley of the transverse wave is at the bow, the string can stick to the hair, which pulls it along and stretches it, adding energy that will be released at the next slip. This form of amplification is extremely nonlinear, but might work.

Amplification through multipliers

As described above, a force multiplier flexure — where the output force is proportional to the product of the input forces — is fairly easy to rig up, although to avoid enormous imprecision, the vast majority

of the input energy remains as elastic energy within the device, with only a tiny fraction reaching the output. But that might be okay if you're multiplying a weak input *signal* by a high-power power supply; it's not really a problem if most of the energy in the amplifier is resonating back and forth between the multiplier and the power supply, or even just being burned up as heat, as long as the output signal is stronger than the input signal.

I'm not sure if this will work.

Topics

- Physics (p. 3632) (119 notes)
- History (p. 3500) (71 notes)
- Mechanical things (p. 3569) (45 notes)
- Physical computation (p. 3631) (26 notes)
- Self-replication (p. 3703) (24 notes)
- Mechanical computation (p. 3568) (7 notes)

Japan can achieve energy autarky via solar energy, but not much before 2027

Kragen Javier Sitaker, 2017-07-12 (4 minutes)

Ian Welsh tweeted, "Japan HAS to import energy."

Is he right? It turns out that photovoltaic changes things; Japan can sustain its current energy consumption on about 5% of its land area, using plain cheap photovoltaic panels. It will probably get the majority of its energy from photovoltaic panels starting in the mid-2020s, based on current growth rates.

Wikipedia says Japan used 477.6 Mtoe in 2011, which is 19.996 exajoules, a number suspiciously close to 20, or 630 GW average. The accompanying chart shows that the number is closer to 21 EJ, or 670 GW, roughly level since 1995. Solargis's solar resource chart gives numbers of global horizontal irradiance (GHI) from 1200 to 1600 kWh/m²/year; a rough average might be 1400, although Wikipedia gives somewhat higher numbers of 4.3 to 4.8 kWh/(m·day), which works out to 1600–1800 kWh/m²/year. Solargis also offer a "potential PV electricity generation" map for optimally-tilted free-standing crystalline silicon modules, ranging from some 900 to some 1600 kWh/kWp, with most of the land around 1200.

Japan's area is 378000 km², which gives us a total GHI of about 500 PWh/year, or 60 TW. With 16%-efficient (polycrystalline Si) horizontal solar cells, this gives us 10 TW, which is about 15 times Japan's total national marketed energy consumption.

That is, Japan could satisfy its current energy demands with about a fifteenth of its total solar resource, occupying perhaps 5% or 10% of its land area. To supply these 670 GW with an unremarkable solar photovoltaic capacity factor of 20% would require 3.3 TW of solar panel nameplate capacity (TWp); currently installed at the end of 2016 is 42.75 GW, about 78 times smaller. Installed PV capacity rose from 3.618 MW at the end of 2010 to 42.75 GW at the end of 2016, a factor of 11.82, an exponential growth rate of 51% per year. At this exponential rate, Japan would need another 10.6 years (mid-2027) to reach 100% of energy from domestic solar photovoltaic. Presumably the growth rate will slow earlier than that, perhaps around 2023.

At current photovoltaic prices of €0.53/Wp (May 2017, Japan/Korea market), 3.3 TWp is 1.7 trillion euros. Japan's nominal GDP is currently US$4.73 trillion per year, although only has US$697 billion in exports (€608 billion at US$1.146/€). So this represents an investment of about 2.8 years of Japan's exports or about 0.41 years (5 months) of Japan's GDP. Currently Japan pays a very high tariff of ¥42/kWh for solar, which, at ¥114/US$, is US$0.37/kWh or €0.32/kWh. So the full €1.7T investment will be paid for by the next 5.3 trillion kWh (19 EJ) of solar energy consumed in Japan.

That's only the cost of the panels, but at this point the levelized cost of new PV plants is under US$1 per peak watt, even including the

uncertainty premium to investors. That is, the cost of the whole plant is less than twice that of the panels. We can expect that cost to continue coming down.

There are some other obstacles to going majority-solar, such as utility-scale energy storage for baseload power (photovoltaic panels produce no significant power at night) and producing liquid fuels for transport. I expect them to be solved once the incentive of abundant but intermittent and stationary solar energy is there, which will probably require reducing the high solar tariff.

So, although Japan can definitely afford to switch to mostly photovoltaic, it can't do it much sooner than 2027, because it doesn't have enough exports.

utility-scale

Topics

- Energy (p. 3438) (63 notes)
- Economics (p. 3424) (33 notes)
- Solar (p. 3717) (30 notes)
- The future (p. 3746) (20 notes)
- Japan
- Capacity factor

How can we build an efficient microcontroller-based amplifier?

Kragen Javier Sitaker, 2016-07-13 (5 minutes)

Some folks at the Fabricicleta were asking me a few years back about making high-efficiency amplifiers for audio from bicycles, so that you can really blast music driven by a five-watt hub generator I guess.

Suppose you use, say, a 180MHz Cortex M3 LPC1830 as your processor, and suppose that you can only output one bit every two CPU cycles, or 90MHz. A simple pulse-density modulation scheme that swings from 100010001000... to 111011101110... should limit the artifactual waveform from the PDM modulation to one fourth of the bit rate, or 22½MHz; it can approximate waveforms of up to -6dB. Then you can feed this bitstream into a Darlington or a power FET or whatever, then filter its output reactively to avoid power dissipation.

(There may actually some lower frequencies that show up from dithering: 10101101010110..., for example, has an oscillation whose period is seven bit times.)

This 22½MHz is three decades or ten octaves above anything we need to reproduce for audio purposes. Even with just a single-pole 6dB/octave LR or LC filter, you get 60dB of attenuation as long as the cutoff frequency is at or below 20kHz; I think that with two stages of LC filter, you'll get 24dB/octave, which would give you 240dB of attenuation. You should be able to use fairly small-value inductors and capacitors for this, because the total amount of energy stored for half a cycle of a 22MHz wave is very small.

Supposing we're driving an 8Ω speaker, we'd probably like the output impedance of the final-stage filter to be a lot smaller than that at 20kHz. This actually means that the entire signal path from the amplifier through the two inductors to the speaker needs to have a lot less than 8Ω impedance, say, 0.8Ω.

Capacitive reactance is $1/\omega C$, while inductive reactance is ωL. This suggests that our maximum total inductance is when $\omega L = 0.8\Omega$, which happens when the inductors total 6.4μH. And we'd like the second one to have most of that, so, say, 0.6μH in the first inductor, and 5.8μH in the second one, thus 75 milliohms of reactance in the first and 730 in the second.

Now we want our capacitors to ground to have the same reactance as the inductors at our 20kHz cutoff frequency. So, for the first one, $75\ m\Omega = 1/(2\pi\ 20kHz)C$, $\therefore 2\pi\ 20kHz\ C = 1/75m\Omega$, $\therefore C = 1/(2\pi\ 20kHz\ 75m\Omega) = 106\ \mu F$. That's a reasonably attainable value, though it has to be electrolytic. The second one need only be 11μF.

Digi-Key's most popular inductor, with 1,077,238 in stock at the moment, is a 10¢ TDK 82nH 150mA inductor, the MLK1005S82NJT000. Sadly, its DC resistance is 1.8Ω, because it's in an 0402 package (1mm × 0.5mm), so it can't serve as a high-efficiency reactive filter for an 8Ω speaker unless we put a bunch of them together. Which is actually a reasonable thing to do.

Digi-Key's most popular inductor in the right range of inductance and DC resistance is Murata's LQM2HPN1R0MG0L, a 34¢ 1μH

(±20%) 1.6 amp 55mΩ ferrite-core inductor in a 2520 (2.5mm × 2.0mm) package. Its minimum self-resonant frequency is 60MHz, which suggests that it may actually not be a great choice here; it might capacitively couple through the exact high-frequency energy we want to stop.

If I restrict to only inductors with a self-resonant frequency over 180MHz, the most popular is the TDK MLZ1005MR47WT000, a 28¢ 470nH 500mA 200mΩ multilayer ferrite-core inductor with a 260MHz self-resonant frequency, in the same 0402 package as the most popular one. It's marketed as an "inductor for decoupling circuits". Its saturation current is 120mA, which I guess means that above 120mA it doesn't induct any more. 120mA on an 8Ω load would be 115 milliwatts, but I guess you could use several of these inductors.

I don't know, I guess I have a lot more to learn about reactive filters.

Could you just wind the inductor yourself?

$L = \mu n^2 A / l$, where μ is the permeability, n is the number of turns, A is the cross-sectional area, and l is the length. So if I want an inductor of 6 μH or so, 12 turns with a radius of 1 cm and a length of 1 cm and no core would be about right; or 39 turns with a radius of 1 cm and a length of 10 cm and no core. I feel like you could probably keep the parasitic capacitance and resistance down to trivial levels with that approach.

Topics

- Electronics (p. 3430) (138 notes)
- Pricing (p. 3646) (89 notes)
- Audio (p. 3331) (40 notes)
- Microcontrollers (p. 3580) (29 notes)

Git data

Kragen Javier Sitaker, 2007 to 2009 (5 minutes)

I looked at checking all the almost 400MB of crawl data into Git. Here's what I learned:

• nobody on #git had experience doing anything like this, so it's not terribly well-traveled ground.
• git's compressed index file format crunches the 389MB down to 110MB.
• despite this, doing a git clone on the same machine is more than twice as slow as cp -a, despite occasional claims to the contrary for other data sets.
• currently checkouts take less than a second, on the same machine. This would make them take a couple of minutes.
• downloading the data over my Argentine cable modem connection takes almost 45 minutes, as opposed to 13 seconds now. git is about 10% slower than rsync for the initial download. rsync transfers only about 101MB. I suspect that git is slower primarily because my laptop's disk is encrypted and painfully slow as a result.
• git is a lot quicker at subsequent small data updates than rsync, but neither one is slow enough to make much of a difference.
• git is better at dealing with subsequent updates in the sense that it can propagate them in either direction.

Here's what I concluded:

• checking the data into the main dev repository, which was my initial thought, is probably a bad idea, because it makes initial checkouts on a new machine take at least a couple of minutes, and maybe more than an hour, if your connection is slower than mine.
• checking the data into a git repository, rather than syncing it around using rsync, is probably a good idea. (But probably not for really large single files like the Freebase dump.)

Creating and doing initial checkouts:

```
kragen@watchdog:~$ time cp -a /home/watchdog/web/data web-data-copy

real    0m50.081s
...
kragen@watchdog:~$ cd web-data-copy/
kragen@watchdog:~/web-data-copy$ git init
Initialized empty Git repository in .git/
kragen@watchdog:~/web-data-copy$ git add .
^C
kragen@watchdog:~/web-data-copy$ time git add .

real    0m32.319s
```

(I'm sure it took longer than that the first time before I hit ^C.)

```
kragen@watchdog:~/web-data-copy$ git commit
...
 create mode 100644 votesmart/sigs.json
```

```
 create mode 100644 votesmart/states.json
 create mode 100644 votesmart/websites.json
kragen@watchdog:~/web-data-copy$
kragen@watchdog:~/web-data-copy$ du -h .git
...
110M    .git
kragen@watchdog:~$ time git clone web-data-copy another-web-data-copy
Initialized empty Git repository in /home/kragen/another-web-data-copy/.git/
remote: Generating pack...
remote: Done counting 5334 objects.
remote: Deltifying 5334 objects...
...
real    2m22.970s
```

By contrast:

```
kragen@watchdog:~$ time git clone /home/watchdog/git/dev.git tmp.foo
Initialized empty Git repository in /home/kragen/tmp.foo/.git/
remote: Generating pack...
remote: Done counting 111 objects.
remote: Deltifying 111 objects...
remote:  100% (111/111) done
Indexing 111 objects...
remote: Total 111 (delta 38), reused 0 (delta 0)
 100% (111/111) done
Resolving 38 deltas...
 100% (38/38) done

real    0m0.822s
user    0m0.140s
sys     0m0.030s
```

On my laptop:

```
kragen@thrifty:~/devel$ time git clone \
    watchdog.notabug.com:/home/kragen/web-data-copy web-data-test
remote: Generating pack...
remote: Done counting 5334 objects.
remote: Deltifying 5334 objects...
remote:  100% (5334/5334) done
Indexing 5334 objects.
remote: Total 5334 (delta 3631), reused 0 (delta 0)
 100% (5334/5334) done
Resolving 3631 deltas.
 100% (3631/3631) done
Checking files out...
 100% (5714/5714) done

real    43m43.731s
user    2m9.444s
sys     0m16.441s
kragen@thrifty:~/devel$ time rsync -Pavzz \
    watchdog.notabug.com:/home/watchdog/web/data/ web-data-rsync-copy/
...(12000 lines omitted)...
sent 127708 bytes  received 101644227 bytes  45668.36 bytes/sec
```

```
total size is 374634655  speedup is 3.68

real    37m9.568s
```

An rsync with no changes is faster:

```
kragen@thrifty:~/devel$ time rsync -Pavzz \
    watchdog.notabug.com:/home/watchdog/web/data/ web-data-rsync-copy/
receiving file list ...
6044 files to consider

sent 20 bytes  received 106850 bytes  6894.84 bytes/sec
total size is 374634655  speedup is 3505.52

real    0m16.043s
user    0m0.228s
sys     0m0.140s
kragen@thrifty:~/devel$
```

Checking out the current dev git is much faster than downloading all that data:

```
kragen@thrifty:~/devel$ time git clone \
    watchdog.notabug.com:/home/watchdog/git/dev.git
remote: Generating pack...
remote: Done counting 111 objects.
remote: Deltifying 111 objects...
remote:  100% (111/111) done
Indexing 111 objects.
remote: Total 111 (delta 38), reused 0 (delta 0)
 100% (111/111) done
Resolving 38 deltas.
 100% (38/38) done

real    0m13.494s
user    0m0.360s
sys     0m0.124s
```

After adding a file on each end:

```
kragen@thrifty:~/devel/web-data-test$ time git pull
remote: Generating pack...
remote: Done counting 4 objects.
Result has 3 objects.
Deltifying 3 objects...
 100% (3/3) done
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking 3 objects
 100% (3/3) done
* refs/heads/origin: fast forward to branch 'master' of
    watchdog.notabug.com:/home/kragen/web-data-copy
  old..new: cab3ce4..15efdff
Trying really trivial in-index merge...
Wonderful.
In-index merge
```

```
 foo |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 foo

real    0m7.782s
...
kragen@thrifty:~/devel/web-data-test$ time git push
updating 'refs/heads/master'
  from 15efdfff9a666495af1a3de8e062f07177e8dbbf
  to   062da7cf925ff4d7127c4a53d1698b0829a54ffd
Generating pack...
Done counting 7 objects.
Result has 5 objects.
Deltifying 5 objects.
 100% (5/5) done
Writing 5 objects.
 100% (5/5) done
Total 5, written 5 (delta 2), reused 0 (delta 0)
refs/heads/master: 15efdfff9a666495af1a3de8e062f07177e8dbbf ->
    062da7cf925ff4d7127c4a53d1698b0829a54ffd

real    0m4.806s
```

Topics

Calculations about desalination in Israel

Kragen Javier Sitaker, 2016-08-11 (3 minutes)

(A comment I made on an article on the orange website.)

There are some crucial details here I'm not understanding.

The article says Israel needs (or needed?) 1.9Gm³/year (60 kiloliters per second, 60kℓ/s) of water, of which it got 1.5Gm³/year (48kℓ/s) from natural sources; but now (or when the plants opened?) it gets 127Mm³/year from the 2005 Ashkelon desal plant, 140Mm³/year from the 2009 Hadera desal plant, and 150Mm³/year from the new Sorek plant. These total 417Mm³/year (13kℓ/s), but the article says Israel's current total is 600Mm³/year (19kℓ/s) from desalination plants, which is more than 417. Also, it says, Israel now gets 55% of its domestic water from desalination. But 55% of 60kℓ/s is 33kℓ/s, which is 14kℓ/s more than the 19kℓ/s it says Israel gets from desalination. I suspect there's some confusing mixing of categories here leading to these numbers not adding up properly.

The other really crucial questions are about capex and opex. How much do these plants cost to build, and how much do they cost to run? How much of that is the energy cost?

They give the figure that the Sorek plant's reverse osmosis runs at 70 atmospheres (7.1 MPa; fucking Christ on a stick, why do people keep inventing new non-SI units?) which means that each liter of output water requires 7.1kJ of mechanical energy to force it through the membrane. There are presumably some other energy costs, but that may be the bulk of it. At US$100/MWh (US$28/GJ; fucking non-SI units again), which is a reasonable ballpark for the levelized cost of electrical energy, that's about 200 μUS$/ℓ or 200 US$/Mℓ. Irrigation water is commonly quoted in US$ per acre foot in the US; this is US$240 per acre foot, which would be a very competitive cost.

But it says the cost is US$0.58/kℓ, which is 580 US$/Mℓ or US$715/acre foot, about three times the cost of the mechanical energy and high enough that many crops are uneconomical. It is crucial to understand where that extra cost is coming from.

One of the comments on the Ensia version of the story claims that the new Carlsbad desal plant is selling its water for US$2000/acre foot. I don't know if that's true, but it's about 150% higher than the projected costs.

Links, in case the Scientific American web site is lost:

Original article

Bar-Zeev et al.'s article about their bioflocculation anti-biofouling prefiltering apparatus DOI: 10.1016/j.watres.2013.03.013

"Climate change in the Fertile Crescent and implications of the recent Syrian drought" doi: 10.1073/pnas.1421533112

Topics

• Physics (p. 3632) (119 notes)

Derivative based control

Kragen Javier Sitaker, 2019-11-12 (6 minutes)

In cybernetics, hierarchical control systems have a "higher-level" control system which provides set points to (possibly multiple) "lower-level" control systems, which the lower-level systems then seek. For example, a reaction-wheel-robot attitude-control system might plan a maneuver and then command reaction-wheel control systems to seek particular velocities for their respective reaction wheels.

It occurs to me that under some circumstances it makes sense to send more than just a set point. For example, if the central control system for a robot arm commands the servo in each of its joints to a particular position, the force the servo applies to reach that position will depend on the difficulty of the movement — the mechanical impedance, as they say, and may reach a very high level if unexpected impedance is encountered, perhaps because a research assistant's head is unexpectedly in the way of the arm. It is possible for the central control system to pay attention to these error signals and send new commands under those circumstances, but this puts stringent demands on the response time of the central control system.

It might make more sense to command the lower-level control with not just a set point but also a gradient, including a derivative with respective to any of a variety of variables — in the above case, it might be the force. In particular, the derivative of position with respect to force is simply the mechanical compliance, so this amounts to commanding not just a position but also a compliance.

Under some circumstances the commanded compliance (or other analogous gradient) would be achieved by electronic or other active feedback control — position sensors feeding back to motor-control systems, for example — but in other cases it could be achieved wholly or partly through passive means.

For example, a "brushless DC motor" or a stepper motor has a variable stationary holding torque determined by the armature current when the motor is stopped; moreover, an analogous kind of control is possible when the motor is moving as well by controlling what percentage of the time the windings are carrying current that tends to slow down the rotation rather than no current; and a tendon-actuated system, such as parts of the humans' bodies, will be stiffer when both the extensor and retractor tendons are under tension ("muscle tone") than when both of them are relaxed.

In other cases, mechanical compliance can be usefully altered by taking advantage of redundant degrees of control. Holding chopsticks close to the base rather than close to the tip, for example, is a technique for increasing mechanical compliance and thus making it easier to pick up food.

In an electronic context, this "variable compliance" might amount to things like a varying gain on an amplifier, a variable DC bias on a varactor, or a bias current on a secondary winding that varies the permeability of a magnetic core, as in a magamp.

The variables whose values the lower-level control uses to perturb the set point are not necessarily directly related to the control

mechanism. For example, it sometimes makes sense for a hand computer display to brighten when the hand computer senses increased ambient light, even if the human using it doesn't command it to brighten; a quadcopter rotor might reduce the setpoint of its speed if a local accelerometer indicates that it's accelerating upwards; and so on. In general, the idea is for the lower-level control loop to have a set point that is a locally-linear approximation to the potentially more complex and time-varying function used by the higher-level loop to figure out what set point to send to the lower-level loop.

It is observable that the humans use mechanisms such as these at several levels in their motor control; this is demonstrated, for example, by attempting to use a mouse rotated 180° or to write legibly in a mirror.

Controlling the gradients of such "reflexive" low-level control loops can dramatically reduce the demands on the high-level control, in particular often permitting much higher communication and processing latency before the whole system begins to unintentionally oscillate. Also, it can allow the high-level control loop to send commands that suppress, excite, or control oscillations in the lower-level control loop (such as human voices producing higher pitches and the humans doing that freaky thing where they vibrate their eye irises sideways), also reducing the demands on the stability of that lower-level control loop.

Beyond just the gradient of the set point, it might make sense for the high-level control loop to alter other parameters of the low-level control loop; for example, if it's a PID controller, it might make sense to alter the gains of the P, I, and D components depending on circumstances. With modern electronics it is possible to get subnanosecond response times from analog electronic control loops and trivial to get submicrosecond response times.

Note that a PD controller (a PID controller with no I term) will also perturb its operating point according to the plant's response — linearly so if the plant is linear — and this can limit the force the robot arm exerts on the research assistant's head. So in a sense this is a special case of the more general concept I'm describing; but it is missing the crucial ability for the higher-level control loop to set the compliance separately from the set point, and it can only work at all in that context if the robot actuator is working against a spring or something.

It isn't necessary for the lower-level setpoint perturbation to be linear. It might be whatever nonlinear function is most convenient to compute (for example, using the exponential Ebers–Moll response of BJTs) or it might be a precision higher-degree approximation, as in mechanical naval fire control computers.

Topics

• Robots (p. 3688) (9 notes)
• Control (p. 3390) (9 notes)

What would a better Unix shell look like?

Kragen Javier Sitaker, 2018-11-27 (1 minute)

A better shell might have some or all of the following properties:

• Opens a new pty for each command and allows you to either include the whole command output or a scrollable window on it in the overall scroll. Clearing the screen should, by default, fullscreen the command output.
• Displays which commands failed and which relied on now-changed inputs. (This requires tracking the inputs of every command.)
• Optionally re-executes selected commands periodically
• Displays per-command resource usage, such as CPU time and RAM usage. By default runs commands with limits to prevent horking the whole machine.
• Provides notifications of finished, idle, and newly non-idle commands in a way that doesn't interfere with foreground tasks. This requires at least some minimum UI for configuring it.
• Copes with voluminous command output gracefully, at least up to gigabytes. Should have options to FIFO-discard, block (like ^S), and increase the scrollback-retention limit.
• Supports distributed operation, including running commands on remote hosts, inside of Docker containers, and across multiple remote hosts, as well as session recovery from network disconnection and even machine restart.
• Options for fancifying command output; for example, hyperlinking `ls` and `ps` output, or colorizing and relayouting other commands' output.
• Programmable autocomplete, of course.

Topics

C bad

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

The other day I was helping a friend of mine with a bug in his C program. It turned out that he hadn't initialized a variable that was used as an in-out parameter, carrying a buffer length on the way into a Win32 API function, and the length of the data in the buffer on the way out; and consequently the function was failing with an error, which he didn't check for. This took me about 45 minutes to figure out, talking with him, partly because my Win32 is pretty rusty. It would have taken a much longer time if he hadn't been fairly competent himself, having very nearly figured out the problem before we talked.

I asked why he was writing the program in C, where both of the two bugs that combined to create this problem were possible, instead of, Python, Java, OCaml, C#, Haskell, or really any high-level programming language. His answer surprised me:

we do everything at my work in C or C++
I'm sure that almost makes you puke

I said that was stupid. C is a beautiful language, but that doesn't make it the best tool to use for everything. Quartz crystals are beautiful too, but if you build a bridge by gluing quartz crystals together, it's going to take a lot of work, and when you're done, your bridge is still likely to be fragile. (Although, if you do it right, it will be beautiful.) Much better to build your bridge by welding steel I-beams together, maybe using some cables to bear its weight.

Descending from analogy to reality, working large programs are built by incrementally improving working small programs. There are four major reasons that each incremental improvement to a C program takes longer than the corresponding improvement to a program in a higher-level language:

• The C program contains a lot more code than the corresponding higher-level program, which makes it harder to find the parts you need to change, and harder to understand them well enough to change them when you find them.
• The new code you add is several times as large, with the consequence that it contains several times as many opportunities for bugs, and therefore probably several times as many bugs.
• In C, cause and effect are not localized. If any part of the program invokes some kinds of undefined behavior (uninitialized data, out-of-bounds pointers, double-frees, and so on), then it can interact with any other part of the program at all. This kind of spooky action-at-a-distance makes debugging difficult --- a newly manifesting bug may be caused by an error introduced in any part at all of the code. The traditional Unix solution to this problem is to *not write large programs*: instead, write small programs that do one thing only ("and do it well"), and then connect them together with a scripting language.
• The first three points are, in part, related to the absence of garbage collection in C; but garbage collection not only reduces the amount of code in your program and helps to keep cause and effect localized,

it also has other benefits. For example, it reduces coupling between modules, since the interfaces don't have to specify who owns memory allocations when. It improves performance, usually (although it reduces the predictability of performance).
• The C toolchains are suboptimal; you often have to futz with Makefiles and compiler options, compilation has to be manually invoked and takes long enough that you have to wait for it, and when you make a mistake in your Makefile, you often end up with a program that crashes or otherwise mysteriously misbehaves until you `make clean; make`.

Topics

• Programming (p. 3658) (286 notes)
• C (p. 3359) (28 notes)

Randomizing delta-sigma conversion to eliminate aliasing

Kragen Javier Sitaker, 2014-04-24 (7 minutes)

Straightforward PDM (Pulse Density Modulation) is basically the Bresenham line drawing algorithm --- you bump up an error counter by the desired slope every iteration, and when that error counter hits a threshold, you move to the next discrete row of pixels on the display, and subtract from the error counter. Like this:

```
error_counter += desired_slope * 2 * threshold;
output_value = (error_counter > threshold);
if (output_value) error_counter -= 2 * threshold;
```

As `desired_slope` increases from 0 to 1, the duty cycle of `output_value` (which is always either exactly 0 or 1) likewise increases from 0 to 1. (And of course in practice you probably want to store `desired_slope * 2 * threshold` in a variable, i.e. prescaling `desired_slope`.) To draw a line, you add or subtract `output_value` to your X or Y coordinate, and always increment or decrement the other coordinate, according to which octant the line is in.

One potential problem with this for, say, audio applications, is that it adds an artifact: a high-power pulse train at a single high frequency, and that frequency can potentially be fairly low; it's highest when you're using PDM to produce a 50% duty cycle, at which point you have a square wave at a frequency of half the "sampling" frequency. At 25% or 75% duty cycle, the fundamental frequency dips to a fourth of the sampling frequency, which is to say two octaves lower; at more extreme duty cycles, the frequency drops reciprocally. At a 1% or 99% duty cycle, your artifact frequency is 100 times lower than your sampling frequency, more than six octaves lower.

In an environment that can produce subharmonics, for example due to aliasing, even artifacts too high in frequency to perceive can become disturbing by producing strong perceptible subharmonics. Pixel line-drawing is actually just such an environment: lines that cross one another interact nonlinearly, and it's easy for the high-frequency jaggies of Bresenham line-drawing to alias down into large-scale moiré patterns.

But, ideally, in the PDM case, you keep this artifact frequency high enough that you can filter it out easily, restricting the available duty cycles if necessary, in the analog domain if that's where you're ultimately sending the pulse train. But you may not be able to achieve that. Alternatively, you could *spread* the artifact out so that it's less problematic, making it more closely resemble white noise:

```
error_counter += desired_slope * 2 * threshold;
output_value = (error_counter > threshold + r * drand());
if (output_value) error_counter -= 2 * threshold;
```

Here `r` is a parameter that controls how much bandwidth the artifact noise is spread over. At `r == 0` you have exactly Bresenham

line-drawing, and as `r` grows to `threshold` the artifact noise gets spread over, I think, an entire octave, and I think grows slightly. If `r` grows further, it will *increase* the total artifact noise, but it might still become less obtrusive up to some point, by virtue of having lower spectral density.

I suspect that you could probably produce nearly-telephone-quality audio this way with a "sample frequency" of only about 30kHz and an `r` of about `threshold`, giving you a sort of "shaped dither" spreading the 1-bit quantization noise over the upper octave or so of human hearing, where we're relatively insensitive. A 100Hz wave would have 150 cycles positive and 150 cycles negative, so you'd have only at best an ENOB of about 8, and less for higher frequencies; but as long as the quantization noise stuck strictly to an annoying high-frequency hiss, that might not be so bad. By comparison, a 25%-to-75%-duty-cycle pure-PDM at the same sample rate would on occasions produce a high-power frequency-modulated whistle as low as 7.5kHz, which would be extremely noticeable; and an equivalent 7-bit PWM would use a fundamental frequency of 30kHz/128 = 234Hz, right in the heart of the sounds you're trying to reproduce, making it almost completely unusable.

If you're doing this on a microcontroller, such as an Arduino, you might be extremely interested in how many clock cycles the above code needs, since if the timer ISR you're running it in takes too long, you have to lower the sample rate further in order to have time to run other code. The basic 8-bit Bresenham PDM code is something like

```
   r1 := ram[error_counter]
   r2 := ram[scaled_desired_slope]
   r1 += r2
   ram[error_counter] := r1
   jump_if_overflow 1f
   io[output_pin] := 0
   reti
1: io[output_pin] := 1
   reti
```

using the wraparound carry to implicitly subtract `2 * threshold`. The overflow bit (assuming you have an overflow bit) gets set when the byte overflows from positive to negative. If you only have a carry bit (are there CPUs with only a carry bit?), you can use that instead by initializing `error_counter` to the most negative number (e.g. -128) instead of zero.

(As a side note, I don't know why this logic isn't embedded as a PDM-generation circuit in microcontrollers.)

I think that's about 8–16 clock cycles on a RISC machine, so on an Arduino you could probably up to do 1MHz or 2MHz, or more if you can reserve a couple of registers for the ISR so it doesn't have to access RAM. Another three cycles would give you 16-bit `error_counter` and `scaled_desired_slope` values, and maybe something approaching CD-quality audio:

```
   r1 := ram[error_counter]
   r2 := ram[error_counter+1]
   r3 := ram[scaled_desired_slope]
```

```
   r4 := ram[scaled_desired_slope+1]
   r2 += r4
   r1 += r3 + carry_bit
   ram[error_counter] := r1
   ram[error_counter+1] := r2
   jump_if_overflow 1f
   io[output_pin] := 0
   reti
1: io[output_pin] := 1
   reti
```

Adding the random diffusion is probably best done on the Arduino, which has a two-cycle hardware multiplier, with a linear congruential generator whose results are right-shifted to the proper scale. The LCG probably needs at least 16 bits, so you need at least four 8-bit multiplies and four 8-bit additions, an additional 12 cycles. However, this is not the end of it; the comparison complicates things somewhat, because we can no longer depend on a single test of a carry bit. Probably it's best to restrict the `error_counter` range to a subset of its possible values, requiring an actual conditional subtraction, and use an actual subtraction for the threshold test.

If you're thus limited to, say, 300kHz, then at 30kHz you'll be using 10% of the microcontroller's compute cycles for the audio PDM.

(Hmm, I should probably try this out with Python programs generating WAV files before I worry too much more about its efficiency...)

Topics

- Algorithms (p. 3310) (123 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Audio (p. 3331) (40 notes)
- Microcontrollers (p. 3580) (29 notes)
- Arduino (p. 3324) (6 notes)
- Aliasing (p. 3315) (4 notes)

Prolog table outlining

Kragen Javier Sitaker, 2019-07-05 (11 minutes)

I've been thinking again about Darius Bacon's Sniki and the space of CRUD webapp development, which people still seem to be mostly doing using Django, Rails, and similar 2005-vintage designs, despite the popularity of schemaless and schema-lite database systems like MongoDB and Redis.

"Sniki"?

A few years back, Darius Bacon wrote an interesting "Semantic Network Wiki" supporting RDF-like triples, called Sniki. It explained:

> This is basically like a wiki only with typed links. Any page can serve as a link type. Each page lists all the links it's involved in, as source, type, or target. You edit the links in a separate text area.

It was really a free-form database. On any page, you could include a set of property-value pairs, using square brackets to enclose multi-word items:

```
written 2019-03-11
updated 2019-07-05
concerns politics
supersedes [Taxation without representation]
```

Each of these property-value pairs becomes a triple describing the page; for example, if that page were called "Declaration of Independence", we would have triples such as ([Declaration of Independence] concerns politics) and ([Declaration of Independence] supersedes [Taxation without representation]) in the semantic network.

Such a database is only interesting insofar as you can query it, and Sniki had a very easy and powerful way to query:

> [Y]ou can produce tables from database-style queries like this:
> [tabulate recipes recipe ID, ID title Title, ID description Description, ID date Date]
> … In general a query looks like [tabulate clause1, clause2, ...] where each clause is a triple denoting a typed link, with variables capitalized. The table then shows one row for each way of filling in all the variables.

So the query is Prolog-like, and the results of matching the query against the database of triples is displayed as a table. The "recipes" page I wrote on it when Darius had it running read:

> These came from an introduction to Ruby on Rails. I wondered how easy we could make the same sort of thing without 'programming' or a relational database. That example app also had forms for update and templates for the different pages, though.
> [tabulate recipes recipe ID,
> ID title Title,
> ID description Description,
> ID date Date]
> Here's just desserts:
> [tabulate recipes recipe ID,
> ID title Dessert,
> ID is-dessert true]
> Just recipes Beatrice likes:
> [tabulate recipes recipe ID,
> ID title Title,

ID description Description,
ID date Date,
ID beatrice-likes yes]

This rendered as three tables: one with ID, Title, Description, and Date columns, another with ID and Dessert columns, and a third with the same columns as the first but fewer rows.

This page had the following property-value pairs:

```
recipe recipe1
recipe recipe2
recipe new-recipe-for-rohit
```

The page new-recipe-for-rohit had the following property-value pairs:

```
title [hot cocoa]
description [Everyone's favorite]
date [20 April 2005]
is-dessert true
[see also] sniki
beatrice-likes yes
author [Kragen]
```

Being linked from the "recipes" page with a "recipe" property marked it as being a recipe, although an [is recipe] property like the is-dessert property would have worked too. Because it's a dessert and Beatrice likes it, it shows up in all three tables.

Another page explained:

Experiments with Oval from 1994 partly inspired this. It's a hypertext system built around Objects, Views, Agents, and Links, to make it possible for ordinary people to create collaborative apps without 'real' programming. The current to-do list comes partly out of this paper.

Sadly I still haven't read the Oval paper.

Providing an editable spreadsheet view of a query result

A feature of Sniki that was at times annoying was the fact that every variable in the query was visible in the table; the above queries, for example, include an ID column that says meaningless things like "recipe2". However, by the same token, if you added an "add row" button, it would be straightforward to insert the necessary triples into the database, or "assert" them, in Prolog terminology.

This is fairly asymmetric with deletion, since the absence of any one of the triples necessary for a table row would result in deleting the table row; in my recipe example above, deleting either the title, the description, or the date property — as well as the recipe link — would remove the row from the table. In cases like these, you'd probably want deletion to delete the entity and all the links to and from it, as well as perhaps the links that use it as a link type.

This suggests that in some cases you'd want to use a nested query, like a Prolog `findall`, for entities you want to appear in the table even if they lack a certain property — which probably suggests you'd like them to occur only once even if they have more than one value for

the property. Perhaps you could use nested [] for this if you want a lightweight textual syntax:

```
[Invoice for Customer, Invoice date Date, Customer name Name,
    [Invoice line-item Item,
    Item qty Quantity,
    Item price Price,
    Item sku SKU,
    SKU description Description],
    [Invoice payment Payment, Payment amount Amount, Payment date Date],
]
```

These nested queries would appear as nested tables in the table view, optionally with the ability to add, copy, update, and delete rows, or collapse the nested table.

An alternative, nesting-free way to handle nested queries is with a sort of outline view using something very vaguely like Prolog's ! cut operator:

```
[Invoice for Customer, Invoice date Date, Customer name Name
| Invoice line-item Item,
  Item qty Quantity,
  Item price Price,
  Item sku SKU,
  SKU description Description]
```

The idea here is that, interactively, you can navigate around a three-column table with Invoice, Customer, Date, and Name columns, and whatever row you have highlighted in that table, a detail table appears to the right with the Item, Quantity, Price, SKU, and Description columns for all the line items for that invoice. It's like Prolog's "cut" in the limited sense that once you succeed in getting to the right of it (in this case, because the user has selected the row) and exhaust the possibilities there, you don't cross to the right of it again. Unlike Prolog's cut, though, you do keep examining possibilities to the left of it in order to fill up the rest of the table there.

Here's a crude mockup with example data mostly from exampledb.py:

Invoice	Customer	Date	Name
i8032	c8021	2018-02-01	Edward Brooks, Morris Petroleum Enterprises
i8033	c8021	2018-06-07	Edward Brooks, Morris Petroleum Enterprises
i8034	c79474	2018-04-01	Anthony Hill, LHHB Engineering
i8035	c15376	2018-08-08	Janet Parker, Gray–Wright LLC

+

Item	Quantity	Price	SKU	Description
li181312	5	$300.50	sku901	Gray daily glass toaster oven ×
li181313	1	$21.00	sku353	Gray polyester ultrasonic sweater ×
li181314	1		sku759	Gold spandex surgical dress, size L ×
li181315	2	$20.00	sku751	Black spandex electronic boxers, size M

×
+

Ideally you could tag the generated ID columns so they don't display by default and so that an "add" command generates them automatically from some kind of id-generating function, even if Joe Celko *does* think this is a bad way to make a database:

Date Name
2018-02-01 Edward Brooks, Morris Petroleum Enterprises
2018-06-07 Edward Brooks, Morris Petroleum Enterprises
2018-04-01 Anthony Hill, LHHB Engineering
2018-08-08 Janet Parker, Gray–Wright LLC
+
Quantity Price Description
5 $300.50 Gray daily glass toaster oven ×
1 $21.00 Gray polyester ultrasonic sweater ×
1 Gold spandex surgical dress, size L ×
2 $20.00 Black spandex electronic boxers, size M ×
+

Field editing

I've said that record *creation* is straightforward with such a system, but *update* is surprisingly rather complicated. In a database viewer that displays a single SQL table as a table, it's straightforward to know which record to update: you update the database record that corresponds to the table row on the screen. Here, editing fields is a bit trickier.

In the above example query, the Description field in the line-item table came from the SKU entity, that is, the description of the product. In a case like this, it might not be ideal to automatically propagate edits to the SKU (and thus to all the other invoices and perhaps the web page for the product as well); conceivably you'd want the field to display read-only or to be a link to a detail page for the SKU entity itself, so that you could see the scope of the change you were making.

How about the SKU field itself, which links the line-item entity with the SKU entity? Suppose you choose to display the field and then edit it, changing "sku759" to "sku751". There are at least two reasonable candidate results of such an action:

- You could update both the triple (li181314 sku sku759) and the triple (sku759 description [Gold spandex surgical dress, size L]) to use "sku751" instead of "sku759". This would probably need to propagate to all the other occurrences of "sku759" in the system, as well. And, in this case, this would amount to merging two SKUs, since "sku751" already exists (it's the black spandex electronic boxers, size M), which is perhaps a situation the user should be warned about.

- You could update just the triple (li181314 sku sku759), where we originally got "sku759", to point to some other SKU that has a description; this will change the value displayed in the column to the right. Displaying this description during the selection process is probably necessary, too.

Recursion

If you name queries, you could very reasonably refer to a query within itself:

```
explore(Dir) :- Dir contains-file File, File file-size Size | explore(File).
```

With the columnar presentation described above, this is a Miller-columns filesystem explorer like the one in MacOS X, while with a collapsible tree view, it would be a tree-style filesystem explorer.

Programming-by-example query editing XXX

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Databases (p. 3400) (20 notes)
- Prolog and logic programming (p. 3667) (8 notes)

Self replication changes

Kragen Javier Sitaker, 2017-01-16 (5 minutes)

Efficient programmable self-replicating machinery does not need nanotechnology, and it will change economics and human life in many ways that are currently inconceivable.

The first and most obvious change is that it will change the nature of factors of production in a major way. If we analyze factors of production into land, labor, capital goods, information, and energy, it will provide abundant capital goods for any conceivable production process. This will increase the value of land, labor, information, and energy, but of these, information and energy are likely to be abundant.

Because capital goods will be abundant, we will stop counting them as a significant part of our assets. A brewery does not measure its wealth by how much yeast it has on hand, or a farmer measure his wealth by the amount of seed corn he has available to plant. Similarly, we will no longer consider machinery to be a form of wealth — perhaps architecture, either. We will instantiate them when we want them, then recycle them when we no longer do.

For the last three million years or so, accumulating capital goods — whether hand axes, bottle gourds, or Ferraris — has been a defining aspect of human culture. It is part of what it has meant to be human. And now that is going away.

Recycling capacity is becoming a new factor of production. So far our conversion of material into moop has been modest, despite occasional incidents like the New York garbage barges and the EU's e-waste directives. So far, our garbage production — ultimately the same as our production of goods — has been limited to a few tons per year per person. You could dry it out and store it in your basement if you had to. This will no longer be the case; we will have to recycle our moop into something else, something like Earthships or fresh metals.

Our concepts of quality will change. Historically, high-quality capital goods were those that were durable, pleasant to use, easy and inexpensive to repair, broadly applicable rather than overly specific, and augmented our economic productivity by a large factor. Once self-replicating machinery is in play, durability becomes a secondary concern that we can easily trade off against other kinds of merit. Traditionally, a bicycle chain breaker that breaks on the fourth use would be considered crap, but if you have it fabricated on the spot when you need to change the length of a bicycle chain and recycle it when you're done, you don't need much durability, and you don't need it to fit different kinds of chains. Instead you will care about how quickly it can be fabricated and how much energy it costs to fabricate and recycle it; perhaps you will also prefer it to last dozens of uses if you are in a bicycle shop, but you will be willing to trade that off against ease of fabrication.

Many repairs will likely be replaced by recycling, as they are in biological systems like our bones, or often in computer filesystems.

Two particular figures of merit are the exponential growth rate of the machines and the amount of life-cycle cost per machine (or unit

of productivity) measured in factors of production other than capital goods. The exponential growth rate is determined by the replication time — the time for one machine to construct another, replicating itself — and by the durability of the machines. If a machine's MTBF is lower than its self-replication time, the exponential growth rate will be negative; if the MTBF is equal, the growth rate will be zero. But once the MTBF is a few times greater than the self-replication time, further improvements in reliability and durability will have little effect on the population growth rate.

I think a good and plausible target is a 24-hour self-replication time and an 8-replication MTBF.

Once self-replicating machines are capable of producing solar cells with a reasonable EROEI, energy will also cease to be a significant limitation on their growth, leaving only labor, land, and knowledge.

Many jobs that can be done by self-replicating machines can be done at very small scales, perhaps even submicron scales, even if they do not amount to molecular nanotechnology in themselves. This reduces the material resources needed for replication proportionally. For things that can be reduced in this way, land for resource extraction will be a minimal, even negligible, cost; only labor and knowledge will add significant costs.

We should strive to ensure that everyone has access to knowledge and self-replicating machines, so that they will not be reliant on owners of capital goods for jobs, as they have been since the Industrial Revolution.

Topics

Quintic upsampling of time-series with 1½ multiplies per sample

Kragen Javier Sitaker, 2018-10-28 (2 minutes)

If f is a cubic $k_0x^3 + k_1x^2 + k_2x + k_3$, then $f(0) = k_3$ is a linear function of f(-3), f(-1), f(1), and f(3). Furthermore, the coefficients for f(-3) and f(3) are equal, as are the coefficients for f(-1) and f(1). This means that we can cubic-spline interpolate the samples at the midpoints of existing sample intervals using only two multiplications per sample, plus some additions and subtractions.

More specifically, $f(1) = k_0 + k_1 + k_2 + k_3$, and $f(3) = 27k_0 + 9k_1 + 3k_2 + k_3$, and f(-1) and f(-3) are the same but with alternating signs. This gives us this matrix equation:

```
[-27 9 -3 1]       [f(-3)]
[ -1 1 -1 1] k⃗ = [f(-1)]
[  1 1  1 1]       [ f(1)]
[ 27 9  3 1]       [ f(3)]
```

Inverting that matrix with `sympy.Matrix(...).inv()` gives us:

```
[-1/48  1/16 -1/16  1/48]
[ 1/16 -1/16 -1/16  1/16]
[ 1/48 -9/16  9/16 -1/48]
[-1/16  9/16  9/16 -1/16]
```

Of that, the last row is the one we want, which gives us $k_3 = f(0) = (-f(-3) + 9f(-1) + 9f(1) - f(3))/16$.

This means that in fact you don't need much in the way of multiplication: you only ever need to multiply samples by 9 when you're interpolating between them, which amounts to adding x_i<<3 to x_i. And if you're doing multiple iterations of interpolation, you only need to do the multiplication by 9 for the new samples on each iteration; you don't need it for the ones you already multiplied by 9 in the previous pass.

Doing the same exercise for quintics, we get:

```
[-3125 625 -125 25 -5 1]
[ -243  81  -27  9 -3 1]
[   -1   1   -1  1 -1 1]
[    1   1    1  1  1 1]
[  243  81   27  9  3 1]
[ 3125 625  125 25  5 1]

[-1/3840   1/768  -1/384   1/384  -1/768 1/3840]
[  1/768  -1/256   1/384   1/384  -1/256  1/768]
[  1/384 -13/384  17/192 -17/192  13/384 -1/384]
[ -5/384  13/128 -17/192 -17/192  13/128 -5/384]
[-3/1280  25/768 -75/128  75/128 -25/768 3/1280]
[  3/256 -25/256  75/128  75/128 -25/256  3/256]
```

So we get [3 -25 75 75 -25 3]/256, which requires 3 multiplications

per input sample for a single pass, or 1½ in the limit for many passes.

In two dimensions the situation is hairier.

Topics

Lithium battery welder

Kragen Javier Sitaker, 2018-06-21 (updated 2019-01-22) (2 minutes)

810 amps at 3.7 volts is 3000 watts, which is 150 amps at 20 volts, plenty for arc-welding; different 18650 lithium-ion (Li-ion) cells are rated at 10–35 amps, with most being 20 or 30 amps in theory; being cautious and only expecting 5–10 amps out of each cell, you'd need 80–160 cells, quite a reasonable quantity really. I mean a 9×9 array of cells would be 81 cells.

Charge and discharge rates are rated in "C"; 1C is a current that would yield the battery's rated capacity over one hour. 2C and 3C are typical charge rates, but discharge rates range from 2C to 15C, depending on application. Typical capacities range from 1000 to 3000 milliamp hours.

Dimensions are 18.5 mm diameter, 65 mm length, 47 g, so an 11×11 array of 2000 mAh 15C 3.7V cells would weigh 5687 g and deliver 3.6 kA. So uh I guess you could probably go with a quarter of that: a 6×6 array of 2000 mAh 15C 3.7V cells, or using a hexagonal array[0], 37 cells in a 7-across-the-corners hexagon (130 mm). This weighs 1700 g and can deliver 1100 amps at 3.7 volts, 4100 watts. An appropriate output circuit could convert this to 70 amps at striking voltage of 60 volts, then 200 amps at arc-sustaining voltage of 20 volts.

[0] N.concatenate(([1,], 6 * N.arange(1, 10))).cumsum() # [1, 7, 19, 37, 61, 91, 127, 169, 217, 271]

These cells cost about US$10 each, so the whole battery pack would cost about US$370.

Topics

bytecode interpreters for tiny computers

Kragen Javier Sitaker, 2007-09 (61 minutes)

Introduction: Density Is King (With a Tiny VM)

I've previously come to the conclusion that there's little reason for using bytecode in the modern world, except in order to get more compact code, for which it can be very effective. So, what kind of a bytecode engine will give you more compact code?

Suppose I want a bytecode interpreter for a very small programming environment, specifically to minimize the memory needed for a program; say, on a 32-bit microcontroller with 40KiB of program flash, where the program flash size is very often the limiting factor on what the machine can do.

My dumb fib microbenchmark looks like this in Smalltalk:

```
fib: n
        n < 2 ifTrue: [^1]
              ifFalse: [^(self fib: n - 1) + (self fib: n - 2)]
```

And in Squeak bytecode:

```
9 <10> pushTemp: 0
10 <77> pushConstant: 2
11 <B2> send: <
12 <99> jumpFalse: 15
13 <76> pushConstant: 1
14 <7C> returnTop
15 <70> self
16 <10> pushTemp: 0
17 <76> pushConstant: 1
18 <B1> send: -
19 <E0> send: fib:
20 <70> self
21 <10> pushTemp: 0
22 <77> pushConstant: 2
23 <B1> send: -
24 <E0> send: fib:
25 <B0> send: +
26 <7C> returnTop
```

Or, as I translated to pseudo-FORTH, "n 2 < if 1 return then self n 1 - recurse self n 2 - recurse + return".

The metric of goodness for a CPU instruction set is a little different from that for a bytecode interpreter. Bytecode interpreters don't have to worry about clock rate (and therefore combinational logic path length) or, so far, parallelism; they can use arbitrary amounts of storage on their own behalf; they're easier to modify; their fundamental operations can take advantage of more indirection.

Here are some examples of things a bytecode interpreter can do that a hardware CPU might have more trouble with:

• you can have a very large register set (which is more or less what Squeak's VM does, treating local variables as registers) without incurring slow procedure call and return; MMIX suggests how this could be done in hardware as well.
• you could imagine that every procedure could have its own register set (as, perhaps, on the SPARC), and a few of the instructions could access the contents of these registers; again, Squeak's VM does this
• you could have an instruction to create a new preemptively-scheduled thread, perhaps switching between threads every instruction, as in Core Wars or the Tera MTA;
• if the language is object-oriented, you could have a few instructions for calling certain distinguished methods of self, or the first argument, as in the Squeak VM;
• or, as a more general form of the same thing, entering some context might reprogram certain instructions to do some arbitrary thing;
• you can do all kinds of tag tests and dynamic dispatch on fundamental CPU operations, as in the Squeak VM, the LispMs, or Python's bytecode;
• you can support associative array lookups, appending to unbounded-size arrays, and the like, as fundamental machine operations.

Indirect Threaded 16-bit FORTH: Not Great For Density

I don't have a FORTH handy, but I think the definition looks something like this in FORTH:

```
: FIB DUP 2 <  IF DROP 1
  ELSE DUP 1- RECURSE  SWAP 2 - RECURSE + THEN ;
```

which I think, in an indirect-threaded FORTH, compiles to a dictionary entry containing something like this:

```
DUP (2) < (IF) #3 DROP (1) (ELSE) #8 DUP 1- FIB SWAP (2) - FIB + ;
```

That's 18 threading slots, so 36 bytes, plus the overhead of the dictionary structure, which I think is typically 2 bytes for a dictionary that has forgotten the word names. Better than PowerPC assembly (at 96 bytes) but not great, noticeably worse than Squeak.

Naive Lisp: Terrible for Density

What if we interpret fib with a simple Lisp interpreter that walks tree structures? We could define it as follows:

```
(labels fib (n) (if (< 2 n) 1 (+ (fib (- n 1)) (fib (- n 2)))))
```

That's 17 non-parenthesis tokens and 9 right parentheses, for a total of 28 leaf-nodes on the cons tree. That means the tree contains 27 conses, for 54 memory-address-containing cells in interior nodes, probably a minimum of 108 bytes. I conclude that while this program-representation approach is very simple, it takes up a lot of

space. I don't think cdr-coding would help enough, since none of the lists are very long; if you had 9 lists containing 25 pointers and 9 one-byte lengths or one-byte terminators, you still have 59 bytes.

Lua's VM: Four-Byte Register-Based Instructions

According to "The Implementation of Lua 5.0", Lua's virtual machine has been register-based since 2003. They claim that their four-byte register instructions aren't really much more voluminous than stack-based instructions, perhaps in part because they're comparing to stack-based instructions for a single-stack machine that has local variable storage in addition to its stack.

Lua's register-based virtual machine is fairly small: "[O]n Linux its stand-alone interpreter, complete with all standard libraries, takes less than 150 Kbytes; the core is less than 100 Kbytes." They've previously said that the compiler is about 30% of the size of the core, which suggests that the rest of the core, including the bytecode interpreter, is about 70KB.

They mention that it has 35 instructions, which would almost fit in 5 bits of opcode: MOVE, LOADK, LOADBOOL (converts to boolean and conditionally skips an instruction), LOADNIL (clears a bunch of registers), GETUPVAL, GETGLOBAL, GETTABLE, GETGLOBAL, SETUPVAL, SETTABLE, NEWTABLE, SELF, ADD, SUB, MUL, DIV, POW, UNM (unary minus), NOT, CONCAT (string concatenation of a bunch of registers), JMP, EQ, LT, LE, TEST, CALL, TAILCALL, RETURN, FORLOOP, TFORLOOP, TFORPREP, SETLIST, SETLISTO, CLOSE, and CLOSURE.

CALL passes a range of registers to a function and stores its result in a range of registers; this implies that the virtual machine handles saving and restoring of the stack frame. The paper uses the term "register window" to compare it to what the SPARC does.

The comparison instructions skip the next instruction on success.

Here's their example code to show how much better the register machine is:

```
local a, t, i   LOADNIL 0 2 0
a = a + i       ADD     0 0 2
a = a + 1       ADD     0 0 250
a = t[i]        GETTABLE 0 1 2
```

The old stack machine compiled this as follows:

```
PUSHNIL 3
GETLOCAL 0
GETLOCAL 2
ADD
SETLOCAL 0
GETLOCAL 0
ADDI 1
SETLOCAL 0
GETINDEXED 2
SETLOCAL 0
```

It seems that you should be able to compile this on a two-stack machine as NIL NIL DUP >R + 1+ R> NIL GETTABLE, which is 9 instructions instead of 11, and also clearly stupid, since nil is neither a table nor a number. If you could really fit that into 6 bytes, it might be an improvement over the 12 bytes of their current scheme or the 11 bytes of their previous one. It might be better to try more realistic code fragments.

The paper also discusses an interesting implementation of closures, in which captured variables migrate into heap-allocated structures upon function return.

The MuP21 and F21 instruction sets

The MuP21 was implemented in 6000 transistors, including an NTSC signal generator and a controller for external DRAM, so it ought to be possible to emulate its behavior with a fairly small amount of software. Here's the instruction set:

Transfer Instructions: JUMP, CALL, RET, JZ, JCZ Memory Instructions: LOAD, STORE, LOADP, STOREP, LIT ALU Instructions: COM, XOR, AND, ADD, SHL, SHR, ADDNZ Register Instructions: LOADA, STOREA, DUP, DROP, OVER, NOP

COM is complement. The CPU has an A register, accessed with LOADA and STOREA, that supplies the address for LOAD and STORE; I think LOADP and STOREP increment it as well. I think JCZ jumps if the carry bit is zero. (Each register on the stack has its own carry bit; the "21" refers to the 20-bit memory word size, plus the extra bit.)

The F21 had 27 instructions to the MuP21's 24. (Only 23 are listed above, hmm.) They were renamed:

Code	Name	Description	Forth (with a variable named A)
00	else	unconditional jump	ELSE
01	T0	jump if T0-19 is false w/ no drop	DUP IF
02	call	push PC+1 to R, jump	:
03	C0	jump if T20 is false	CARRY? IF
06	RET	pop PC from R (subroutine return)	;
08	@R+	fetch from address in R, increment	R R@ @ R> 1+ >R
09	@A+	fetch from address in A, increment	A A @ @ 1 A +!
0A	#	fetch from PC+1, increment PC	LIT
0B	@A	fetch from address in A	A @ @
0C	!R+	store to address in R, increment R	R@ ! R> 1+ >R
0D	!A+	store to address in A, increment A	A @ ! 1 A +!
0F	!A	store to address in A	A @ !
10	com	complement T	-1 XOR
11	2*	left shift T, 0 to T0	2*
12	2/	right shift T, T20 to T19	2/
13	+*	add S to T if T0 is true	DUP 1 AND IF OVER + THEN
14	-or	exclusive-or S to T	XOR
15	and	and S to T	AND
17	+	add S to T	+
18	pop	pop R, push to T	R>
19	A	push A to T	A @
1A	dup	push T to T	DUP
1B	over	push S to T	OVER

```
 1C push  pop T, push to R                >R
 1D A!    pop T to A                      A  !
 1E nop   delay 2ns                       NOP
 1F drop  pop T                           DROP
```

T is top-of-stack; R is top-of-return-stack; S is the element right under the top of stack. I think @R+ and !R+ are two of the three new instructions; push and pop are probably the other one, since they don't seem to be listed in the MuP21 list.

I'm not sure what +* is for, but I'm guessing it was ADDNZ.

I'm not sure where the else, T0, and C0 instructions jump to; maybe the next address on the operand stack.

Interestingly, there doesn't seem to be a straightforward way to get a "1" onto the stack without using the # instruction, which is annoying because that takes 25 bits of instructions. dup dup -or A! @A+ drop A is another approach at 30 bits, but it clobbers the A register and issues a useless memory reference. dup dup -or com 2* com is another 25-bit approach.

So here's my dumb fib benchmark expressed in F21 code, according to my limited understanding, and without trying to be very clever:

```
fib: dup #-2 + #returnone swap c0 dup #-1 + #fib call
                                swap #-2 + #fib call + ;
returnone: drop drop #1 ;
```

That loses pretty badly on literals; if we assume that # pushes its value immediately and doesn't require any NOPs (e.g. to avoid having multiple # instructions per word) then we have 22 instructions and 7 literals --- 6 words of instructions and 7 of literals, for a total of 32.5 bytes. Not the code density direction I was hoping this would take me!

But it's possible to avoid the redundant literals:

```
fib: dup #-2 dup push + #returnone swap c0 dup #-1 + #fib dup push call
                                        swap pop swap pop + swap call + ;
returnone: pop drop drop drop #1 ;
```

And actually #1 is somewhat redundant with #-2, being its bitwise complement:

```
fib: dup #-2 dup push + #returnone swap c0 dup #-1 + #fib dup push call
                                        swap pop swap pop + swap call + ;
returnone: drop drop pop com ;
```

That makes it 29 instructions but only 4 literals --- 8 words of instructions, 4 of literals, for a total of 12 20-bit words, or 30 bytes. Still worse than the Squeak version on size --- and quite hard to read! And some of the literals are still probably too close together to work on a real machine.

If we were instead using three-instruction 16-bit words with a high bit used to tag literals, we could maybe win a little more.

```
fib: #-2  dup push over  + nop nop  #returnone  swap c0
     dup  #-1  + nop nop  #fib  dup push call
     swap pop swap  pop + swap  call + ;
```

```
returnone: drop drop pop  com ;
```

That's 33 instructions, but the four literals don't count, so 29 instructions or 10 16-bit instruction words, plus four 16-bit literal words. That's 28 bytes, almost the same as the Squeak version, but still worse! And that's with me trying to get clever with the instruction reordering, too.

Now I begin to understand why Chuck Moore was getting to the point where he would repeat FOO twenty times by doing : FOO5 FOO FOO FOO FOO FOO ; FOO5 FOO5 FOO5 FOO5 instead of using a DO loop. Numbers are a real pain on the F21! (But perhaps that's as it should be; programming isn't about numbers, anyway.)

Local Variables: Registers Or Stacks?

Having two stacks removes the need for local argument vectors; you can always shift the variables left and right between the call and return stack, possibly swapping as you go, to get to the values you want. (This could be shortened if there was a "repeat next instruction four times" instruction: >R, >R >R, 4x >R R>, 4x >R, 4x >R >R, 4x >R 4x >R R> R>, 4x >R 4x >R R>, 4x >R 4x >R, and so on; and similar in the other direction.) It wasn't apparent to me which approach would use less code, or whether it would depend on the number of arguments and local variables.

I thought I'd see what the distribution is like in a body of real code, so I ran the following code in Squeak 3.8-6665. (No doubt any Smalltalk programmer could improve it.)

```
gatherMethodStats
        "How common are methods with lots of temps?"
        | totaldict tempdict argsdict update |
        tempdict := Dictionary new.  "Maybe not the best container."
        argsdict := Dictionary new.
        totaldict := Dictionary new.
        update := [:dict :key | dict at: key put: (1 +
            (dict at: key ifAbsent: [0]))].
        Smalltalk allClassesDo: [:class |
            (Array with: class with: class class) do: [:cl |
                cl selectorsAndMethodsDo: [:sel :meth |
                    update value: tempdict value: meth numTemps.
                    update value: argsdict value: meth numArgs.
                    update value: totaldict
                            value: meth numTemps + meth numArgs.
                ]
            ]
        ].
        ^ {'temps' -> tempdict. 'args' -> argsdict. 'total' -> totaldict.}
```

In a fraction of a second, this returned the following (reformatted):

```
#('temps' -> a Dictionary(
    0->18952 1->13665  2->6366   3->3697  4->2301
    5->1492   6->939    7->676    8->426   9->346
    10->196  11->193   12->139    13->99   14->60
    15->47    16->46    17->30    18->15   19->20
```

```
    20->12    21->11    22->15     23->6     24->3
    25->5      26->4     27->3     28->5     32->1
    33->2      37->1     39->1               50->1)
'args' -> a Dictionary(
    0->26114  1->15903  2->4717  3->1712  4->756
      5->309    6->138    7->64    8->37   9->13
       10->8     11->2    12->1    13->1 )
'total' -> a Dictionary(
    0->18952  1->3240   2->11976  3->3128  4->4290
     5->1760  6->1947     7->999   8->983   9->558
     10->521  11->293    12->276  13->155  14->196
     15->115   16->81     17->57   18->52   19->31
      20->43   21->24     22->20   23->11   24->17
       25->9    26->6      27->9    28->7    29->5
       30->6    33->1      35->1    36->1    38->1
       42->1    44->1      46->1    62->1 )
)
```

That's out of 49775 methods; so roughly 95% of these methods have 8 or fewer arguments and temporaries, 90% have 6 or fewer, 75% have 3 or fewer, and 69% have 2 or fewer. That suggests that in a codebase like Smalltalk, it would probably be a marginal cost to use two stacks in the bytecode instead of a local-argument vector.

But probably the methods that have a lot of local variables and arguments are longer, so inefficiency in implementing those methods might cause inefficiency out of proportion to their number. How much does that skew the results? The CompiledMethod class has initialPC and endPC methods which return the bounds of its bytecode, so I changed the code to count bytecodes rather than methods:

```
gatherMethodStats
    "How common are methods with lots of temps?"
    | totaldict tempdict argsdict update |
    tempdict := Dictionary new.
    argsdict := Dictionary new.
    totaldict := Dictionary new.  "Maybe not the best container."
    update := [:dict :key :incr |
        dict at: key put: (incr + (dict at: key ifAbsent: [0]))].
    Smalltalk allClassesDo: [:class |
      (Array with: class with: class class) do:
        [:cl | cl selectorsAndMethodsDo: [:sel :meth || methbytes |
            methbytes := meth endPC - meth initialPC + 1.
            update value: tempdict value: meth numTemps value: methbytes.
            update value: argsdict value: meth numArgs value: methbytes.
            update value: totaldict value: meth numTemps + meth numArgs
                                    value: methbytes.
        ]
      ]
    ].
    ^ {'temps' -> tempdict. 'args' -> argsdict. 'total' -> totaldict.}
```

This counted 1 334 542 bytecodes:

```
'total' -> a Dictionary(
```

```
 0->171811  1->95663  2->182923  3->117718  4->125591
 5->92320   6->92671   7->72908   8->61526   9->49807
10->46568  11->36943  12->27096  13->21759  14->27821
15->17379  16->13309  17->10390  18->9528   19->7659
20->10162  21->5969   22->5238   23->3087   24->5804
25->5229   26->2915   27->3747   28->2551   29->2217
30->3014   33->12     35->405    36->505    38->341
42->357    44->235    46->336    62->1028 )
```

50% of them are defined in a context with 4 or fewer locals and args; 60% with 6 or fewer; 70% with 7 or fewer; 80% with 10 or fewer; 90% with 14 or fewer; 95% with 27 or fewer. That's not quite as encouraging as the raw method counts, but it still suggests that the approach is viable and probably does not need the "4x" instruction I suggested earlier. (Even in a method with 14 local variables, all of which are simultaneously live, with really random access, I think the average distance from the variable you're currently at to the variable you want is only a third of 14, or 4.7.)

Adapting the MuP21 Instruction Set to a Smalltalk-Like System

Maybe I could follow the MuP21's lead and use five-bit zero-operand instructions for a two-stack abstract machine. Probably I should pack them five to a 32-bit word, or three to a 16-bit word; the left-over bits can be used for tagging immediate data in the instruction stream, as in Leong, Tsang, and Lee's MSL16 FPGA-based FORTH CPU.

The appeal of the 5-bit instructions is that, say, my sample fib program could perhaps be expressed in less than 26 bytes, or 13 16-bit words: 39 instructions or 16-bit literals. Can we do that? Clearly it depends on the instruction set. An ideal FORTHish instruction set for the sample dumb fibonacci program would make it simply

```
dup 2 return-1-if-less-than dup 1- recurse swap 2- recurse + ;
```

which is 11 instructions in length, 8 bytes, with 9 distinct instructions. Some of these instructions --- dup, swap, +, and ; --- would clearly be included in any FORTH-like CPU; others --- 1-, return-1-if-less-than, 2, 2-, and recurse --- are less likely. Here's a version with a more likely instruction set:

```
dup 1 swap 2 - negative? conditional-return pop
dup 1- literal(fib) call swap 1- 1- literal(fib) call + ;
```

call, literal, and pop are also almost certain to exist; this version uses additionally only 1, 2, -, negative?, conditional-return, and 1-. It contains 17 non-literal instructions and two literals, so it would be 16 bytes if literals were two bytes.

For this function, we don't really need 2 or - as instructions; "2 -" can be rewritten just as easily as "1- 1-". That brings the required instruction repertoire down to 9 regular instructions, plus literal.

The only dubious instruction in the remaining repertoire is negative?, and it's only dubious because the MuP21 doesn't know about negativity. I think it amounts to testing the carry bit, which is

actually probably a pretty reasonable thing to either have an operation to test or to have conditional-return test.

Following the MuP21/F21 model, maybe we could improve on Squeak's bytecode by avoiding the use of a special space and special instructions for local variables, by avoiding the need for message argument counts (and by supporting multiple return values), and probably by putting references to message selectors inline in the bytecode rather than in a separate literal table. My instance of Squeak currently only has 30474 different message selectors, so 16 bits for the selector identifier would probably accommodate many more years of evolution.

These erasures would not be at the cost of safety --- in Smalltalk, the argument signature of a method is implicit in the selector, and as long as the bytecode compiler was bug-free, whatever bogus method got called would pop the right number of arguments and push a single return value.

Probably the stack manipulation instructions (# dup over push pop nop drop) and the control-flow instructions (call else To Co ret) would stay the same, with the addition of a "send" instruction; it would probably also be good to keep the A register around in some form, as the destination for messages, which implies keeping the A and A! instructions as well, for a total of 14 fixed instructions. The "send" instruction could simply leave the object reference in A during the call, and expect it to be preserved --- an "A push" sequence before clobbering it and a "pop A!" sequence before returning is probably not too much to ask.

Smalltalk's blocks might have a little difficulty in this environment --- to access method-local variables, to answer from their containing method, and to call methods on self; none of these are difficulties in the cases where the compiler inlines the control structure, of course. It is, of course, possible to make them into full-fledged objects, as the abstract semantic models of Smalltalk and Scheme do.

As an alternative to the complete omission of a literal table, a literal table could override on a per-element basis the elements of a default literal table that defined the meanings of most of the instructions; the most common instruction meanings would be at one end of the table, while the literals would override meanings (with messages) starting from the other end. Probably messages to call and constants to push have roughly equal frequencies, in which case we could use the low bit of the instruction to distinguish them. If the stack-manipulation and control-flow instructions are non-overridable at 11 instructions, that gives us a maximum literal-table size of 18 redefinable instructions, which would default to a statically-determined set of the most common constants and messages in the system.

Literals that overflowed the literal table could still be used inline with the # instruction, possibly followed by call or send. If, as previously suggested, the # instruction were merely a high bit in a 16-bit word, the disadvantage relative to a literal-table entry would be fairly small --- more a 15-bit size limitation than anything else. If we trust our byte-compiler (or some Java-like type-inferencing stack-effect verifier), we can probably do type erasure and avoid the overhead of tag bits here, at least for message selectors.

Larger constants can be constructed fairly easily with a sequence of multiple literal words and some combining operation such as (\x y ->

x * 8192 + y).

I said the control-flow operators should stay the same, but probably the To and Co conditional branches aren't quite the right thing for Smalltalk; maybe ifFalse and ifNotNil instead.

Speculative Case Study: An F21-like Squeak

It might be worthwhile to profile the selector and constant usage of, say, Squeak, to see what the 18 default literals would be, how many literals are used more than once in a method (and therefore might benefit from being put into a literal table). From that perhaps I could estimate the literal table size of each method in the new regime, and then I could hand-translate a few methods to see if they were smaller.

For now I am going to look at a sort of worst-case: suppose we only had dup, over, push, pop, nop, drop, and literals for stack manipulation; send, jump/else, ifTrue, ifNotNil, ifFalse, and ret for control flow; A and A! for changing the destination of messages; and everything else were done by message sends getting their messages from inline literals (rather than a literal table), with the literals distinguished from instructions by a high bit in a 16-bit word. From results with this worst-case, we can estimate how much better some piece of code would be if all its selectors and constants had bytecodes assigned to them.

That gives us 14 opcodes, so we can stuff them four to a 16-bit word normally, let the high bit be 0 for literals, and make sure that "nop" has its high bit be zero so we can insert it in the instruction stream where necessary.

Furthermore, let's assume that we have to handle all blocks, other than those for ifTrue, ifNotNil, and simple loops, by lambda-lifting. Lambda-lifting means that we turn each block into, effectively, an object class; when we instantiate that class, we send it the variables over which the block is closed, and it stores them in instance variables.

If the block modifies the variables, we will have to extract their current values from the block thenceforth, since without further control-flow analysis there's no way to tell when the block might be invoked by some apparently unrelated message send.

Here's a method chosen at pseudorandom, CompiledMethod>>copyWithTrailerBytes: bytes.

```
| copy end start |
start := self initialPC.
end := self endPC.
copy := CompiledMethod newMethod: end - start + 1 + bytes size
            header: self header.
1 to: self numLiterals do: [:i |
    copy literalAt: i put: (self literalAt: i)].
start to: end do: [:i | copy at: i put: (self at: i)].
1 to: bytes size do: [:i | copy at: end + i put: (bytes at: i)].
^ copy
```

In Squeak's bytecode, with suggested bytecode translations

interspersed, and a display of the two stacks separated by a : after a .

```
    37 <70> self
    38 <D0> send: initialPC

#initialPC send  \ bytes start : retaddr

    39 <6B> popIntoTemp: 3
    40 <70> self
    41 <D1> send: endPC

#endPC send  \ bytes start end : retaddr

    42 <6A> popIntoTemp: 2
    43 <43> pushLit: CompiledMethod

#CompiledMethod push  \ bytes start end : CompiledMethod retaddr

    44 <12> pushTemp: 2
    45 <13> pushTemp: 3
    46 <B1> send: -

A push  \ bytes start end : self CompiledMethod retaddr
A! \ bytes start : self CompiledMethod retaddr
dup push \ bytes start : start self CompiledMethod retaddr
#- send  \ bytes end-start : start self CompiledMethod retaddr

    47 <76> pushConstant: 1
    48 <B0> send: +

A push \ bytes end-start : end start self CompiledMethod retaddr
A! #1 #+ send \ bytes end-start+1 : end start self CompiledMethod retaddr

    49 <10> pushTemp: 0
    50 <C2> send: size

over A!
#size send  \ bytes end-start+1 bytessize : end start self CompiledMethod ret..

    51 <B0> send: +

push A! pop #+ send \ A=bytes; end-start+1+bytessize : end start self Com...

    52 <70> self

pop pop A pop A! \ A=self; es1b end start bytes : CompiledMethod retaddr

    53 <D4> send: header

#header send  \ A=self; es1b end start bytes selfheader : CompiledMethod ret...

    54 <F2> send: newMethod:header:
    55 <69> popIntoTemp: 1

A push  push push push push  A! \ A=es1b; : end start bytes selfheader self C..
```

```
pop pop pop A  \ end start bytes es1b : selfheader self CompiledMethod retaddr
pop pop pop A! \ A=CompiledMethod; end start bytes es1b selfheader self : ret..
push #newMethod:header: send \ end start bytes copy : self retaddr

    56 <70> self
    57 <D5> send: numLiterals

pop A! #numLiterals send  \ A=self; end start bytes copy numlits : retaddr
```

So far, we're at 46 pseudo-FORTH operations and 11 literals (10 distinct), or about 46 bytes of this "worst-case" code, nearly half of which is literals. That compares poorly to Squeak's 21 bytes up to this point; even if all the literals in our pseudo-FORTH were instructions, Squeak's bytecodes would still be slightly smaller up to this point! (Not counting the Squeak method's 32-byte literal table, most of which is for the part of the method we haven't gotten to yet.)

Squeak doesn't seem to be getting a big space advantage from its literals table, since none of the literals have been repeated yet (except for #+, which would probably be an opcode in either case).

If I could evaluate subexpressions of a method call in an arbitrary order, the above might be smaller (I could avoid "push push push push push"), but I wouldn't count on it.

This method is at the median of about four parameters and named temporaries, but it also has to deal with unnamed temporaries.

Now we're about to start a loop, from 1 to numlits. The last method send in the loop is to "copy", so we're going to arrange to have it in the A register when we enter the loop as well.

```
    58 <6D> popIntoTemp: 5
    59 <76> pushConstant: 1
    60 <6C> popIntoTemp: 4

\ A=self; end start bytes copy numlits : retaddr

push #1  \ A=self; end start bytes copy : 1 numlits retaddr
A push A! \ A=copy; end start bytes : self 1 numlits retaddr
pop \ A=copy; end start bytes self : 1 numlits retaddr

    61 <14> pushTemp: 4 ; loop starts here

\ A=copy; end start bytes self : i numlits retaddr

    62 <15> pushTemp: 5
    63 <B4> send: <=

A pop dup A! \ A=i; end start bytes self copy i : numlits retaddr
pop dup #<= send \ A=i; end start bytes self copy i numlits stillgoing : retaddr

    64 <AC 0D> jumpFalse: 79

#79 ifTrue  \ A=i; end start bytes self copy i numlits : retaddr

    66 <11> pushTemp: 1
    67 <14> pushTemp: 4
    68 <70> self
```

```
    69 <14> pushTemp: 4
    70 <E7> send: literalAt:

push push push A! \ A=self; end start bytes : copy i numlits retaddr
pop pop dup \ A=self; end start bytes copy i i : numlits retaddr
#literalAt: send \ A=self; end start bytes copy i selfati : numlits retaddr

    71 <F6> send: literalAt:put:

A push push push A! \ A=copy; end start bytes : i selfati self numlits retaddr
pop dup pop #literalAt:put: send \ A = copy; end start bytes i trash : self nu..

    72 <87> pop

drop

    73 <14> pushTemp: 4
    74 <76> pushConstant: 1
    75 <B0> send: +
    76 <6C> popIntoTemp: 4

#1 #+ send  \ A=copy; end start bytes i+1 : self numlits retaddr
```

Now we have to get the stack back to the state for starting the loop, which turns out to be more work than I'd like:

```
A push A!  \ A=i+1; end start bytes copy : self numlits retaddr
pop A  \ end start bytes copy self i+1 : numlits retaddr
push push A! \ A=copy; end start bytes : self i+1 numlits retaddr
pop  \ A=copy; end start bytes self : i+1 numlits retaddr

    77 <A3 EE> jumpTo: 61

#61 else
```

So here we are at the end of the first loop. 44 more ordinary instructions (22 bytes), plus 8 literals. Squeak, by contrast, did the whole loop in just 21 bytes. Again, even if all the literals went away, Squeak's bytecode design would still be tighter.

It might be possible to do a better job of arranging things on the stack so that the computation feels less like programming a Turing machine --- run over here to fetch that, run back there to put it down --- and it seems like there's probably an initial state for the loop that doesn't require 9 instructions to re-establish it at the end.

Still, if there were any code where we'd expect the two-stack machine to shine, it would be stuff like this --- where we only have three variables (and a loop limit) accessed inside the loop.

I also made a bunch of mistakes, but I don't think they undermine my basic conclusion: the two-stack machine design is not density-competitive with a design with a local-variable vector.

Here's the rest of the Squeak bytecode, which I haven't bothered to translate:

```
79 <13> pushTemp: 3
80 <6C> popIntoTemp: 4
```

```
81 <14> pushTemp: 4
82 <12> pushTemp: 2
83 <B4> send: <=
84 <AC 0D> jumpFalse: 99
86 <11> pushTemp: 1
87 <14> pushTemp: 4
88 <70> self
89 <14> pushTemp: 4
90 <C0> send: at:
91 <C1> send: at:put:
92 <87> pop
93 <14> pushTemp: 4
94 <76> pushConstant: 1
95 <B0> send: +
96 <6C> popIntoTemp: 4
97 <A3 EE> jumpTo: 81
99 <10> pushTemp: 0
100 <C2> send: size
101 <6D> popIntoTemp: 5
102 <76> pushConstant: 1
103 <6C> popIntoTemp: 4
104 <14> pushTemp: 4
105 <15> pushTemp: 5
106 <B4> send: <=
107 <AC 0F> jumpFalse: 124
109 <11> pushTemp: 1
110 <12> pushTemp: 2
111 <14> pushTemp: 4
112 <B0> send: +
113 <10> pushTemp: 0
114 <14> pushTemp: 4
115 <C0> send: at:
116 <C1> send: at:put:
117 <87> pop
118 <14> pushTemp: 4
119 <76> pushConstant: 1
120 <B0> send: +
121 <6C> popIntoTemp: 4
122 <A3 EC> jumpTo: 104
124 <11> pushTemp: 1
125 <7C> returnTop
```

You could make the argument that the abstract machine Smalltalk presents to the user is more like a register machine than a stack machine, and that this may account for the code being awkward when you translate it to a stack machine. If I were programming this originally in a Forth dialect, I probably would structure the code a little differently, but I doubt it would make that much of an improvement in the code size, unless we used some kind of auxiliary non-stack storage for local variables --- at which point we're pretty much back to Squeak bytecode.

Steve Wozniak's SWEET 16 Dream Machine

Steve Wozniak's SWEET16 16-bit virtual machine, included as part of Integer BASIC, supposedly doubled the code density of the 6502. The virtual machine itself was 300 bytes of 6502 assembly, implementing these instructions; here "#" means "[0-F]".

```
0x1# SET: load immediate                0x2# LD: copy register to accumulator
0x3# ST: copy accumulator to register   0x4# LD: load byte indirect w/ increment
0x5# ST: store byte indirect w/incr     0x6# LDD: load two bytes ind w/incr
0x7# STD: store two bytes ind w/incr    0x8# POP: load byte indirect w/predecr
0x9# STP: store byte ind w/predecr      0xA# ADD: add register to accum
0xB# SUB: subtract register from acc    0xC# POPD: load 2 bytes ind w/predecr
0xD# CPR: compare register w/acc        0xE# INR: increment register
0xF# DCR: decrement register            0x00 RTN to 6502 mode
0x01 BR unconditional branch            0x02 BNC branch if no carry
0x03 BC branch if carry                 0x04 BP branch if positive
0x05 BM branch if minus                 0x06 BZ branch if zero
0x07 BNZ branch if nonzero              0x08 BM1 branch if -1
0x09 BNM1 branch if not -1              0x0A BK break (software interrupt)
0x0B RS return from sub (R12 is SP)     0x0C BS branch to sub (R12 is SP)
```

0x01–0x09 and 0x0C have a second byte which is a signed 8-bit displacement. If you want a 16-bit jump, you can push it on the stack and RS.

That's it, 28 instructions, 300 bytes of machine code to implement them. And I thought the 6502 was already reasonable on code density, so this was apparently quite a win.

It's pretty terrible compared to Squeak's bytecode, though. I think our fib microbenchmark should do fine, since it's all arithmetic and local jumps. Let's assume a calling convention that puts the first argument in R0 and returns the return value in R0. (I don't care where other arguments go; they can go hang, because this function only has one.) Here's my first attempt, which may be buggy:

```
FIB   DCR R0    ; subtract 2 by decrementing twice
      DCR R0
      BM BASE   ; if it was <2, go to the base case
      INR R0    ; re-increment, so R0=n-1
      STD @R12  ; save a copy of n-1 on the stack
      BS FIB    ; recurse; now R0=fib(n-1)
      ST R4     ; save fib(n-1) so we can retrieve n-1
      POPD @R12 ; now we have n-1 in R0
      ST R3     ; stick n-1 in R3 so we can use R0 to save fib(n-1) on stack
      LD R4     ; now R0=fib(n-1) again
      STD @R12  ; and we push fib(n-1) on the stack
      LD R3     ; now R0=n-1
      DCR R0    ; now R0=n-2
      BS FIB    ; now R0=fib(n-2)
      ST R3     ; we have to get it out of the way so we can pop fib(n-1)
      POPD @R12 ; great, R0=fib(n-1) and R3=fib(n-2)
      ADD R3    ; now R0 = fib(n)
      RS        ; return
BASE  SUB R0    ; base case: R0=R0-R0=0
      INR R0    ; increment
      RS        ; return
```

That's 21 instructions, three of which have parameter bytes, so 24 bytes. It may be possible to cut this by a couple of bytes, but not more, so it's not really a win over Squeak's system. But it's not a huge loss. (As I said, the code may be buggy, but it's probably good enough for size estimation.)

Suppose we were trying to translate CompiledMethod>>copyWithTrailerBytes: bytes from earlier. You could imagine starting like this, with self in R1 and bytes in R2, and a calling convention that requires us to preserve all registers, including arguments (but, naturally, lets us use the stack), except for R0.

```
CWTB  LD R3          ; We don't have to save anything to preserve it
      STD @R12       ; from the call, because of the calling
                     ; convention, but we do need a place to keep the
                     ; return value.
      BS *+1         ; These four instructions, 7 bytes, are a far call.
      SET INITIALPC
      STD @R12
      RS
      ST R3          ; store return value in R3 (start)

      LD R4          ; now clear out R4 to receive "end"
      STD @R12
      BS *+1
      SET ENDPC
      STD @R12
      RS
      ...
      POPD @R12
      ST R4
      POPD @R12
      ST R3
      RS
```

We're only two lines of code into the method, and we're already at 24 bytes, where the Smalltalk had used six bytecodes and two literals for a total of 14 bytes; and that's glossing over the issue of polymorphic sends for now, assuming that you could compile each "virtual function" into a real function that you could far-call. "end - start + 1 + bytes size", if we write it monomorphically for 16-bit integers, looks something like this:

```
  LD R7       ; clearing out another reg
  STD @R12
  LD R4       ; end
  SUB R3      ; - start
  INR R0      ; + 1
  ST R7
  LD R8       ; a temp slot for expression result
  STD @R12
  LD R1       ; also we have to change "self"
  STD @R12
  LD R2       ; bytes
  ST R1       ; self <- bytes
  BS *+1
```

```
SET SIZE
STD @R12
RS
ADD R7      ; pedantically, this is "bytes size + (end - start + 1)"
ST R8
```

That's 18 instructions plus three parameter bytes, for 21 bytes. Squeak's version was from byte 44 to byte 51, 8 bytecodes, referring to no literals. Unsurprisingly, I guess, SWEET 16 was roughly equivalent on "fib", but much worse on more realistic code.

NanoVM: Java Bytecodes on the AVR

NanoVM is an AVR implementation of Java bytecode; it is about 7100 bytes of AVR machine code and includes garbage collection, arithmetic, inheritance, presumably polymorphism, and needs about 400 clock cycles per Java bytecode, plus 256 bytes of RAM for the VM. However, it only supports "a small subset of the Java language", without "exceptions, threads, floating point arithmetic and various other things like e.g. inheritance from native classes."

As I posted previously, the Java bytecode instruction format looks like it's in the same ballpark with Squeak's, but the bytecode file format may have some hefty overhead; adding a second copy of the "fib" method to Fib.java, under a different name, inflated the .class file by 82 bytes, from 577 bytes to 659, even though javap -c only shows 15 bytecode instructions occupying 21 bytecode slots in the method.

So it's probably possible to fit a bytecode engine similar to Squeak's into 8 kilobytes of ROM, but 4 kilobytes may be pushing it. Two orders of magnitude performance loss is heavy but may be acceptable.

Code-Compact Data Structures

If you want a flexible but painfully slow language to run on a machine without much code space, you probably need some built-in way to represent common data structures so that you don't have to implement them yourself in your user-level code. The usual set present in modern high-level languages (JavaScript, Python, Lua, Perl) includes numbers, (generally immutable) strings, dictionaries, and mutable, growable lists or arrays.

Symbols, as in Lisp or Smalltalk, are probably a very useful optimization in this setting; they allow you to throw away the keys to your dictionaries if you never print them out.

You may be able to save code space by implementing strings as arrays or lists of character or integer objects, but the run-time space cost is terrible; this may be OK if you never or rarely have strings.

Lua's dictionary ("table") implementation uses some hashing technique I don't understand to be able to operate with a load factor of 100%; I don't know how much code it needs. FORTH's dictionary structure is probably the simplest efficient growable dictionary structure: an eight-entry hash table with separate chaining. If you have space-efficient resizable arrays, you could perhaps store each chain in one of those instead.

Here's a working implementation in my pidgin OCaml of these growable arrays and hash tables. I wrote it in OCaml because I don't have an assembler handy, that's the only language implementation I

have handy that produces reasonably compact assembly code, and I haven't written enough code in any assembly language to be able to write this in assembly from memory. It's 42 lines of OCaml code and comes out to 477 instructions, and it omits only two necessarily polymorphic sends: one for hashing in get_table, and one for equality testing in hsearch. I'm pretty dissatisfied with the number of instructions there --- I feel like I could do better, by a factor of 2 or 3 --- but the example at least provides an upper bound that doesn't look insane.

```
(* code-compact data structures. *)

(* The point of this file is not to provide data structures you'd want
   to use in OCaml (OCaml provides other data structures) but to see
   how much assembly code they compile to. *)

(* growable array *)
type 'a ary = { mutable a: 'a array; mutable n: int;
                mutable allocated: int } ;;
exception Out_of_bounds of int ;;
(* emptyary: 21 PowerPC instructions *)
let emptyary () = { a = [||]; n = 0; allocated = 0 } ;;
(* aryappend: 121 PowerPC instructions *)
let aryappend a i =
  (if a.n = a.allocated then
      let newalloc = a.allocated * 2 + 1
      in let newary = Array.make newalloc i
      in (
          (* normally we would use Array.blit here, but the point is
          to count instructions *)
          for i = 0 to a.n - 1 do newary.(i) <- a.a.(i) done ;
          a.a <- newary ;
          a.allocated <- newalloc
      )
  ) ;
  a.a.(a.n) <- i ;
  a.n <- a.n + 1
;;

(* boundscheck: 30 PowerPC instructions *)
let boundscheck a n =
  if n < 0 || n >= a.n then raise (Out_of_bounds n)
  else () ;;
(* aryat: 40 PowerPC instructions *)
let aryat a n = boundscheck a n; a.a.(n) ;;
(* aryatput: 40 PowerPC instructions *)
let aryatput a n i = boundscheck a n; a.a.(n) <- i ;;

(* end of growable array code, totaling 252 PowerPC instructions,
   which I think is probably 1008 bytes of machine code. *)

(* hash table.  Specialized for integer keys because OCaml doesn't
   support polymorphic sends to integers. *)

type 'a hashtable = (int * 'a) ary array ;;
```

```
exception Key_not_found of int ;;
(* hashint: 2 PowerPC instructions *)
let hashint i = i land 7 ;;
(* hsearch: 41 PowerPC instructions *)
let rec hsearch tbl k i =
  if i = tbl.n then raise (Key_not_found k)
  else let kk, v = aryat tbl i
    in if kk = k then i else hsearch tbl k (i+1) ;;
(* get_table: 35 PowerPC instructions *)
let get_table h i = h.(hashint i) ;;
(* hashput: 53 PowerPC instructions *)
let hashput h i nv =
  let tbl = get_table h i and newpair = (i, nv)
  in try let pos = hsearch tbl i 0 in aryatput tbl pos newpair

    with Key_not_found _ -> aryappend tbl newpair ;;
(* hashget: 17 PowerPC instructions *)
let hashget h i =
  let tbl = get_table h i
  in let (_, v) = aryat tbl (hsearch tbl i 0)
  in v ;;
(* hashhaskey: 32 PowerPC instructions *)
let hashhaskey h i =
  try ignore(hashget h i); true with Key_not_found _ -> false ;;
(* newtable: 45 PowerPC instructions *)
let newtable () =
  let rv = Array.make 8 (emptyary ())
  in for i = 1 to 7 do rv.(i) <- emptyary () done ; rv ;;
(* end of hash table code, totaling 225 PowerPC instructions,
   which I think is probably 900 bytes of machine code. *)
```

Here's a pidgin Squeak version of just the resizeable array:

```
'From Squeak3.8 of ''5 May 2005'' [latest update: #6665]'!
Object subclass: #Tinyarray
        instanceVariableNames: 'a n allocated'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'My stuff'!
!Tinyarray commentStamp: '<historical>' prior: 0!
Tiny OrderedCollection to see how small we can make stuff.!

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:32'!
add: i
        "max bytecode = 59"
        n = allocated ifTrue: [| newalloc newary |
                newalloc := allocated * 2 + 1.
                newary := Array new: newalloc.
                1 to: n do: [:ii| newary at: ii put: (a at: ii)].
                a := newary.
                allocated := newalloc.
        ].
        a at: (n+1) put: i.
        n := n + 1.! !
```

```
!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:33'!
at: nn
	"max bytecode = 16"
	self boundscheck: nn.
	^ a at: nn.! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:33'!
at: nn put: i
	"max bytecode = 17"
	self boundscheck: nn.
	^ a at: nn put: i.! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:37'!
boundscheck: nn
	"max bytecode = 43"
	(nn < 1 or: [nn > n]) ifTrue: [Error new
		signal: 'subscript out of bounds: ', nn printString].
! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:34'!
initialize
	"max bytecode = 20"
	a := {}.
	n := 0.
	allocated := 0.! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:36'!
size
	"no bytecode --- quick-return field 1.
	Would probably be max bytecode = 6 otherwise."
	^ n.! !
```

This puts all the code for the class (except its instance variable list) in 161 bytes, including ten four-byte literals, so we'd be down 20 bytes if the literals were 16-bit instead. This is noticeably less than the probably 1008 bytes OCaml wanted for essentially the same code. The corresponding Tinyhash is 301 bytes of methods, including 65 (20%!) for the 'keys' method I left off of the OCaml version and 100 bytes of literal pointers. As variants, I did a version using no hashing at all, just Associations in a Tinyarray, which was 199 bytes, a version that used parallel arrays instead of Associations, which was 329 bytes (9.3% larger). A minimal variant of the Association class is 31 bytes (of methods) so I suspect that including Associations is probably a win --- they don't have to save much code anywhere else to be an absolute win.

(I also wrote a version that was just an alist, without Associations or hashing, which was 131 bytes, and a hash table wrapped around that, which was 154 bytes plus a 27-byte Tinyarray>>#do: method, which all adds up to be 312 bytes, slightly larger than the non-alist-based hash-table version; but Tinyarray>>#do: is likely to be useful in other contexts as well.)

(All of this code depends on very little from Array --- it doesn't fall back on its bounds-checking at all, doesn't query it for its size, just

uses #at: and #at:put: and allocates new arrays of fixed size --- so it should be able to run atop a very primitive Array implementation.)

Library Design

Historically people have often made the mistake of thinking that computers were for computing --- that is, arithmetic, with numbers. But the number-handling in many programs is confined the lower levels.

Consider these numbers, again from Squeak:

method name	calling methods
sinh	0
ln	13
tan	15
squared	40
raisedTo: (a power)	55
signal: (raising an error)	78
sin	78
-> (creating a pair)	114
signal (raising an error)	119
on:send:to: (metaprograms)	117
\\ (integer modulo)	263
addMorph: (GUI design)	333
next (stream input)	493
bitAnd: (bit twiddling)	609
value (invoking a thunk)	674
notNil (testing for nil)	702
collect: (mapcar)	862
// (integer divide)	912
nextPut: (stream output)	968
add: (incremental constr)	1055
* (multiply)	1919
at:put:	1990
<= (numeric comparison)	2186
@ (creating a pair)	2372
, (sequence concatenation)	2399
do: (sequence iteration)	2508
at: (collection indexing)	3290
- (arithmetic)	3684
new (instantiating class)	5150
+	5186
= (comparison)	5342

This is out of about 50 000 methods.

Transcendental functions are really unpopular. Most of the math isn't all that popular, in fact, compared to things like operations on sequences. #+ is an exception, but it's used for several things besides arithmetic on numbers: sound mixing, pointer arithmetic, color mixing, date/time manipulation (which is arguably numerical), animation compositing, and voice composition. I suspect that the surprising popularity of #- is due to Smalltalk's unfortunate decision to index its collections from 1 and use closed intervals everywhere, resulting in lots if "-1" in otherwise clean, arithmetic-free code.

The methods for things like iteration and conditional testing are probably more widely used than any of the above, but the compiler

inlines them, so I can't get statistics easily.

In general, a system that provided no arithmetic *at all* would be of limited use, but it's possible to get surprisingly far without floating-point or transcendental functions, let alone complex numbers; you probably get +, -, and < almost for free. It's likely that you'd be better off devoting precious ROM space to some kind of flexible collection classes than to transcendental functions, rational numbers, or possibly even division.

Other Existing Small Interpreters

• S21

Jeff Fox's S21 simulator for the MuP21 microprocessor (1995-1998) is a 187KB MS-DOS EXE file. It includes the simulator for the processor, a a single-stepping debugger with an MS-DOS console user interface, and it itself is running inside the FPC FORTH virtual machine, which also includes an interactive development environment with a program editor, a virtual memory system, and cooperative multitasking.

• Squeak?

On this Intel Mac, the Squeak virtual machine binary is 967868 bytes. I don't understand how that's possible, since I thought it was built from just the Interpreter and ObjectMemory classes, which are only 10 000 lines of code in total, about the same as the Lua interpreter.

• pepsi/Albert/the golden box

Ian Piumarta's "pepsi" system, the lowest-level substrate for which Alan Kay's NSF-funded reinvention-of-programming system, is 144 lines of C code and compiles to 1451 bytes, or 1602 bytes with inline caches enabled. It provides very efficient, very dynamic method dispatch, and a minimal object system, but no bytecode interpreter.

• The Basic STAMP

The Basic STAMP uses a sort of bytecode for executing BASIC on a PIC; most of the variable-length instructions are less than a byte long. Chuck McManis's 1994 article "Decoding the BASIC Stamp" describes them. They're really little more than a tokenized BASIC. I have no idea how big the STAMP ROM is.

• UCSD P-System

The UCSD P-System, a stack-based bytecode interpreter with compilers from Fortran and Pascal, ran on a lot of small microcomputers back in the day. Z8080:INTERP.TEXT, the UCSD Pascal Interpreter for the Z-80 and Intel 8080A by Peter A. Lawrence and Joel J. McCormack, is about 6100 lines of 8080 assembly when you include all the things it .INCLUDEs. (I found this in a file called I5Z80Interp.TXT.) This includes floating-point math, set arithmetic, transcendental functions, booting from disk, character terminal addressing, single-stepping, virtual memory, a tokenizer for Pascal, binary search trees, I/O interfaces for CP/M, and all kinds of such nonsense. However, I don't have an 8080 assembler handy on this Mac, and there are a fair number of macros (on one hand) and conditional compilation (on the other), so I'm not sure how big that actually ends up being. It's 5000 non-comment non-blank lines.

The PDP-11 interpreter ("mainop.mac" or "I.5-PDP-Interp.TXT")

seems to be just the bytecode interpreter itself, and it's only around a thousand lines of PDP-11 assembly.

The virtual machine instructions listed in the assembly are quite similar to the set of P-Code instructions in Steven Pemberton and Martin Daniels's book, "The P-Code Machine" and also Jensen and Wirth's 1973 "ASSEMBLER AND INTERPRETER OF PASCAL CODE", (PROGRAM PCODE(INPUT,OUTPUT,PRD,PRR)). Said Jensen and Wirth code is 775 non-comment non-blank lines of Pascal, which suggests very roughly about 4000 assembly instructions.

• PICBIT

PICBIT is Feeley and Dubé's bytecoded Scheme implementation for PIC microcontrollers. It uses a register-based virtual machine with six registers for object references, plus PC and number-of-args registers; it has 17 instructions, which are inadequately explained in their paper, "PICBIT: A Scheme System for the PIC Microcontroller," but which reflect a very Schemey view of the world, with a "continuation" register, a CALL instruction that's really JMP-and-adjust-nbargs, and so on. One of their example programs was 38 lines of Scheme (about 1000 bytes), which compiled to 2150 bytes of bytecode. Larger programs were inflated by less than half as much, but they were still much larger than I would expect with other bytecode systems.

I venture to guess that this suggests that their register-based virtual machine bytecode was a memory hog. Dubé's earlier BIT bytecode system, after which they modeled their system, used stack-based bytecode, which was less than half as big on their five example programs.

However, the entire R4RS Scheme library only took 11248 bytes of bytecode, so it seems likely that you can get a relatively powerful set of primitives into not very much space with the general approach of using bytecode. With BIT, it was under 8000 bytes.

They report 37000 bytecodes per second, but it's not clear whether they mean on a 10MHz PIC microcontroller or extrapolated to a 40MHz microcontroller.

• F-83

F-83 was an indirect-threaded FORTH programming system available on most microcomputers in 1983, at a time when many of them had 64kiB of total address space. I have the F-83 books somewhere, but I don't have a copy of the sources here, and I don't know how big the whole system was, but I think it was normally all in memory at once --- the assembler, the FORTH compiler, the indirect-threaded-code interpreter, the interactive interpreter, the full-screen editor, the decompiler, and the virtual-memory system (which couldn't be effectively used for code).

• Various combinator-graph-reduction machines

I haven't looked very much at these, but I suspect that laziness may reduce program size. (Perhaps the same thing could be said of backtracking and concurrency.)

Conclusions: Squeak Rules, We Should Be Able To Do Python In A Few K

Most of the approaches I looked at are considerably more compact

than PowerPC machine code on the toy "fib" problem, by a factor of 2–4, with Squeak among the best; this includes Java (?), Squeak, indirect-threaded 16-bit FORTH, SWEET 16, and the F21 CPU. A couple (OCaml bytecode, PICBIT) sound much worse on density. Beating a factor of 4 (25 bytes for the "fib" program) looks difficult. The factor-of-4 improvement in code density looks realistic from the Tinyhash and Tinyarray examples, and based on previous work, I think a virtual machine for some Squeak-like bytecode can be contained in 4000–8000 bytes of machine code, with a rich library requiring a few thousand more bytes.

Probably Squeak's approach, using a stack for expression intermediate results (to keep instruction size down), a local-variable vector for slightly-longer-term usage (to cut down on stack-manipulation noise words), and a local-literal vector for constants and linkage, with nearly all instructions contained in a single byte, is the best one known for tight bytecode. I suspect that conditional return may be a more compact control-flow primitive than conditional jump, but it could make compiler implementation more challenging.

I'm going to assert that polymorphism, even (especially!) on "fundamental" operations that have machine instructions assigned to them, increases code density. In the case that you don't have any polymorphism in your program, it costs you very little code density (none in bodies, maybe a couple of bytes each in definitions), and in the case where you do, it saves you conditionals. This should hold a fortiori for multiple dispatch.

If we are going for absolute minimum run-time code size, it's perhaps best to have a small kernel written in machine code (probably in a stack-oriented fashion, such that you can put CALL instructions one after the other with no intervening setup) that implements a fairly primitive stack-based virtual machine, atop which a more Squeak-like virtual machine is implemented. (They need not be separate abstract machines --- perhaps unimplemented bytecodes will trap into a user-defined instruction handler.) For example, the hash tables and growable arrays mentioned previously should probably be mostly implemented in this level; in Squeak bytecode, they need around 500 bytes.

Library design probably makes a big difference in how few literals you have to use --- if most of the messages in your system belong to a few small interfaces like #at: and #at:put: or arithmetic, you'll have a much easier time with the bytecode.

With this approach, it should be possible to get a very slow language, with flexibility something like Python's, into maybe 2000–6000 bytes of a microcontroller's ROM. This should allow you to interactively get out-of-memory errors with great convenience and flexibility.

Topics

Notations for defining dynamical systems

Kragen Javier Sitaker, 2016-10-03 (updated 2016-10-06) (6 minutes)

Playing with this scientific calculator and thinking about the Mill CPU and write-once memory (such as, roughly, NOR flash), and then spending some quality time with Gnumeric and watching Joel Spolsky and Martin Shkreli coax magic from Microsoft Excel, it occurred to me that there was maybe an interesting and expressive programming model that captures pointwise calculations on vectors as well as generalized prefix sums over them.

The programming model is as follows. You specify a sequence of updates to apparently scalar variables, such as

```
x := y + 1
z := z + x
```

These updates are run, in sequence, repeatedly.

As syntactic sugar, we use → for assignment, put the variable on the right side, make the assignment an expression and return its value from the assignment expression (as in Lisp or C), and support augmented assignment operators as in C, but spelled +→, -→, ·→, and so on. This allows us to abbreviate the above example as follows:

```
y + 1 → x +↦ z
```

By itself, this is just a formalism for deterministic dynamical system evolution rules. But you might want to visualize a trajectory or the trajectories of a dynamical system, such as the famous Minsky circle:

```
x · k +↦ y · k -→ x
```

This allows us to consider the dynamical system as a function from initial values to trajectories, which we could conceptualize as vectors of successive values taken by the different variables. From that point of view, we are no longer overwriting the previous value; we are merely appending to a vector of values.

Note that in particular this expression has a somewhat APLish meaning in this interpretation:

```
x + y → z
```

If x and y are vectors that something else is building at the same time, corresponding items in them will be added and the sums appended to z; if one of them is a scalar (i.e. a value that doesn't change during the loop), then it will be added to each value of the other and the sums appended to z; if they are both scalars, then some undetermined number of copies of the sum will be appended to z.

From this point of view, it is natural to add an operator to look back in history, like Git's ~ operator; `HEAD` is the current value of `HEAD`, while `HEAD~1` is its previous value, and `HEAD~2` is the value before that (equivalent to `HEAD~1~1`). For example, if we initially have two values

in x, we could write linear extrapolation to the next value as follows:

```
2 · x - x~1 → x
```

As a more complex example, we can write the Goertzel algorithm as

```
x + b · s - s~1 → s - c · s~1 → y
```

where b is 2 cos ω_0 and c is exp(-i ω_0). Note that this is potentially confusing in that the expressions s and s~1 occur twice in the same formula with different meanings. It might be less confusing to write this as follows:

```
x + b · s - s~1 → s
s - c · s~1 → y
```

Now, in this form, this is sort of incomplete; if x decays to the latest value added to x, it only implements the Goertzel algorithm if stuff is getting added to x while the above code is running.

There are a couple of different ways we could handle that. One is that we could package the above code into a function, and define a function composition operation that implicitly gloms together the various state transition functions into a single evolution rule. A more conventional alternative would be to provide a while or foreach construct. In this case, due to the one-dimensionality of the temporal constructs described thus far, a foreach construct could pun the sequence with its iterator. If we use @, we get:

```
x @ x + b · s - s~1 → s - c · s~1 → y
```

This in some sense creates a new local variable x inside the scope of the loop (the right side of the operator) that is an alias for the nonempty prefixes of the original x on sequential iterations of the loop.

Augmenting this with a : sequence-construction operator like Octave's (but zero-based), we now have a way to control the number of iterations of something like Minsky's circle algorithm; we can write, for example:

```
1→x→y
:1000@x·k+→y·k-→x
```

However, I think it's useful to consider evolution rules like x·k+→y·k-→x as entities in themselves rather than incomplete code snippets. It may be useful to compose them, perhaps with β-reduction and α-renaming, as part of a larger evolution rule involving more variables, or to attempt to differentiate them with respect to their inputs, or to attempt to find inverses for them, in general or from a particular point.

You could iterate over multiple sequences in parallel; vector dot product can be expressed as follows:

```
0 → t
u, v @ u · v +→ t
```

Nested loops make sense in only a few circumstances without adding a way to store arrays with more dimensions, but for example, here's a time-domain FIR filter convolving `x` with kernel `k`, storing intermediate sums in a variable `t`:

```
:#k → n
x[#k:] @ 0 → t
        k, n @ x~n · k +↦ t
        t → y
```

In an environment with very limited screen space, such as a scientific calculator or an Excel formula, you could write that without indentation, whitespace, or newlines:

```
:#k→n;x[#k:]@0→t;(k,n@x~n·k+↦t);t→y
```

That's 36 characters, which should fit on the screen even on a fairly low-end scientific calculator.

This data model is more convenient in many cases than Excel's, but somewhat less powerful, because it doesn't natively support two-dimensional arrays. But maybe the right thing is actually to have this kind of merged native support for one-dimensional arrays and scalars, but require some kind of extra operation for two-dimensional arrays, the way C and Perl5 require an extra operation to dereference pointers.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Programming languages (p. 3656) (47 notes)
- Calculators (p. 3362) (11 notes)
- Mill (p. 3584) (7 notes)
- Logging (p. 3554) (5 notes)
- Goertzel (p. 3476) (4 notes)
- Minsky algorithm (p. 3586) (3 notes)

Microfinance

Kragen Javier Sitaker, 2007 to 2009 (6 minutes)

James Surowiecki criticizes nanocorps, in "What Microloans Miss":

This vogue has translated into a flood of real dollars: institutional and individual investments in microfinance more than doubled between 2004 and 2006, to $4.4 billion, and the total volume of loans made has risen to $25 billion, according to Deutsche Bank. Unfortunately, it has also translated into a flood of hype. There's no doubt that microfinance does a tremendous amount of good, yet there are also real limits to what it can accomplish. Microloans make poor borrowers better off. But, on their own, they often don't do much to make poor countries richer.

This isn't because microloans don't work; it's because of how they work. The idealized view of microfinance is that budding entrepreneurs use the loans to start and grow businesses—expanding operations, boosting inventory, and so on. The reality is more complicated. Microloans are often used to "smooth consumption"—tiding a borrower over in times of crisis. They're also, as Karol Boudreaux and Tyler Cowen point out in a recent paper, often used for non-business expenses, such as a child's education. It' s less common to find them used to fund major business expansions or to hire new employees. In part, this is because the loans can be very small—frequently as little as fifty or a hundred dollars—and generally come with very high interest rates, often above thirty or forty per cent. But it's also because most microbusinesses aren't looking to take on more workers. The vast majority have only one paid employee: the owner. As the economist Jonathan Morduch has put it, microfinance "rarely generates new jobs for others."

This matters, because businesses that can generate jobs for others are the best hope of any country trying to put a serious dent in its poverty rate. Sustained economic growth requires companies that can make big investments—building a factory, say—and that can exploit the economies of scale that make workers more productive and, ultimately, richer. Microfinance evangelists sometimes make it sound as if, in an ideal world, everyone would own his own business. "All people are entrepreneurs," Muhammad Yunus has said. But in any successful economy most people aren't entrepreneurs—they make a living by working for someone else. Just fourteen per cent of Americans, for instance, are running (or trying to run) their own business. That percentage is much higher in developing countries—in Peru, it's almost forty per cent. That' s not because Peruvians are more entrepreneurial. It's because they don't have other options.

Surowiecki's argument seems to have two parts: first, that the only way to create jobs is to hire people, and second, that small investments have a lower internal rate of return than larger investments --- that is, they are more efficient uses of capital.

I think neither of these arguments is correct.

As Coase pointed out decades ago, there isn't a large distinction between a service supplier and an employee; in either case, you pay money and receive services. If there are two people engaged in making nails and five more engaged in carpentry using those nails, then if five more people take up the trade of carpentry, there is work available for two more nail-makers. It doesn't matter whether these seven people work for the same firm, or whether they buy the nails on the open market; the number of jobs created is just the same. Similarly, economies of scale do not necessarily depend on all the workers working for the same firm.

To put it another way, traditional companies create jobs by hiring people; nanocorps create jobs by buying goods and services.

Coase also pointed out that there is some distinction; one arrangement or the other might be more efficient, depending on the costs of transacting in the market and the inefficiencies of large firms.

Perhaps the Peruvians are entrepreneurial because they have particularly harsh pressures against large firms, for example due to government corruption or organized crime.

On to the issue of "big investments".

The number of jobs created is largely a function of the amount of value produced by the capital investments in question, and so the number of jobs created per dollar of credit is largely a function of the IRR of those investments, modulo the effects of inequality in distribution. If US$1000 of investment increases workers' productivity by US$1000 per year, it creates some number of jobs; but that number is likely to be more or less the same whether it is increasing 20 entrepreneurs' income by US$50 per year each (some of which they can spend on inputs such as cellphone minutes, electricity, wool, or wood, creating jobs in those industries) or whether it is increasing a factory's income by US$1000 per year (in which case it might do the same, or it might hire another worker or three).

The high (not to say usurious) interest rates of the microcredit loans in question, coupled with their low default rates, are strong evidence that the internal rate of return on these small investments is fairly high. Most of those loans are probably good choices for the people who receive them, I believe, or microcredit would have a serious image problem. (Journalists love nothing more than unmasking hypocrisy.)

There is no principled reason to think that in general, large capital investments will have high IRRs that are systematically unavailable to small capital investments. This kind of thing depends greatly on the technology available.

There are probably some big investments that are worthwhile.

Topics

Hybrid RAM

Kragen Javier Sitaker, 2016-09-24 (5 minutes)

There were a couple of papers published in the 2000s about "hybrid memory" systems with SRAM and phase-change memory (PRAM or PCM-RAM).

Here's the deal. The standard memory hierarchy for the last 35 years has been SRAM → DRAM → spinning rust, and now that's diversifying a bit, with SRAM → DRAM → NAND Flash → spinning rust, sometimes missing the spinning-rust part. Each item in this hierarchy is much more costly per bit than the next one, but also much faster. On each of these, the time to write one byte at a random location, the retail cost of that location, and the amount of such storage on a typical small netbook or cellphone, might be, very approximately:

- SRAM: 0.2 ns, 20 000 nanodollars, 1 MiB
- DRAM: 100 ns, 3.5 nanodollars, 4 GiB
- NAND Flash: 10 µs (10 000 ns), 0.26 nanodollars, 64 GiB
- spinning rust: 8 ms (8 000 000 ns), 0.03 nanodollars, 1000 GB (or 0!)

You'll note that the cost of each of these levels of the memory hierarchy is on the order of US$20.

From this you might think it was some kind of a natural law that faster memory costs more to make, but the truth is that there's plenty of memory that's both more costly to make and slower than something in that list. It's just that people stopped making it. Bubble memory, core memory, TTL SRAM, acoustic delay lines, punched paper tape, Williams tubes, magneto-optical disks, even magnetic tape and most photo and movie film have fallen to this economic logic.

I think this kind of thing is called the "efficient frontier" or "Pareto frontier", and it's a pretty general economic phenomenon: possible technologies are scattered around some cost/benefit plot space at random, but the ones that are economically viable are the ones that have more benefit than everything that's less costly, dramatically reducing the diversity of technologies. When there are more different benefits to trade off among, more diversity can survive.

A weird thing about this is that the time to read a byte is the same in all cases except for NAND Flash, because the process of erasing a block of NAND is slow, but reading it is potentially quite fast. In fact, reading it can be as fast as reading SRAM or DRAM, depending on how the Flash is designed. (A friend of mine went off to found a CDN startup based on this observation a couple of years ago; it's called Fastly and now powers a substantial fraction of the internet, disrupting the business of some internet giants.)

This is not as visible as it could be, because Flash has been slotted into the existing computing ecosystem as a disk replacement, so to some extent it's been limited by the I/O hardware and software built to support spinning-rust disks. Also, I think NAND Flash typically doesn't support byte reads, just reads of 256-byte blocks. I have to

investigate further.

This suggests that below a certain ratio of writes to reads, Flash (and similar technologies like PRAM, FeRAM, CBRAM, and MRAM) can displace not only some spinning rust but also many applications of DRAM and SRAM. This is especially interesting to me because of a different benefit of nonvolatile technologies like Flash: they use much, much less power than DRAM.

This is especially important because refreshing DRAM is actually a major user of computer energy nowadays, especially in mobile devices where power is so crucial. If it's possible to replace a large amount of DRAM with a larger amount of nonvolatile RAM and a much smaller amount of SRAM, it would be a huge win. This is what inspired all that "Hybrid RAM" research.

Of these alternatives to Flash, PRAM was the first to hit the market, and so it's the one that inspired the papers on hybrid RAM; it was only made by Micron, but only from 2012 to 2014. Intel is trying to commercialize crosspoint PRAM it developed with Micron under the name "3D XPoint". CBRAM, FeRAM, and MRAM are still available and mainstream.

A typical PRAM part might have been the Micron NP8P128A13TSM60E, 128 megabits (16 megabytes) of PRAM in a 56-pin package with 115-ns random reads (using a 25 MHz clock) and 50-ns random writes (using a 50 MHz clock), which is read performance slightly better than DRAM. (This was through an SPI interface.) It was rated for a million write cycles, and although its write speed was not as fast as DRAM, it didn't require a separate slow erase step. It supported writing or "programming" (ANDing) at 64-bit granularity.

Topics

- Electronics (p. 3430) (138 notes)
- History (p. 3500) (71 notes)
- Economics (p. 3424) (33 notes)
- The future (p. 3746) (20 notes)
- Nonvolatile ram

US$10M for a new, much better McMurdo Base, or less

Kragen Javier Sitaker, 2016-05-18 (updated 2016-05-19) (7 minutes) (on http://idlewords.com/2016/05/shuffleboard_at_mcmurdo.htm)

Reading this article (highly entertaining, as always, Maciej; thanks for brightening my day with your cynicism) led me to reflect on architecture. It seems pointlessly wasteful to have built a bunch of Quonset huts and trailers that people have to walk between, and also to have built the buildings above ground in the first place. I suppose digging in permafrost is difficult, and bringing in machinery so that you can have economies of scale is difficult, even if we no longer have the problems of cold steel embrittlement and tin pest that bedeviled the old polar explorers.

A little quick calculation, since calculation is always what I end up doing when confronting stories of human folly and suffering.

If you need to house, say, 1024 people, with 128 m² of floor area for each one (home plus office plus bar, etc.), with a mean ceiling height of 4 m, that's 524 288 m³. If you want to enclose that volume inside a hemispherical dome, the radius (and thus the height) of the dome is about 64 m, or 16 floors. The skin of the dome — the part that insulates the people from the cold wind — is about 26000 square meters, 25 square meters per person. The cross-sectional area that the dim sun illuminates during the summer is about 6400 m², so if we assume a bit less than 1000 W/m², you receive about 6 megawatts of solar energy during the summer, and about 3 megawatts year-round (3 kW per person).

How much insulation do you need? If the inside-outside temperature difference is 40° C, and you need to maintain that on 1500 W per person (maybe your thermal solar collection is only 50% efficient and you don't have significant other sources of heat) then you need insulation with an average R-value of about 3.8. 1-inch polyisocyanurate foam panels have an R-value of about 6, and they only cost US$19 for a 4'×8' panel (US$6.40/m²; this is the Home Depot retail price and includes aluminum facers) which works out to about US$160 000 to cover the whole dome. Unfortunately they aren't transparent, so you can't get solar radiation through them; you kind of need some kind of non-imaging optics heliostat if you want to gather the solar heat to illuminate and keep warm with. As far as I know, these don't exist yet.

At this point, and certainly when McMurdo was built, it would make more sense to use heavier insulation, and do your climate control by dissipating energy that you generate by some other means, such as with the nuclear reactor or by burning fuel oil. If you're dissipating 500 watts per person, you need three times the R-value (11.4, a bit under two inches of foam insulation, or US$320 000 of insulation). You need to use countercurrent heat exchangers to keep the air from going stale.

16 stories is small enough that people can avoid using elevators most of the time, at least if the common areas they usually travel to are

intelligently located; 128 meters diameter is small enough that you can walk anywhere (in about two minutes), but large enough that bicycles or skateboards would occasionally be convenient.

If each floor is concrete 250mm thick (suitable for essentially any purpose that doesn't involve armored vehicles), we need 32768m³ of concrete, or about 65536 tonnes, if we use somewhat lightweight concrete. Concrete typically costs about US$120/m³, so that's about US$4 million of concrete, plus probably a similar cost in rebar. This cost isn't particularly sensitive to whether you build a bunch of separated buildings or a single giant arcology like I'm suggesting above, as long as the buildings are more than two or three stories tall. It is sensitive to things like whether each person gets 128m² or 32m² and to the flooring material.

If you could somehow get by with more inexpensive floor materials like expanded steel sheet with drywall under it, you could reduce the cost dramatically — the 250mm reinforced concrete I suggested above costs about US$60/m², while 9-gauge expanded steel sheet might cost US$24/m² (according to MetalsDepot.com). I feel like that kind of thing might be acceptable for a lot of floors that don't separate unrelated strangers.

This is important not so much for the cost of the materials (although that is kind of important) but more because you don't have a cement plant or even a quarry onsite there in McMurdo; all your manufactured materials have to be shipped in, as if you were in Alaska or something. A heavy-tested TEU only holds 28 tonnes; the amount of concrete suggested above is 2340 TEUs' worth of mostly sand and rocks. That's because the concrete weighs a ton per square meter, while the expanded sheet metal weighs 8.8 kg per square meter. That way, you might only need 25 or 30 TEUs instead of 2300 of them.

(The above conveniently omits the sheetrock...)

What about trash? Is it really necessary to haul it away from Antarctica? Let's make some pessimistic assumptions: suppose we need to plan to store 64 years' worth of garbage — ideally, frozen — and that the McMurdo Base residents and visitors produce the same amount of garbage per capita as New Yorkers, who are twice as productive of waste as any other major metropolis, at 7.8 million tons per year (220 kg/s) out of 8.6 million people (26 mg/s per person). Over 64 years and 1024 people, this is 54 000 tonnes of garbage, or about 54 000 m³.

This is about an order of magnitude smaller than the size of the people dome. If we just build a garbage dome near the people dome and put garbage in it every day and let it freeze, the garbage dome will only be five stories tall if it's built for 64 years' worth of garbage.

Presumably it will take less than 64 years for it to become economic to mine the rich deposits of refined mineral resources (indium, gallium, gold, copper, maybe even aluminum if energy prices don't fall dramatically) in the garbage pile.

Hopefully, the McMurdo residents will demolish less buildings, buy less new clothes, junk less taxis, and compost more of their food than New Yorkers do, so hopefully their garbage volume will be even smaller.

So you could probably build a new, much better McMurdo Base for under ten million dollars.

Topics

Alphanumerenglish

Kragen Javier Sitaker, 2015-04-06 (updated 2016-07-27) (6 minutes)

Arabic has some phonemes that don't have a reasonable corresponding phoneme in Latin languages. A common approach when writing Arabic with Latin letters is to use European digits for the missing phonemes, as in "3arabawy". This has the great advantage over X-SAMPA that it doesn't deprive you of the uppercase/lowercase distinction entirely — although the digits themselves, in ASCII, don't show the distinction visibly, you can still distinguish between "3arabawy" and "3ARABAWY", or between "a7a", "A7a", and "A7A". The European digits are assigned to sounds represented by Arabic letters that physically resemble them.

This seems like a great idea for English, too! English has a few more phonemes than the Latin alphabet, and consequently has to force some letters to represent multiple different phonemes, but its traditional orthography worsens the problem by wasting some of the Latin letters (c, q, and x) on redundant assignments. A rough phonological inventory of English follows, with a corresponding set of alphanumeric codes.

Latin	IPA	Alphanumerenglish
a	æ a ɑ eɪ ə	4 a 2 8 e
b	b	b
c	k ʃ tʃ	k c tc
d	d ɾ	d d
e	ɛ i	3 i
f	f	f
g	g dʒ ʒ	g dj j
h	h	h
i	aɪ ɪ i	ai 1 i
j	dʒ ʒ	dj j
k	k	k
l	l	l
m	m	m
n	n ŋ	n 6
o	u ʊ ɑ ɔ oʊ æʊ	u 5 2 0 o au
p	p f	p f
q	k	k
r	ɹ ɚ	r er
s	s ʃ z	s c z
t	t θ ð d ɾ	t x q d d
u	u ʌ ju	u 7 yu
v	v	v
w	w hw	w hw
x	ks	ks
y	j i ɪ aɪ	y i 1 ai
z	z ʒ	z j

In a few cases, I've taken the liberty of giving a diphthong a spelling that isn't quite what you'd deduce logically, in the interest of making it easier to write: /oʊ/ is "o" rather than "o5" or "ou", because I think /o/ doesn't occur in outside of that diphthong in

English, and likewise /eɪ/ is assigned a single digit "8"; /æʊ/ is "au" rather than "4u" or "45", because although [au] is not the central instance of the /æʊ/ class, it's certainly a valid pronunciation of it; and /aɪ/ is "ai" rather than "a1" for the same reason.

The mnemonics for the digits and other nonstandard pronunciations are as follows:

0	ɔ	looks like an 0
1	ɪ	looks like an I
2	ɑ	no clue, sorry
3	ɛ	looks like an ɛ, backwards
4	æ	looks like an A
5	ʊ	no clue, sorry
6	ŋ	looks like the G in NG
7	ʌ	looks like ʌ tilted a bit
8	eɪ	8 is pronounced /eɪt/ in English
c	tʃ	"c" is pronounced like standard "ch"
e	ə	looks like ə; also, the most common vowel
q	ð	no clue, and this one is super weird, sorry
x	θ	no fucking clue, sorry

In the interest of ease of typing and reading, I've tried to assign the less common sounds to digits, and especially digits that don't look like anything. You might be able to get some benefit from swizzling around the assignment of 2 and 5, and maybe x and q too.

You might also get a readability improvement out of reassigning /ə/ to "a" rather than "e", which would leave "e" free for /ɛ/, but would require reassigning /a/ to something else, such as "9" or "3". /a/ alone is relatively rare in English, but "ai" and "au" are relatively common.

I've written a short Python hack to use eSpeak to convert traditional English orthography to Alphanumerenglish, not perfectly but with a tolerably low error rate. It turns out Alphanumerenglish is about 10% shorter.

Q7s wi ken rait i6gl1c perfektli fen3t1keli 1n 4lfenum3r1k w1q4ut nidi6 qe c1ft ki or qe los ev k8s d1sti6kcenz, werd sp8si6, or p76kcu8cen. 1t's almost ridebel w164ut sp3cel tr8ni6, despait qe n3ses3ri k2mpremaizez 1n qi esainment ev gr4fimz te fonimz.

'Tw7z br1l1g, end qe slaiqi tovz
D1d gair end g1mbel 1n qe w8b.
Al m1mzi wer qe borogovz,
End qe mom r4xs autgr8b.
  Biw3r qe dj4berwok, mai s7n!
Qe djoz qet bait, qe kloz qet kotc!
Biw3r qe dj7bdj7b berd, end c7n
Qe frumies b4ndersn4tc!
  Hi t5k h1z vorpel sord 1n h4nd.
Lo6 taim qe m86ksom fo hi sot.
6en r3sted hi bai qe t7mt7m tri
End st5d e hwail 1n xot.
  End 4z 1n 7f1c xot hi st5d
Qe dj4berwok, w1x aiz ev fl8m
K8m hw1fli6 xru qe teldji w5d
End berbeld 4z 1t k8m!
  W7n, tu! W7n, tu! End xru end xru
Qe vorpel bl8d w3nt sn1ker-sn4k.
Hi l3ft 1t d3d, end w1x 1ts h3d
Hi w3nt gel7mfi6 b4k.
  "End h4st qau slein qe dj4berwok?
K7m tu mai armz, mai bim1c boi!

O fr4bdjes dei! Kelu, kel8!”
Hi tcordeld 1n h1z djoi.
  ’Tw7z br1l1g, end qe slaiqi tovz
D1d gair end g1mbel 1n qe w8b.
Al m1mzi wer qe borogovz,
End qe mom r4xs autgr8b.

Kemp3r q1s te qi X-SAMPA verjen:

twVz brIlIg, @nd D@ slAIDI toUvz
dId gAIr @nd gImb@l In q@ weIb.
al mImzi w@` D@ bOrOgovz,
@nd D@ moUm r4Ts aUtgreIb.

Te mai ai, et list, qi eks-s4mpe verjen 1z box lo6ger end l3s ridebel ez normel i6gl1c, ez w3l ez luzi6 qe degriz ev fridem ev k4p1telez8cen end sem p7nkcu8cen tu. Ev kors, X-SAMPA also h4z qi eb1ledi te r3prez3nt l86guedjez 7qer qen i6gl1c, almost ez izeli, hw1tc 4lfenum3ri6gl1c s4kr1faisez.

1ded bi jusfel te rait e w3b skr1pt te kenvert st4nderd i6gl1c te 4lfenum3ri6gl1c, m8bi jusi6 `espeak -qx`.

Topics

Dercuano plotting

Kragen Javier Sitaker, 2019-09-03 (updated 2019-09-05) (34 minutes)

Related to Dercuano rendering (p. 2300), Dercuano formula display (p. 495), Dercuano calculation (p. 3135), and Dercuano drawings (p. 64), but not the same — I want some kind of equation-and-data-plotting thing in Dercuano, with some kind of Jupyter-like rapid feedback. I think I can make it simpler and less fiddly than Numpy or Pandas, using the ideas in APL with typed indices (p. 3264), A principled rethinking of array languages like APL (p. 1995), Relational modeling and APL (p. 1217), and First impressions on using the µMath+ calculator program for Android (p. 195), plus some I got from Darius Bacon, and get some formula display out of it into the bargain.

As discussed in those notes, prerendering images to PNG or JPEG files like the humans normally do is not really an option for Dercuano because of its 5MB total download size budget.

A concrete example

In Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) I wrote 4800 words about an algorithm; the resulting HTML compresses to 11kB. I tried it in an IPython notebook which contains 511kB of text, mostly compressed images, so I'm not including it as part of Dercuano (that, and because it's in a file format that browsers don't recognize). The actual Python code in the notebook is 2.3kB and compresses to 0.7kB. With reasonable JS signal processing and plotting libraries, this implementation could be part of the text, also costing only about 0.7kB, the plots would be viewable alongside the text, and they could be an "explorable explanation" in the sense that you could interactively vary the parameters and observe how this affects the plots.

Rapid feedback HCI

In 2015 I wrote an RPN editor with numbers and 1-D arrays that generates simple plots and formula renderings as you calculate, a followon to an older JS calculator I wrote in 2005; in 2016 I did a similar hack where instead of calculating on *arrays* the elements you calculate on are *functions*, starting from the identity function $f(x) = x$ and constant functions $f_k(x) = k$, then combining them pointwise. These are all fairly keyboard-driven (Interactive calculator (p. 2771) explores how to do a multitouch UI) and prototype-quality.

One of the interesting things about the 2015 RPN editor above is that it uses the URL #fragment identifier to store the entire application state, much like erlehmann's `glitch:` URLs for bytebeat, so that you can bookmark the calculation state or pass it to someone else in a link. In some sense, it's an interactive *viewer and editor* for a calculation *text*, with some linguistic representation — in this case, RPN, since nothing more complex is needed.

The 2015 RPN editor also allows you to highlight subexpressions (with the ←→ keys) to see their values, and to structure-edit it (with ^←/^→), although that is confusing.

Another thing about all three of these prototypes is that you don't have to request for a result to be plotted — as soon as it exists, it gets plotted. But they need more flexibility in *how* to plot things. (The 2005 one gives you the option of resizing a plot with the mouse, while the others don't even do that much.) Every value has an infinity of possible visible presentations; peremptorily displaying two of them is not enough.

These all feel much more immediate than the experience with IPython/Jupyter, where you are constantly faced with the alternative between *using* a value you have calculated:

```
t = dt * arange(20e-3 / dt)
```

and *seeing* it:

```
dt * arange(20e-3 / dt)
```

and plotting a function so you can see *both its domain and range* and have it *labeled* and have *more than one plot* requires bending over backwards:

```
subplot(211)
plot(t, VR, label='$V_R$')
plot(t, VL, label='$V_L$')
plot(t, VC, label='$V_C$')
legend()
subplot(212)
plot(t, I, label='$I$')
legend();
```

Consider, instead, being able to say:

```
(VR over VL over VC) atop I
```

or the equivalent with keystroke or touch commands? I mean `VR` isn't dependent *just* on `t` — in this notebook it also depends on `C`, `L`, `R`, and `dt` — but `t` is the axis I've been thinking of it as varying with here, while I've been treating those other variables as constants. So is it too much to ask that my calculating and plotting system would be able to infer that, at least unless I override it? Especially when I'm plotting `VL` *on the same axis where I already plotted `VR` against `t`*? Sheesh!

Another thing is that, if you're evaluating a function of more than one variable at many points so you can plot it, Numpy (like APL, Octave, and R) can't keep straight which variations belong to the X-axis and which belong to the Y-axis. It chokes on this:

```
R = array([1000, 2200, 4700, 10e3, 22e3, 47e3])
C = array([100e-9, 220e-9, 470e-9])
matshow(R * C)
```

It complains, "ValueError: operands could not be broadcast together with shapes (6,) (3,)", which is to say that it was trying to multiply corresponding elements of `R` and `C` to get time constants. If we want the two to vary independently, `R.T * C` doesn't work as you might expect, but we can say

```
matshow(multiply.outer(R, C))
```

or

```
matshow(R.reshape((6, 1)) * C)
```

But then the next time you do a calculation involving both `R` and `C`, you have to tell Numpy *again* that you want them to vary independently. And this is what A principled rethinking of array languages like APL (p. 1995) is about. (Also, don't forget `colorbar()`, which is not the same as `legend()`.) This is actually the same problem as getting the X-axis labels right by default: for Numpy, `R` is just a vector of six numbers, just as earlier `VR` was an array of 100'000 numbers. It doesn't have any idea why there are six.

The only software I've seen that does get this right is μMath+; see First impressions on using the μMath+ calculator program for Android (p. 195) for details.

Of course, I keep using Jupyter, despite the above, and even though I can't incorporate the plots into Dercuano. That's because in my 2015 prototype calculator I haven't even implemented typing in negative numbers or decimals, much less multidimensional arrays, the Fast Fourier Transform, or singular value decomposition; moreover, I can probably expect a 10× slowdown just from switching the inner loops of these numerical algorithms to JS from Fortran in LAPACK.

Integration with my current workflow

This AJAXy thing I described above has a difficulty: I'm mostly writing Dercuano in Emacs, not in some kind of browser-based IDE. I could reasonably pop out of writing to the browser to do some graphing (I could even add a keybinding in Emacs), but ultimately whatever I put together in the browser needs to be something I can paste into a text editor, and ideally something that will diff reasonably well.

Probably the best I can hope for there is to pop open a textarea that says something like

```
<script>
calc(`jasiodj jiaji aoj ioaj iojgosjo
jaiogjaoj
aijgwj jaiojgioawj owj oiajio jaweoj jaiojo
jaiogjwojao ioj ioaj oij ioawj oaj aj iawjiawejisjga0 auj
ajigwaj jawjiawjipwuj0aweuj890ejgp aji
ajiajijwijapgjawpj`)
</script>
```

where the text inside the `` encodes the calculations and plotting options. Then I can copy and paste this into the text editor, hopefully remembering to delete the previous version. A hassle, but manageable. (Maybe a keybinding can find the surrounding `<script>` tag, paste in whatever is on the clipboard, and if it looks like a new `<script>` tag, delete the old one.)

The `` syntax is new in recent versions of JS.

Even the hairiest plots I've been doing so far should be encodable in a kilobyte or two of text, and maybe different plots in the same

document could talk to each other.

Why I don't want to try an embedded DSL in JS

Numpy is an embedded DSL in Python, rather than a separate language implemented in Python, much less a funky keystroke-driven RPN UI. So why not do the same thing in JS?

First, JS doesn't support operator overloading, which is a bigger deal for readability than it sounds like. (a + b + c + d)/e.sum() becomes a.plus(b.plus(c.plus(d))).divide(e.sum()). It's already hard enough to tell if the computation you specified was really the computation you wanted; this makes it much harder still.

Second, I prefer the RPN UI because it's a lot more fluid than typing in strings of Python or JS. See the above complaints about Numpy for some of the reasons.

Third, I want to be able to define functions in a somewhat abstract, static way so that replotting them over different regions is a reasonable thing to do. I even want to be able to do this for Runge–Kutta integration and things like that, although I don't know how successful I'll be. Embedding your DSL means that the host-language facilities are always ready-to-hand, but they would frustrate this ambition.

Rendering improvement

As explained in Antialiased line drawing (p. 1803), we could go a long way to improve the readability of graphs by using LCD subpixel antialiasing and a bit of signal-processing theory, instead of drawing PostScript-style convolutions of the graph line with a one-dimensional boxcar kernel at right angles to it.

Ideally, the line plotted on a plot is infinitely thin, a line-shaped Dirac delta, but rendering it that way requires not only infinite resolution but also infinite dynamic range. (The infinite dynamic range is particularly a problem for dark lines, since it would require either emitting negative amounts of light (at infinite concentration) from the line itself, or drawing on an infinitely bright background.) Bandlimiting the Dirac delta to a sinc that won't alias at the screen resolution and maybe inverse-filtering a bit to compensate for the rectangularity of LCD pixels (which amounts to a low-pass filter through convolution with a rectangle) should give a high-quality rendering; windowing the sinc should make it more computationally tractable, but of course requires a little more frequency headroom. Reducing the height of the peak in the center of the filter kernel should help at reducing the demands on the dynamic range of screen pixels, but maintaining the sharpness of the peak there should help with visibility. High-pass filtering the filter kernel a bit, maybe without terribly strong stopband attenuation, should also improve the precision/visibility/dynamic-range tradeoff.

Attenuating the lowest- and highest-frequency components this way has the effect of spreading the line's brightness over more pixels, which means that it can vary more within the same dynamic range; this is important when lines cross or pass very nearby. However, I don't know whether the dynamic range increases proportional to the number of pixels or to their square root.

Any opacity, even if it merely results from saturation, is nonlinear and tends to generate alias frequencies. (It might be possible to avoid the generation of alias frequencies through some kind of very careful balancing, but if you don't manage to do that, they will be present, and for the right pattern of lines they will be overwhelmingly strong.)

Windytan's oscilloscope-emulation algorithms demonstrate what can be achieved with closer-to-ideal plot rendering — aside from the issues of correct interpolation close to the Nyquist frequency, there's lots of detail that is lost to the nonlinearities of the standard approach to waveform plotting but visible on an analog oscilloscope.

It might be possible to get such effects purely in SVG — SVG 1.1 in 2003 already defined the `filter` element and the `filter` property, which supports an `feConvolveMatrix` filtering primitive that I think could in theory handle this. I've rarely or never seen this element in the wild, making me think its implementation is probably not well tested, and so might have performance or even correctness issues. The spec page is well worth reading as an overview of what 2-D graphical primitives the experts at Adobe thought were important in 2003; they go well beyond what PostScript can do.

To get deep-subpixel line positioning, a brute-force approach is to render with minimal antialiasing at a much higher resolution, then convolve with an antialiasing filter kernel at the high resolution before decimating to screen resolution. This is probably not very computationally efficient. More efficient approaches might include precomputed fractional-delay filters to shift patterns by fractions of a pixel and texture-mapping with a 1-D texture representing the pattern produced by the integrated filter kernel along a line perpendicular to the line being drawn, plus some kind of linear or quadratic adjustment to account for sharp angles or sudden ends.

It's often observed that bright lines on a dark screen background are more visible to the humans than dark lines on a bright screen background; this is particularly a problem for things like visualizing two-dimensional scalar fields such as the signed response of a filter kernel. I don't have a good understanding of why this is; I wonder if it has something to do with the humans' logarithmic brightness perception, where a bit of blurriness diminishes the white around a black line by an imperceptibly tiny amount, while the same blurriness will convert the black around a white line into a slightly dimmer white.

If this is the reason, it means there's an unavoidable compromise between correct in-focus appearance (where the logarithmic perception law means we should do our convolutions in logarithmic color space) and correct out-of-focus appearance (where the defocus inside the human's eye mixes the light linearly, so we should do our convolutions in linear color space). Using strong contrast sparingly should reduce the costs of this compromise.

With these tricks, it should be feasible to get lines that are an order of magnitude more visible than the traditional 250-micron-wide 125-micron-quantization-noise Bresenham lines that Gnuplot will give you by default, while at the same time being more than an order of magnitude more precisely positioned in the X dimension (say, 10 μm), on a traditional 100-dpi, 250-micron-resolution LCD screen with vertical RGB subpixels, and nearly a factor of magnitude more

precisely positioned in the Y dimension (say, 30 μm).

On the high-dpi screens now common on hand computers — 200 dpi, or 127-micron pitch with 42-micron pitch if it has RGB subpixels, is a typical resolution nowadays — it should be possible to get positioning errors on the order of 5 μm in X and 15 μm in Y.

Still, none of this is needed for an "MVP", which can be done straightforwardly with <canvas> or SVG (possibly using d3).

What I use most in Numpy, SciPy, and matplotlib

Maybe if my calculating/plotting thing can do most of the things I can do in IPython/Jupyter, it'll be comfortable to use for a variety of things.

I looked through 16 of my recent IPython notebooks and came up with this top-64 list by frequency of use (in source code, not execution):

```
105 plot
 68 subplot
 55 *
 41 **
 40 len
 40 []
 39 [:]
 35 -
 27 xlim
 27 abs
 26 copy
 19 @
 18 matshow
 17 contour
 17 [:,:]
 16 sum
 16 '.'
 15 set_*scale('log')
 15 resize
 15 print
 15 arange
 15 /
 14 linspace
 14 legend
 14 -=
 13 max
 13 fft.fft
 13 +=
 13 [:,]
 12 stem
 12 colorbar
 12 array([])
 11 zeros
 11 ylim
 10 pi
 10 [,:]=
  9 exp
```

```
9 +
9 [,:]
9 >
8 inv
8 concatenate
8 [:]=
7 []=
7 [:,:]=
6 .T
6 cumsum(axis=)
6 '.-'
5 sin
5 shape
5 reshape
5 min
5 max(axis=)
5 gca().set_aspect('equal')
5 cumsum
5 cond
5 [:,]=
4 xticks
4 where
4 svd
4 plot(linewidth=)
4 [,]
3 sum(axis=)
3 round
```

This is from a bit over 1000 invocations of Numpy array operations and matplotlib operations. `plot` is super popular, and so is damned `subplot`, but `stem`, `matshow`, and `contour` also appear a lot. Arithmetic `*`, `**`, `-`, `@` (matrix multiply), `/`, `-=`, and `+=` are very popular; `+` is less so. Popular aggregate operations are `len`, `sum`, `max`, and to a lesser extent `min`. And `abs`, `exp`, and `sin` are surprisingly popular.

Then there are indexing and slicing operations. A *lot* of indexing and slicing operations. Like, just scalar index reads are #6, more popular than *subtraction*. It might have been worthwhile to break down the kinds of slicing a bit more: sometimes it's between two constant indices like `x[200:400]`, sometimes it's dropping some elements from the beginning `x[3:]` or the end `x[:-3]`, and sometimes it's some other calculated index like `x[pos:pos+size]`. Sometimes it's a coordinate shift, sometimes I intended to select a subset (often for plotting), etc.

Popular plotting options include `xlim`, `'.'`, `'.-'`, `yscale('log')` (and occasionally `xscale` too), `legend`, `colorbar`, `ylim`, `gca().set_aspect('equal')` (which doesn't have a convenient function in pyplot the way `set_yscale('log')` does), and `xticks`.

Popular heavy-duty algorithms are `fft.fft`, `inv`, `cond`, and `svd`. Maybe matrix multiply `@`/`dot` should be included there too.

Popular ways of generating arrays, other than arithmetic, include `copy`, `resize` (which in Numpy repeats an array, like `tile`), `arange`, `linspace`, `array([])` (converting a literal list to an array), `zeros` (typically followed by assignments), and `concatenate`, which puts the elements of one after the elements of the other.

Other miscellaneous facilities I apparently use a lot include `pi`, `cumsum`

, `.T`, `reshape` (a generalization of `.T`), and `where` (conditional: `where(a, b, c)` is `b` where `a` is true, `c` where `a` is false).

Not all of these operations would map over to other environments in exactly the same way. In particular, a lot of the plotting options are maybe things to set with the mouse.

Attaching aesthetics to data

The Grammar of Graphics refers to the visual appearances we attach to data to make it visible as "aesthetics" — as in:

> Aesthetic attribute functions are used in two ways. Most commonly, we specify a variable or blend of variables that constitutes a dimension, such as *size*(**population**) or *color*(**trial1+trial2**). Or we may assign a constant, such as *size*(3) or *color*("red").

They specifically disclaim "the derivative modern meanings [of "aesthetics"] of beauty, taste, and artistic criteria".

In GG, as in most graphics systems, data do not have aesthetics. Instead, aesthetics have data. This is also how matplotlib, d3, and Gnuplot do things. The data are floating around in vectors or whatever, and at some point they collide with a plotting command or a plot-update command, and at that point they get used, perhaps ephemerally, to generate a graphic; but subsequently they lose their connection to the graphic.

I think this is probably not the best approach for an interactive calculator with instant feedback. Instead, aesthetics and indeed a whole presentation should be attached to the data, so that the data can always be plotted in a sensible way at any point in the calculation. (Bret Victor has demonstrated some visualizations of Dan Amelang's Nile which probably inspired this thought.)

I don't know how exactly this should work. Probably if you plot two different voltages in different colors or different linewidths, they should retain those aesthetics whether you're plotting them against time or against their common current — but what if you are plotting them against each other, with one on X and the other on Y? What if the current has its own color? What color should the sum of the voltages be, or the square of one of them? I probably need to try stuff to see what feels least frustrating.

For short discrete signals, `stem` is probably the correct presentation under most circumstances, and plenty of operations on discrete signals are closed; so probably if you add two stem-displayed signals, or multiply one by a constant, you should get another one. But `stem` becomes unwieldy for sufficiently many samples. Do I need conditional formatting?

(One potential benefit of the more symbolic way I'm thinking about doing things is that discrete and continuous signals are not the same.)

Square aspect ratios — a common tweak — are nearly always appropriate when the axes are in the same dimension. But tagging every variable with units of measurement might be unwieldy. (On the other hand, it might help to associate some aesthetics with units of measurement rather than values. And `units.dat`, now `definitions.units`, gzips to 78 kilobytes.)

The implicit, conditional associations in A principled rethinking of array languages like APL (p. 1995) should help somewhat with the problem of associating varying quantities with an aesthetic — it should be just as easy to set the voltage's linewidth to be the current as to set

it to 3. (You might need some kind of scale mapping from amperes to pixels, though.)

A possible alternative is, as in the 2016 prototype, to do computation by *changing* a *variable* — for example, adding a constant to it, or multiplying it by a time-lagged version of itself — and update a pre-existing display accordingly.

Another approach, explored in Relational modeling and APL (p. 1217), is for these quantities to exist as named attributes of a model, which then has one or more visual presentations. A cylinder, for example, has a volume, a cross-sectional area, a lateral area, a total surface area, a radius, a diameter, and a length. But I'm not sure how this would work with plotting a series of different cylinder volumes against some independent variable.

Abstract model/language semantics

There are two pieces here: one is the semantic model of the *plotting*, and the other is the semantic model of the *calculation*.

I anticipate that the model of the calculation is going to be a longish document, so I'm preemptively splitting it out into A formal language for defining implicitly parameterized functions (p. 144).

Performance

JS is not going to be as fast as Fortran, as evidenced by things like PDF.js, modern JS interpreters can be coaxed to be fast enough to do some substantial computation.

Firefox takes about 3.6 seconds to run this JS on my laptop:

```
function tri(n) {
  let t = 0
  for (let i = 0; i < n; i++) t += i;
  return t;
}

tri(1000000000)
```

It also gets the wrong answer, because of 64-bit floating-point roundoff error, but that's not the point. The point is that it was able to chew through 280 million loop iterations per second. Given a 32-millisecond budget to render a graphic, it can do 9 million simple arithmetic operations like the above.

I tried it in C:

```
#include <stdio.h>
#include <stdlib.h>

long long tri(long long n)
{
  long long t = 0;
  for (long long i = 0; i < n; i++) t += i;
  return t;
}

int main(int argc, char **argv)
{
```

```
  printf("%lld\n", tri(strtoll(argv[1], 0, 10)));
  return 0;
}
```

Without optimization, it was the same speed as Firefox; with optimization, I had to make the number a command-line parameter to keep GCC from evaluating the loop at compile time, and it takes 900 ms, four times as fast. (Also, it gets the right answer, unlike JS.)

So the cost of JS for this simple integer numerical code is about a factor of 4. So JS on my laptop or my phone is faster than C on my netbook. And my rule of thumb is that code in Numpy takes 5× longer to run than reasonably written C, so JS might actually be faster than Numpy. We just need to compile the dataflow graph into nested loops in JS before evaluating it in order to get that delicious JITty goodness!

I tried to test array indexing speed but all I found out was that integer division is super slow and now I need to redo everything below

But JS array indexing is bounds-checked, so it might be a lot slower than C. So I wrote these quick functions in Firefox's inspector console to see:

```
function leap(a, n) { let m = a.length, j = 0; for (let i = 0; i < n; i++) { a[j]
 += i; j = (a[j] + j) % m; } }

function time(t) { let a = new Date(); let b = t(); let c = new Date(); return [c
-a, b]; }
function repeat(x, n) { return new Array(n).fill(x); }
```

Thus `time(() => leap(repeat(0, 8), 5000000))` gives 1.49–1.51 seconds (in another run, mentioned below, after a reboot, 1.23–1.28 seconds instead); at 10 million it gives 3.09–3.11 seconds. *Enlarging* the array to 8192 *speeds up* both of these, 5 million to 1.09–1.12 seconds; enlarging it further to 65536 speeds up 5 million to 9.91–1.02 seconds; at 16777216, 2.3–2.6 seconds; at 1048576, 1.46–1.52 seconds; at 2097152, 1.32–1.37 seconds; at 33554432, 3.9–4.6 seconds. Moreover, at 33554432, doubling the loop count to 10 million only extends the time to 5.5–6.9 seconds. It doesn't start to get linear again until 20 million, at 8.5–9.6 seconds.

(I did three trials of each one to get some idea of the variability, but probably the JIT is too unpredictable for just three trials to be decent.)

I don't know what to make of this precisely, but it seems like for small arrays, it can do 3 to 5 million of those Array inner loops per second, which is enormously less than the 280 million it was getting for just adding the loop counter, and then starts to get slower presumably due to cache effects for indexing arrays over 32 mebi-items.

To see if the optimizer is replacing the % with a &, I tried reducing the array size to 33554431, which didn't make any difference. This suggests that maybe I should try explicitly using & to see if the 97% of the work this program is doing has a lot of division in it.

At 10 seconds, Firefox shows its warning that "a web page is slowing down your computer", offering the option to kill the computation; this is a thing to beware of.

Typed arrays

To compare, I tried `time(() => leap(new Float32Array(8192), 5000000))` and got 0.964–0.969 seconds; Int32Array gave 0.924–0.935 seconds; Uint8Array gave 0.50–0.53 seconds; Uint16Array gave 0.74–0.76 seconds; Uint32Array gave 1.170–1.172 seconds; and Float64Array gave 1.21–1.23 seconds. Both of these last two are slower than just using `Array`. This is well below the size where cache effects came into play, and the `leap()` loop is specifically designed to not be vectorizable, so I don't know why the smaller data types give a performance boost (up to 10 million iterations per second!).

To see if we get big caching effects, I tried `time(() => leap(new Float64Array(33554432), 20000000))` and got 8.35–8.41 seconds; with Float32Array I got 5.9–6.2 seconds; and, astonishingly, with Int16Array — which I expected to be *faster* — I got 14.6–15.3 seconds.

I don't think these results are predictable enough to draw very precise conclusions about whether, or even when, typed arrays help or hurt performance. They seem to help performance by a factor of 2 in some cases and hurt it by a factor of 1.5 in others. Maybe a better benchmark function would help.

Adding methods to arrays makes no difference

After the reboot mentioned below, I thought I would try `time(() => { let a = repeat(0, 8); a.method = function(x) { return "hi, " + x; }; leap(a, 5000000)})` to see if the extra method on the array frustrated Firefox's optimizer. (This is a thing I'd done in 81hacks and always wondered if it was the reason for what seemed to me to be relatively poor performance.) It didn't make any difference: it took 1.262–1.265 seconds, within the 1.23–1.28 range observed immediately previously.

However, it's certainly possible that the bottleneck in `leap()` isn't actually the array indexing! (It turns out to be true, so I need to redo this test.)

Memory use

I tried running `x = repeat(0, 1024*1024*64)` and had to reboot after Firefox allocated a few gigabytes of virtual memory.

After rebooting, it was hard to tell which of the many Firefox processes to watch in htop. `x = repeat(0, 16777216)` did not make it apparent. It turned out to be pid 4172, as revealed by `f = n => (n < 2 ? 1 : f(n-1) + f(n-2)); f(36)`, using 1839MB VSZ, 336MB RSS. Rerunning the `repeat` boosted that to 2366MB VSZ, 858MB RSS, a difference of 527 and 522 megabytes respectively. That suggests that each array item is occupying a bit over 32 bytes, which is four times what I expected. `delete x` returned the process to 1976MB/465MB.

Presumably typed arrays should reduce this substantially, and indeed, after `x = new Uint8Array(16777216)`, we see 1972MB/479MB, a 14MB jump in RSS, close to the expected 16MB. `delete x` has no effect (481MB remained constant before and after) but `x = new Float64Array(16777216)` boosts memory use to 2257MB/606MB, 285MB and 127MB respectively; the latter is very close to the 128MB you'd expect at 64 bits (8 bytes) per array item.

So native JS arrays are four times more expensive on memory use

than you'd naively expect, while typed arrays have exactly the memory price they say on the tag, which can be more than an order of magnitude better. Given that there's no consistent runtime cost for using typed arrays, though also no consistent benefit, it is probably better to use typed arrays by default for numeric data arrays. (Presumably using typed arrays will make *other* code run faster by reducing the load on the garbage collector.)

How I found out the speed tests above were totally wrong

This took 1.0–1.4 seconds: `time(() => { const a = repeat(3, 16777216); let t = 0; for (let i = 0; i < a.length; i++) t += a[i]; return t })`. That's, like, 10 or 20 million array indexing operations per second. However, this still took 0.9 seconds: `time(() => { const a = repeat(3, 16777216); let t = 0; for (let i = 0; i < a.length; i++) t += i; return t })`. So it wasn't really the array indexing in the loop; it was the `repeat` function above. Changing to `time(() => { const a = new Array(16777216); let t = 0; for (let i = 0; i < a.length; i++) t += i; return t })` gives 61 milliseconds instead, 15 times faster — so it was the `.fill()` call.

It only takes 73–75 ms to run `time(() => { const a = new Float64Array(16777216); let t = 0; for (let i = 0; i < a.length; i++) t += a[i]; return t })`, which just totals up all the zeros. But I'm not sure how far I can trust Firefox's optimizer here.

This version takes wildly varying times from 168 ms to 639 ms: `time(() => { const a = new Float64Array(16777216); for (let i = 0; i < a.length; i++) a[i] = 3; let t = 0; for (let i = 0; i < a.length; i++) t += a[i]; return t })`. Using `Array` instead slows it to 0.9–1.5 seconds, which is probably slower than calling `.fill()` inside `repeat`.

So, I don't know. Loop analysis and bounds-checking hoisting is easier with loop counters, and maybe that's what accounts for the difference. Certainly my earlier typed-array tests weren't calling `.fill()`; they just relied on the implicit zero-filling provided by these constructors (even the float ones), which, as we see above, is much faster than what I was doing. So maybe it really was the inner-loop division.

Here's a division-free version specialized for power-of-two arrays:

```
function laap(a, n) { let m = a.length-1, j = 0; for (let i = 0; i < n; i++) { a[o
oj] += i; j = (a[j] + j) & m; } }
```

And, with that, `time(() => laap(repeat(0, 8), 5000000))` takes 50–80 ms. On my netbook, it takes 270–310 ms, or 370–420 ms on the netbook in Chromium. I totally fucked up by using division! `time(() => laap(repeat(0, 8), 500000000))` takes 5.36–5.37 seconds, so 93 million loop iterations per second.

I was going to say, "This explains why there was no difference (usually) between JS arrays and typed arrays," but it turns out `time(() => laap(new Float64Array(8), 500000000))` is still three times as slow as the plain-Array version above. Still, the better benchmark function probably will make it easier to understand what differences do exist.

Topics

Argentine electric bill

Kragen Javier Sitaker, 2019-12-18 (3 minutes)

Now that I've been living here for a year and a half, the electric utility finally got around to installing an electric meter. My bill for the first 31 days, from 2019-10-28 to 2019-11-28, is 68 kWh, and it works out to AR$497.98. Most of this, AR$278.94, is the installation of the meter, although supposedly we'd already paid AR$2000 for that; the remainder is divided into a "fixed charge", AR$21.37, and a "variable charge", AR$90.62. This adds up to AR$390.93, to which are added a "municipal contribution" of 6.383% or AR$24.95 and the value-added tax of 21% or AR$82.10, for the total of AR$497.98.

However, they explain they're actually only charging 15 of the 31 days, perhaps because that's when they installed the meter. The "fixed charge" is AR$43.46 per 31 days (presumably this has to do with maintaining the electrical connection?) while the "variable charge" is given as a "unit price" of 2.832, which is multiplied by 32 kWh (apparently 68 multiplied by 15/31 and rounded down, although this calculation makes no sense) to get AR$90.62.

So, in effect, I'm being charged AR$2.832 per kilowatt hour, plus 27.383% of taxes, for a total of AR$3.60748656 per kilowatt hour. At the AR$64.50 per US$ rate that was current last weekend when I checked the prices in Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196), this rounds to US$0.056/kWh, rounding. In diagonally printed letters that make it difficult to read several of the numbers, the bill helpfully explains, "CONSUMO CON SUBSIDIO DEL ESTADIO NACIONAL", consumption subsidized by the national government, even though I'm being charged AR$82.10 of VAT on it.

The new president has just announced a 180-day freeze in electricity and gas prices, which presumably means that this already relatively low energy cost will drop by another 15% or so as our currency inflates --- though whether the drop is 10% or 40% is really anybody's guess at this point.

68 kWh in 15 days is 190 watts, which is a surprisingly low consumption level given how much of the time I've had the air conditioner running, as well as cooking on an electric stove. Presumably this would be doubled if I had a refrigerator. The apartment is wired for 66 amps at 240 V, of which 20 goes to the stove. 66 amps would be 15.8 kW, which would be US$21.24 per day or US$646 per month at this price. Such great consumption might, however, change the category of service and cause variability in the fixed charge.

This at last tells me the cost of the energy to run the air conditioner. If I can trust the label, it sucks 4.5 amps and delivers 2700 kcal/hour (= 3100 W) of cooling. 4.5 amps at 240 V is 1080 W, so that's AR$3.90 per hour or, at the moment, US$0.06 per hour. In all likelihood, the depreciation on the air conditioner is greater than that; various similar portable air conditioners cost AR$25000, which is about 6000 hours of operation.

Topics

Flexible text query

Kragen Javier Sitaker, 2018-07-14 (4 minutes)

Here's an example of the problem I want to solve. I have a crontab which records my battery capacity estimates from the OS once a minute; here's a sanitized transcript:

```
$ crontab -l
# m h  dom mon dow   command

*  *  *  *  *  (date; date +date=\%s; cat /sys/class/power_supply/BAT0/ueven
ot) >> .battery-samples
$ tail ~/.battery-samples
POWER_SUPPLY_CAPACITY_LEVEL=Normal
POWER_SUPPLY_SERIAL_NUMBER=
Sat Jul 14 16:40:01 -03 2018
date=1531597201
POWER_SUPPLY_NAME=BAT0
POWER_SUPPLY_STATUS=Discharging
POWER_SUPPLY_PRESENT=1
POWER_SUPPLY_TECHNOLOGY=Li-ion
POWER_SUPPLY_CYCLE_COUNT=0
POWER_SUPPLY_VOLTAGE_NOW=11400000
POWER_SUPPLY_POWER_NOW=20508000
POWER_SUPPLY_ENERGY_FULL=45828000
POWER_SUPPLY_ENERGY_NOW=24886000
POWER_SUPPLY_CAPACITY=54
POWER_SUPPLY_CAPACITY_LEVEL=Normal
POWER_SUPPLY_SERIAL_NUMBER=
Sat Jul 14 16:41:01 -03 2018
date=1531597261
POWER_SUPPLY_NAME=BAT0
POWER_SUPPLY_STATUS=Discharging
POWER_SUPPLY_PRESENT=1
POWER_SUPPLY_TECHNOLOGY=Li-ion
POWER_SUPPLY_CYCLE_COUNT=0
POWER_SUPPLY_VOLTAGE_NOW=11400000
POWER_SUPPLY_POWER_NOW=20565000
POWER_SUPPLY_ENERGY_FULL=45828000
POWER_SUPPLY_ENERGY_NOW=25216000
POWER_SUPPLY_CAPACITY=55
POWER_SUPPLY_CAPACITY_LEVEL=Normal
POWER_SUPPLY_SERIAL_NUMBER=
```

Now suppose I want to plot battery capacity over time. Getting the capacity itself is easy enough:

```
$ grep -a _FULL= ~/.battery-samples
...(29000 lines omitted)...
POWER_SUPPLY_ENERGY_FULL=45828000
POWER_SUPPLY_ENERGY_FULL=45828000
POWER_SUPPLY_ENERGY_FULL=45828000
$
```

(The -a is necessary because there's a block of 541 NULs that got in there last Wednesday, presumably due to some kind of filesystem corruption on power loss.)

But this only gives me the Y-coordinate. The X-coordinate of time is missing.

Now, I could write it this way:

```
$ perl -lne '$date = $1 if /date=(.*)/;
             print "$date $1" if defined $date
                              and /_FULL=(.*)/' ~/.battery-samples
```

And I can plot that with gnuplot, and it looks right:

```
$ perl -lne '$date = $1 if /date=(.*)/;
             print "$date $1" if defined $date
                              and /_FULL=(.*)/' ~/.battery-samples |
  gnuplot -p -e "plot '-' with linespoints"
```

And that works. But it's a relatively large amount of hacking for a fairly simple task. If we want to include both POWER_SUPPLY_ENERGY_NOW and POWER_SUPPLY_ENERGY_FULL, it's going to start to be complicated.

What I really want here is an interaction like:

- Show me the lines that say date=.
- Okay, now infer POWER_SUPPLY_ENERGY_FULL from the next line that says POWER_SUPPLY_ENERGY_FULL=
- Okay, now infer POWER_SUPPLY_ENERGY_NOW from the next line that says POWER_SUPPLY_ENERGY_NOW=.
- Okay, now display just date, POWER_SUPPLY_ENERGY_FULL, and POWER_SUPPLY_ENERGY_NOW as columns.

At the command line, this could be something like:

- q2 date=
- q2 date= +_FULL=
- q2 date= +_FULL= +_NOW=
- q2 'date=(.*)' '+_FULL=(.*)' '+_NOW=(.*)'

For logfile processing, it's common to want to limit matches to a particular request ID and to exclude "noise" events based on some other kind of pattern. So it's useful to conceptualize this process as the repeated execution of some possibly nondeterministic program:

- First, find date=, and save what comes after it; discard upon fail.
- Then, search forward for _FULL=, and save what comes after it, discarding upon fail; then return to the position from step 1.
- Then, search forward for _NOW= and save what comes after it, discarding upon fail; then return to the position from step 1.
- Then, display the three saved strings.

You could imagine, for example, running one of these subordinate steps on the set of lines that contain "id=$1 ", where $1 is a previously captured id. You don't want to necessarily constrain the entire rest of the query to do that. And you might want to be able to emit nested structures here, and exclude domains in a known spammer list, and whatnot.

This is pretty similar to what I need for my mailreader qyap: I have a nested structure of mail message threads to extract from a possibly out-of-order mailbox (or more than one), and I might want to hide particular threads or subthreads.

(I've done something like this previously with batchagenda.py.)

Topics

The Z-machine memory model

Kragen Javier Sitaker, 2017-07-19 (4 minutes)

I was reading about the Z-machine and playing around with "Lists and Lists", which is a Lisp tutorial interactive-fiction game running on the Z-machine, playable online in Parchment at http://eblong.com/zarf/zweb/lists/.

I was struck by the fact that the basic Z-machine memory model has a combination of two properties very rarely found together: it has no dynamic allocation (and thus no possibility of out-of-memory errors and no need for a garbage collector), but it supports flexible collections and relationships of objects, like Lisp. Also, the basic operations on its memory model are constant-time.

The basic memory model of the Z-machine is a fixed set of objects ("Things", originally) which are referenceable by integer IDs, arranged into an ordered tree. (Or, really, a forest, since there can be any number of separate roots.)

Mutating the tree structure is normally done with this single Z-machine opcode (from http://inform-fiction.org/zmachine/standards/z1point1/sect15.html):

insert_obj: Moves object O to become the first child of the destination object D. (Thus, after the operation the child of D is O, and the sibling of O is whatever was previously the child of D.) All children of O move with it. (Initially O can be at any point in the object tree; it may legally have parent zero.)

Navigating the tree structure is done with three opcodes:

get_child: Get first object contained in given object, branching if this exists, i.e. is not nothing (i.e., is not 0).
get_parent: Get parent object (note that this has no "branch if exists" clause).
get_sibling: Get next object in tree, branching if this exists, i.e. is not 0.

(In addition to the tree structure, the objects have arbitrary mutable fields ("properties") drawn from a small set of 64 field IDs, with values containing a variable amount of data up to 64 bytes, but usually 1 or 2 bytes; they also have 48 boolean "attributes". And they have a "short name" too. Each of the 64 possible properties has defined a "default value" in a "property defaults table" which is returned when you try to read that property from an object that doesn't have it. These aspects of the object memory are somewhat incidental, but without some way of associating other data with the objects, the whole tree thing would be almost pointless. Also, you can access memory as an array of bytes.)

In order to make get_parent and insert_obj constant-time and fast, each object has a redundant parent pointer.

The typical kinds of interactions in text adventure games involve doing things like listing the objects in a room, or searching through the objects in a room to see which one the player is referring to, and whether that might be ambiguous; they aren't sensitive to the ordering of the objects in the room.

So I was wondering what kind of Lispy generic functions you could write on top of this ruthlessly-side-effect-filled structure. Clearly you can write a constant-space function that visits every object in a subtree, for example, or moves all the objects in a subtree, or all the children, to a given other object. But can you write

general-purpose utilities to raise the level of abstraction at which you program? It seems like CLU-style coroutine iterators might be the best you can do in many cases.

Topics

Fukushima leak

Kragen Javier Sitaker, 2014-04-24 (6 minutes)

The recently-revealed continuing leaks of radioactive contaminants from Fukushima, although they are 500 times smaller than the initial release and 100 million times smaller than the natural radioactivity of the Pacific Ocean, could still be dangerous to local ecology and human health, but does not represent a global catastrophe.

This is my conclusion as a non-expert in the field summarizing the publicly available information. It could be wrong.

Details follow.

Different accounts give different amounts of radioactive water leaked into the Pacific from the Fukushima nuclear plant via continuing leaks. Some of them give the amount, rather uselessly, in tons, but the better accounts give the amount of radioactive material in the water in becquerels. One TV report says it's a PBq, but that sounds like it's probably referring to part of the initial, much larger release; other reports give the amount as 10 to 50 TBq: More Fukushima Fallout and a Japan Times article say 30 TBq, Asahi Shimbun's article says 24 TBq, while National Geographic's article gives the groundwater concentration of radioactive cesium in places as around 1kBq/kg, gives the total release as 0.3 TBq/month, describes the immediate aftermath of the disaster as a release of around 10PBq, and contextualizes it by comparing to the 89 TBq release of cesium-137 from the Hiroshima bombing.

The raw becquerel numbers are sort of meaningless without context on how big a becquerel is. Mijlkovic's 2012 article says:

> 9.3 percent of the catches exceeded Japan's official ceiling for cesium, which is 100 becquerels per kilogram (Bq/kg). ... Canada's much higher ceiling, which is 1,000 Bq/kg

Comparing to natural radioactivity, typical rocks and dirt have hundreds of Bq/kg, mostly due to potassium, but with significant amounts due to uranium, thorium, and radium. Seawater has on the order of 10 Bq/kg, almost all due to potassium. A human body contains on the order of 10 kBq, mostly due to potassium and carbon-14. The Pacific Ocean naturally contains about 7000 EBq, because it contains about 700 million km³ of water.

That means that, purely in terms of increased radioactivity, the leak is so small as to be insignificant. A 70 TBq leak of radioactivity would be 100 000 000 times smaller than the natural radioactivity of the Pacific Ocean. Increasing the radioactivity by one part in a hundred million is not dangerous; if all radioactivity were equivalent, it wouldn't even be detectable. Even the initial 10PBq release was only enough to increase total Pacific Ocean radiation by about one part per million.

The fatal dose of cesium-137 is on the order of 100MBq/kg for dogs. So a 70TBq leak, if not sufficiently diluted, is enough to kill about 700 000 kilograms of dogs or other similar animals, such as people — about ten thousand people.

However, it's sensible to have different safety limits for different radioactive elements, because some of them, like potassium, are biologically regulated at a constant level — so eating more radioactive

potassium probably doesn't increase your exposure to radiation at all — while others bioaccumulate, like strontium-90, which replaces calcium in your bones and can therefore continue irradiating you for the rest of your life. Cesium bioaccumulates to some extent, so it becomes more concentrated in animals than in plants, more concentrated in predators than in herbivores, and more concentrated still in secondary predators like tuna. Tritium, a third radioactive contaminant in this case, is not known to bioaccumulate, but it also isn't homeostatically bioregulated like potassium; chemically, it's almost identical to hydrogen.

Also, the radioactive material is not evenly distributed. Ocean-caught fish from near the Fukushima reactor had levels up to tens of kBq/kg in 2012, hundreds of times higher than normal fish, and we can reasonably expect that fish that feed in areas where the water has been released will continue to be contaminated to much higher than normal levels, perhaps 100 times higher than normal. However, it's believed that the water that has leaked has already had most of its radioactive cesium removed, unlike the water that leaked early on; strontium-90 may be a bigger concern now.

So there are real health concerns, but they are not very large with the current size of the leak.

The National Geographic article says that the total amount of contaminated water stored, now and in the future, is on the order of a million tons; only a third of that is there now. Unfortunately, I don't have a good handle on how many becquerels that water contains. It appears that the total water loss was 300 tons, mixing into 400 tons per day of groundwater; that gives us roughly 30 TBq/300 tons or 11 MBq/kg, about a million times more radioactive than natural seawater. A million tons of 11MBq/kg water would add up to 10PBq, roughly the same size as the initial Fukushima disaster. So, in a worst-case scenario where all the tanks vented into the ocean, it would be roughly comparable to the initial incident, and despite bioaccumulation of strontium, probably would not be enough to cause more than local ecological damage.

Topics

Transactional screen updates

Kragen Javier Sitaker, 2015-04-01 (10 minutes)

So I was thinking about one of my usual hobbyhorses: how to design an entire usably efficient interactive computing system as a pure function of some mutable, but in some way orderly, state.

History

The basic idea goes back to McCarthy's "Elephant 2000" proposal, where you define a data processing system in terms of keeping commitments based on its past history, and then the compiler figures out what information the data processing system needs to store about that past history to comply with its commitments. That is, the entire data structure of the system is purely the result of compiler optimization. You just make assertions about the relationships between past events and the system's outputs, and the compiler produces a running system that satisfies those requirements.

I explored it a bit in "rumor-oriented programming", where the idea was that you accumulate an ever-growing history of "rumors" that get automatically synchronized between your devices (because you only ever add new ones, synchronization is trivial and guaranteed to be convergent, though potentially expensive) and the actual screen you see is the output of a query over that rumor history, using a query language that doesn't expose the order in which the rumors arrived on the current device; then you have buttons and shit that add new rumors and possibly cause like a recomputation of the query result you see on the screen.

I also explored it a bit in "dependency-driven composition, or make for websites", where the idea is that you use a `make`-like build process over a dependency graph of resources. Some resources are human-produced and editable, while others are produced on demand from other resources, and then cached.

This is all closely related to functional reactive programming, which is perhaps unsurprisingly (at long last!) getting a lot of attention in the last couple of years in the JavaScript world, with React, Angular, and Meteor all becoming suddenly popular.

Automatic dependency detection

A simpler model than `make` for dependency-driven recomputation in the world of building software projects is `redo`, which was designed by Daniel Bernstein and later implemented by Avery Pennarun. The idea of `redo`, as I understand it, is that the script to build a file `foo` is contained in the file `foo.do`, which is a shell script which recursively invokes `redo` to ensure that each of the things `foo` depends on is up to date, and then does whatever it needs to do to build `foo`. Because `redo` knows that it was trying to build `foo`, it can tell that each of those recursive calls represents an edge in the dependency graph.

This is closely reminiscent of how optimistic STMs (software transactional memories) detect transaction conflicts and automatically retry the transactions. Within a transaction, you read some set of STM variables and write some other set of them, and the STM records each of those sets; when the transaction goes to commit, it

atomically publishes your set of writes, but only if no transaction has written to some variable that you read since the time that you read it. If so, your results are discarded and automatically retried. (This is also, I think, closely related to how speculative instruction execution works in CPUs, although I don't know the details.)

In a sense, the STM has concluded that since it's possible all of the writes could have a dependency on any of the reads, so it needs to discard them and rerun the transaction if any of those variables that have been read has changed.

This is very similar to how Meteor detects dependencies in its dependency graph: if a Meteor-controlled reactive computation reads from a reactive variable, that variable records a dependency, which it can then choose to invalidate later. Unlike `redo`, and similar to optimistic STMs, this causes that reactive computation to be eagerly rerun, a policy choice which I suspect can lead to exponential-time execution from diamonds in the dataflow graph.

The delightful thing about this kind of tracking, taken advantage of by all of Meteor, optimistic STMs, and `redo`, is that it doesn't depend at all on the internal functioning of the computation that it's re-executing; it only has to be able to re-execute it on demand and intercept its I/O, and trust that it is deterministic. You could even imagine an "STM server" which provides these services over the network, although it has to trust its clients to retry when they get a commit rejected.

(I confess I haven't actually used Meteor or `redo`, so I could be totally confused about them.)

The other interesting thing about it is that, usually at least, you can do a simple version of it with very little actual dependency tracking. You can just re-execute stuff blindly. This is how Pennarun bootstraps `redo`.

Dependency-driven and immediate-mode GUIs

So, what if we try to take this kind of thing all the way to pixels on the screen? Like, it would be super cool if I could write some kind of straightforward computation over, I don't know, an editor buffer of text in some form, some kind of specification of where I'm scrolled to in that text, and a font, that results in setting some pixels on the screen, and results in smallish recomputations. It would be even better if I could, like, write a thing that parses HTML and CSS and lays out boxes on the screen and draws glyphs into them, and have that be able to automatically detect changes in the CSS and figure out which parts of the screen they affected.

A crucial part of this is that if you want the dependency-tracking overhead to be small, the dataflow graph probably has to be relatively small, which means the nodes in it have to be fairly large; and if you want a recomputation to be small, the nodes in it have to be relatively small. To be concrete, in the editor example, if there's a change to a part of the text that isn't displayed on the screen, you'd probably prefer to not have to recompute the entire screen contents from scratch; and maybe if there's a change on one line of the screen, it would be nice to redisplay just that line.

There's a somewhat relevant field of work called

"immediate-mode GUIs", popular in videogames, where you don't try to avoid redrawing the screen, part of the reason being that in a videogame you typically redraw the screen 60 or 120 times a second anyway. Immediate-mode GUIs don't have data objects that stick around in memory in some kind of nested window tree structure, the way regular ("retained-mode") GUIs do; instead you do things like this:

```
set_drawing_background(start_button_background);
if (button(120, 200, 600, start_button_height, "Start!")) start_game();
```

which redraws the button in the specified position, with its possibly updated height, and then checks to see if the user has just clicked their mouse in it. The button has no persistent existence in memory, other than as pixels in the framebuffer that have changed color, but while control flow is down inside of it, it exists on the stack and can react to the mouse click.

I feel that immediate-mode GUIs are a lot easier to program — for example, the above example doesn't have to add an event listener to redraw when `start_button_background` or `start_button_height` change, nor does it have to worry about when to destroy the button object — and they use less RAM.

The analogy here is between immediate-mode and retained-mode graphics systems. `<canvas>` is immediate-mode; SVG is retained-mode. Both have their performance advantages. One of the potential disadvantages of immediate-mode drawing is that you have to draw all the pixels on the screen every frame, even when they haven't changed.

So, suppose you execute some series of transactions to draw your immediate-mode GUI on the screen. Each transaction updates some region of pixels on the screen, having read some set of reactive variables (the window size, the text buffer contents, the clock, etc.) and executed some deterministic computation driven by them.

Can we use this to improve the efficiency of immediate-mode GUIs in the case where the entire screen is not actually changing? It's not super straightforward, but I think so. If your `start_button_height` is tracked by something like Meteor, you can automatically rerun the draw-button transaction; and if it fails to overwrite pixels that it was previously overwriting, you can rerun the transactions that generated those pixels in order to get them back. (Or maybe you could retain the pixels in a backing store.)

If you're doing alpha compositing, of course, you may have to redraw the background pixels you're blending the button with first.

Topics

- Transactions (p. 3755) (14 notes)
- JS (p. 3533) (12 notes)
- Immediate-mode GUIs (p. 3515) (8 notes)
- Dependencies (p. 3405) (7 notes)
- HTML (p. 3508) (6 notes)
- Dataflow (p. 3401) (5 notes)
- Sync (p. 3737) (4 notes)
- Stms
- Redo

¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires?

Kragen Javier Sitaker, 2017-04-17 (2 minutes)

Depende. Hice algunos cálculos que sugieren que sí. Si tenés 40m² de techo, estamos a casi 35° de latitud, y en el solsticio el sol llega a casi 24° de latitud de norte, tenés una elevación de 90°-59° = 31°, y sen 31° = 51.5%, así que tu techo de 40m² cuenta con solo 20.6 m² de sol; el constante solar es alrededor de 1kW/m², así que tenés tipo 21 kWt disponible al mediodía solar. Si supongamos que tenemos tipo un tercio de eso en promedio en el recorrido del día, tenés un promedio de 7kWt; si su colector es de una eficiencia de 50%, cosa fácilmente alcanzable, será de 3.5kWt. Si lo estás usando para calentar agua de 10°C a 40°C para tu casa, lo cual necesita (40-10)·4.2 kJ/kg ≈ 126 kJ/kg, podés calentar tan solo 2400 kg de agua por día. Pero probablemente necesitás menos de 100 para ducharte y lavar los platos y la ropa. Así que un día de sol te da lo suficiente de calor para aguantar 23 días de nubes sin calentar tu agua con gas.

Hay otros usos de calor en la casa, tales como calentar el aire, cocinar, secar ropa, y deshidratar frutas, pero son tan pequeños al lado del costo de calentar el agua que podemos considerarlos errores de redondeo.

Así que, para mí, es fácilmente alcanzable con la cantidad de sol que nos alcanza hasta en el medio del invierno. La cuestión es construir la cantidad de colector solar y que puede bancar una diferencia de temperatura de más de 30°C.

--- de https://www.facebook.com/groups/593966090752422/645596198920°02744/?comment_id=646053005543730&notif_t=like&notif_id=1466°0631246627256

Topics

- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Thermodynamics (p. 3747) (49 notes)
- Household management and home economics (p. 3504) (44 notes)
- Solar (p. 3717) (30 notes)
- Water (p. 3773) (13 notes)
- Heating (p. 3498) (9 notes)
- Español (6 notes)

Rubber wheel pinch drive

Kragen Javier Sitaker, 2019-08-16 (updated 2019-08-18) (8 minutes)

I ran across some discarded stroller wheels on the way home, which I resisted the temptation to bring home. They had hard rubber around the outer rim (of about 200-mm diameter and 20-mm thickness), and it occurred to me that a motor shaft pressed directly against that outer rim would be an effective way to drive it with a fairly large mechanical advantage for mobile robotics.

Pressing the motor shaft against the wheel

Suppose we mount a motor above the wheel with its axis parallel to the wheel and its 10-mm-diameter shaft running across the top, held in one bearing on each side of the wheel, each of which is pressed toward the wheel axis by some kind of mounting preload, thus pressing the shaft into the rubber. Steel on rubber has a frictional coefficient around 1, so the bearing radial load needs to be a bit larger than the maximum drive force we need to be able to apply to the wheel. And we have about a 20:1 mechanical advantage in the torque sense (but 1:1 in the sense of linear motion).

Let's say the robot weighs 10 kg and thus 98 N, so we want at least 30 N of force to be able to handle reasonable slopes with some reasonable acceleration and deceleration. With the 5-mm radius of the shaft, we need 0.15 N m of torque from the motor, or 0.11 foot-pounds in medieval units. This is a reasonably large amount of torque; many small electric motors cannot provide it. Ten watts at 30 N is 330 mm per second, a very slow walking sort of speed, at which speed the wheel would be turning 1.9 revolutions per second (113 rpm), and our hypothetical 10-mm-diameter shaft 38 revolutions per second (2300 rpm), which is not an outrageously low speed for a modern small motor, but somewhat lower than optimal. Also, the 10-mm shaft is enormously larger than most small motors have.

So it would be nice to be able to use a smaller shaft in this way, thus getting more mechanical advantage and not needing to attach a thicker shaft to the motor. But if we take a shaft that's much thinner than 10 mm, mount it through bearings on both sides of the wheel 20 mm apart, and press down on the bearings with, say, 50 N or 100 N of force, the shaft will probably bend in the middle, yielding, and be ruined.

A 10-watt NMB motor

I don't know what a typical motor in this power range is like these days, but on the strength of Digi-Key having 221 of them in stock, I'll consider the Minebea DIA42B 10W 31A as a representative. Digi-Key charges US$33 for it, which seems high.

The DIA42B 10W is a ten-watt 24-volt BLDC motor optimized for 500–3500 rpm (running up to 4000 rpm with no load at 300 mA), and although it's only rated for ten watts, it's also rated for ten amps. It weighs 150 grams and it's 42×42 mm with a 6-mm-diameter shaft. It has a 100-pulse-per-revolution rotary encoder. The torque curve on p. 13 of the datasheet shows it at almost 6000 rpm at no load, 60 mN m of torque at 3500 rpm, 90 mN m of torque at 2000 rpm,

100 mN m of torque at 1000 rpm, and about 105 mN m at its 500-rpm minimum speed. At these high torques it's sucking up to 2 amps of current, so if you keep it up it'll burn up; what I think is its top sustainable current of about 400 mA would give you about 20% to 40% of those torques, about 20 mN m.

At the surface of its 6-mm-diameter shaft, 20 mN m works out to 6.7 N, which is significantly lower than 30 N but would still permit significant robot movement, especially if the robot ends up lighter than suggested above. Clearly gearing it up further by using a thicker shaft would be bad!

Rollers

So, how could we press this 6-mm shaft against the wheel, other than running it between bearings 20 mm apart on opposite sides of the wheel? We could press it onto the wheel by backing it with two rollers, not themselves in contact, which trap it in a triangle of compression between the wheel and the rollers.

But the rollers can't be very large. If they're 6 mm in diameter themselves, then they could be as far as 180° apart viewed from the motor shaft before they hit the wheel, but then we haven't really solved the problem, just reduced it by a factor of 2 or 3. If we make them larger, we can improve that situation, but soon they hit the wheel.

However, if we make the rollers out of stacks of parallel discs with spacers between them on a shaft, like hard disk platters, we can *overlap* them, though not arbitrarily far. If the shaft diameter approaches zero, each disc of roller A can extend to the center of roller B, and vice versa; with a physically plausible shaft diameter, they need to be somewhat further out. The discs could be, for example, 1.4 mm in thickness with a 1.6 mm space between them, with somewhat rounded edges, so that the 6-mm shaft only needs to span spans of about 2 mm unsupported from one direction.

For example, you could imagine 16-mm-diameter discs on 9-mm shafts (including the spacers), yielding 11 mm distance from roller center to motor-shaft center and 13 mm distance between roller centers (including ½ mm of clearance). This means that the motor shaft center will be sqrt($11^2 - 6½^2$) mm or about 8.87 mm away from the roller-center-to-roller-center axis. This means that the bottom of the motor shaft will be 11.87 mm from the roller-center-to-roller-center axis, while the roller edges are only 8 mm from it, so they won't touch the wheel. They could even be a little bigger than that.

Bigger rollers would mean not only more stiffness but also improved leverage to overcome the friction in the rollers' bearings, and thus less friction loss.

A quadcopter motor

Consider a higher-powered modern BLDC motor, the Turnigy BC2836-8 quadcopter motor I used as my example in Drone cutting (p. 1106). It has a 4-millimeter shaft and delivers 336 watts at 15000 rpm on 14 volts; this suggests an output force of almost 100 newtons at the surface of the 4-mm shaft --- ample to drive the wheel in this way, but definitely needing something like this roller system.

If you try using the same two rollers, the motor shaft center will be

sqrt(10² - 6½²) mm or about 7.6 mm from the roller-center-to-roller-center axis, and the shaft edge 9.6 mm from it, slightly over the 8 mm of the rollers themselves.

Multiple motors

You could reasonably easily connect multiple motors to the same wheel in this fashion, just clamping them on around its upper rim.

Varying wheel sizes

The wheel in this system is only being used to transfer linear motion from the surface of some motor shaft to the road or floor, so the calculations here are entirely insensitive to the wheel size --- it would work in precisely the same way with rollerblade-sized wheels as stroller-sized wheels, applying the same amount of force to the road at the same speeds. The only relevant difference would be that, driving a smaller wheel, the rollers would have a bit more clearance to avoid contacting the wheel.

Topics

- Mechanical things (p. 3569) (45 notes)
- Ghettobotics (p. 3472) (18 notes)
- Robots (p. 3688) (9 notes)
- Motors

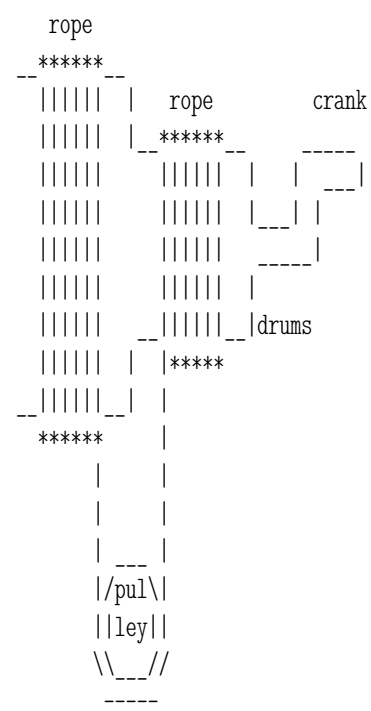
Spiral chinese windlass

Kragen Javier Sitaker, 2019-07-23 (updated 2019-07-24) (7 minutes)

(Related to Differential spiral cam (p. 512), which describes a way to use a similar mechanism to get a complex programmed sequence of motions from a simple mechanism.)

The standard Chinese windlass mechanism

The standard Chinese windlass mechanism gets a very large mechanical advantage from a simple mechanism by using a pulley as a differential:



Rope is wound in the same direction around two drums of different sizes which rotate together, typically turned by a hand crank, or by another wheel with an endless chain or rope or belt around it; rotating the drums will always pay out rope from one and take up rope from the other. The rope runs down from one drum, around a free-hanging pulley, and back up to the other drum. So the pulley moves up or down by half the difference in the rope paid out from one drum and taken up by the other.

Through the magic of the differential mechanism, this very simple mechanism provides an arbitrarily large mechanical advantage, inversely proportional to the absolute difference in diameter of the two drums. If the two drums are equal in size, the mechanical advantage is infinite.

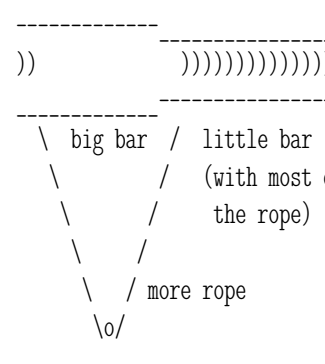
To take a more practical example, if you want to lift two tonnes (20 kN) by exerting 100 N on a 200-mm-radius crank, you need a mechanical advantage of 200: a 2-mm difference between the two drums will provide you with this. (That's in radius; it's a 4 mm difference in diameter.) Perhaps one drum is 50 mm in diameter, while the other is 54 mm in diameter, or perhaps one is 400 mm in diameter, while the other is 404 mm in diameter (although it may be

difficult to find such a drum); the mechanical advantage is the same either way.

Let us suppose that we use a 50-mm and a 54-mm drum, or rather bar. Each lifting revolution lifts the pulley by 2πmm ≈ 6.3 mm, paying out π50mm ≈ 157 mm of rope from one bar and taking up π54mm ≈ 170 mm of rope on the other bar. Lifting the two-tonne weight by 2000 mm requires some 320 turns of the crank, paying out 50.24 meters of rope from one bar and taking up 54.4 meters of rope wrapped onto the other.

Some difficulties with it

This has the somewhat annoying result that you will need 55 meters of one-tonne rope or webbing wrapped around your windlass to lift the two tonnes by only two meters. This rope might average 3 mm thick, and if we want predictable mechanical advantage, it cannot freely wrap in multiple layers — that would increase the effective diameter of the windlass by 6 mm. And we need about a meter of length of each drum, since we need 320 turns of 3-mm rope, one right next to the other. This has the rather dismaying result that our pulley hangs *down* by two meters but *across* by half a meter relative to its attachment point on each bar:



This is going to put rather significant side forces on the rope that will attempt to slide it along the length of the bar, add extra tension to the rope that isn't part of the load it's bearing, and change the mechanical advantage.

The helical-grooved cone pulley

So, the alternative I was thinking about is that, instead of two drums, you can use a *single* tapered conical pulley with a single spiral groove on it, and run the pulley between two near-adjacent turns of the groove, one with a radius 2 mm greater than the other.

For example, suppose the groove is 3 mm wide, and there is a 16.5-turn blank spot with no rope on it in between the takeup part and the payout part. This makes the blank spot 49.5 mm long and means that the pulley tapers by 2 mm of radius every 49.5 mm of its length, an included angle of 4.6°. Because you still need 320 turns of rope, you still need a meter-long tapered cone, but now it's only one meter long instead of two, or 1.01 m to be more exact. Its radius at the wide end is 40.8 mm greater than at the narrow end, and the pulley hangs from two points that are only separated by 49.5 mm of length and, say, 40 to 80 mm of width.

(You might be able to squish adjacent turns of rope together a bit if you skimp on making a full half-circle profile for the groove, or make

it a bit elliptical in cross-section, deeper than a perfect torus, I don't know. Maybe you could shorten the apparatus by a third that way.)

High-precision movement

There are three different reasons for wanting very large mechanical advantages like the one provided by this mechanism. The first, discussed above, is to develop very large forces. (Tapered thread (p. 363) discusses another very simple mechanism that does this using a tapered helix.) A second is precisely the reverse — achieving very high *speeds* but low forces by applying force to backdrive the mechanism through its stiffer part. In this case this is not applicable because the frictional losses are too large, and this self-locking behavior is considered a feature in the original use of the Chinese windlass to lower buckets down wells. The third is to achieve very high *precision* by precisely translating a reasonable-sized movement, such as moving the crank by a millimeter, into a much smaller movement, such as moving the pulley by ten microns. Differential screws are an analogous mechanism commonly used for such high-precision movements.

With high-rigidity materials like UHMWPE, this mechanism may be applicable to such uses, in particular in parallel kinematics; the groove guiding the cable should reduce uncertainty about where the cable is anchored to, and hexapod-crane-type mechanisms can be considerably more rigid for a given total mass than mechanisms that rely on resisting bending moments over long distances. Particularly with UHMWPE's high lubricity on steel, it may be possible to dispense with the movable pulley and its bearings and simply run the cable through a metal eye, but the bearings of the conical drum will still be a source of error.

I worry, though, about the unknown and presumably load-dependent amount of squish in the transverse dimension of the cable as it wraps around the drum.

Topics

• Mechanical things (p. 3569) (45 notes)
• UHMWPE (p. 3762) (11 notes)

Transistors vs. Microcontrollers

Kragen Javier Sitaker, 2018-06-17 (updated 2018-07-05) (8 minutes)

I was just looking at the ATTiny13A on Digi-Key, the one in an 8-pin SOIC. It costs 33–40¢, depending on quantity, has six GPIO pins, runs at anything up to 20 MHz (or 0.128, 4.8, or 9.6 MHz on the internal RC oscillator), and can source or sink 20 mA per output pin, up to a max of 60 mA across all the pins. Its input pin bias current is 1 µA. It has the usual assortment of AVR peripherals, cut down a bit: one 8-bit timer/counter with two PWM channels, a separate watchdog, a 10-bit ADC with an internal voltage reference (same speed as the one on ATMega328, 15ksps at max resolution), and a comparator. It only has a kilobyte of Flash and 64 bytes of RAM. It's 5.4 mm × 5.35 mm × 2.2 mm in the SOIC version.

So far this sounds like a pretty wimpy microcontroller, even if it is an AVR. But I think maybe instead of comparing it to microcontrollers I should compare it to transistors.

(Incidentally, the same chip also comes in a much smaller MLF package with 20 pins, which gives you 18 GPIOs instead of 6. And the ATTiny5 costs 17¢ now, but is limited to half the memory, 12MHz, and 8 bits of ADC resolution, but has a 16-bit timer — in 2 mm × 2 mm.)

(And the STM32F031x4/6 comes in a WLCSP25 package that is 2.5 mm × 2.4 mm and has 20 GPIOs, or 18 GPIOs and an I²C interface, or 16 GPIOs and an SPI interface. It can handle somewhat less current or voltage than the AVRs but is not at all wimpy, rather the opposite. I think it costs 56¢ but I may have it confused with a larger package of the same chip.)

For example, a 2N7002T-2 MOSFET costs US$0.51 (or down to 9.4¢ in quantity 1000), about 30% more than the microcontroller (again, in quantity 1). It can switch 115 mA at up to 60 volts, a much higher voltage than the microcontroller but less than twice the current, and its input bias current is 0.1 µA, ten times higher. It can do analog things the AVR can't. But you need 10 volts to turn it all the way on, it can only reasonably be used up to a few megahertz, and its biggest disadvantage is that it has only one input and one output.

Depending on the circumstances, you could replace as many as six 2N7002s (US$3) with a single ATTiny13A (US$0.40). That's nearly an order of magnitude cheaper, and it's smaller too.

But maybe it's unfair to compare the microcontroller to power MOSFETs, even lightweight ones like the 2N7002. Maybe you should be comparing it to small-signal transistors, like a 300MHz MMBT3904, which only costs 10¢ and can handle 200 mA at up to 40 V, or a 12¢ BC849, which is 30V and 100mA, which is good to 100MHz. But those still cost more per output pin than the microcontroller. And typically they come in a SOT-23 package, which is 2.9 mm × 1.3 mm × 1 mm, larger than many of the microcontroller packages mentioned above.

It seems like if you just want to turn things on and off, even very fast, the microcontroller is best up to 60 mA at 5 volts — 300 milliwatts — and perhaps 5 megahertz out, 7 kilohertz analog in. If you need higher voltage, higher current, analog output, or higher

frequency, then maybe you should go with discrete transistors or perhaps other ICs.

This suggests a real potential niche for microcontrollers with open-drain GPIO outputs, which could potentially directly switch much higher-power loads — I suspect 60 V at 20 mA wouldn't be unreasonable, and would be over a watt without dissipating any more power in the μC itself than an ATTiny does. Hmm, and any I²C/TWI interface already has open-drain hardware… I wonder if it's possible to take advantage of this?

The STM32 line of microcontrollers has 5V-tolerant GPIO pins that can be configured as open-drain, but the datasheet warns not to try to run them over 5.5V, and gives Vdd+4V as the "absolute maximum". So 60 V is probably unwise.

An interesting question is how you design things to make this potential advantage real. What kind of software do you run on the microcontroller? How much autonomy do you give it? How do you divide up the functionality of a complex device among different microcontrollers?

A difficulty with these super tiny devices with 4 or 6 GPIOs is that you don't have much left over for communication. An ATtiny48, by contrast, has 24 or 28 GPIOs (depending on the package), and three of them suffice for SPI at 3 Mbps. In between 21 remaining GPIOs, you could plausibly charlieplex 441 LEDs, which would be a pretty decent chunk of transistors, or run 10 tiny DC motors in an H-bridge configuration. These packages, not counting the DIP, range from 7 mm square to 4 mm square.

Other possible chips are reasonable alternatives to microcontrollers. There are I²C GPIO extenders, for example, and there are op-amp chips, and there are shift registers, and maybe you could use JTAG. The LM7321 op-amp comes in a SOT-23 (1.75 mm wide, 3.05 mm long, not counting the pins) or dual in a 8-pin VSSOP (3 mm square) or SOIC.

An amusing possible alternative is to use some kind of voltage regulator — Horowitz & Hill suggest treating an LM317 as a cheaper alternative to the LM395 "protected transistor", with "~1.2 V base-to-emitter voltage and 50 μA pullup base current," and mention using 7805s in a similar way despite their annoyingly large base-to-emitter voltage. But you should also be able to use a buck or boost converter in this way, generating a sort of class-D amplifier.

GPIO extenders include the Microchip MCP23008, which Digi-Key classifies as "Interface — I/O Expanders", a category with 1643 chips in it. These are mostly SPI and I²C with 8 or 16 pins and can sink or source a few tens of mA. They tend to cost about 50–100¢. Examples include the 129¢ TI TCA9534APWR, a 400kHz I²C 16-TSSOP (4.4mm square) with 8 push-pull GPIOs that runs anywhere from 1.65–5 V; the 144¢ Semtech SX1502I087TRT, a 400kHz I²C 20QFN (3mm square) with 8 push-pull GPIOs of 8mA source, 24mA sink capacity at anywhere from 1.2 to 5.5 V; and the 150¢ TI TCA9535PWR, a 400kHz I²C 24TSSOP (4.4mm) with 16 push-pull GPIOs of 10 mA source, 25 mA sink at anywhere from 1.65 to 5.5 V; the MCP23008-E/ML, a 113¢ I²C 20QFN (4 mm square) with 8 push-pull GPIOs of 25 mA source or sink at anywhere from 1.8V to 5.5V; and the obsolete 5MHz SPI NXP PCA9701D,112, with 16 input-only pins, in a 7.5mm 24-SOIC. (All prices from Digi-Key

in quantity 1.)

The thing to notice here is that the price per GPIO here is 16¢, 18¢, 9.4¢, 14¢, and ∞¢, respectively. While this is reasonable compared to the price of a whole discrete transistor, it compares unfavorably with the price of GPIOs on a full-fledged microcontroller. And the size, too, compares unfavorably.

Topics

• Electronics (p. 3430) (138 notes)
• Pricing (p. 3646) (89 notes)
• Microcontrollers (p. 3580) (29 notes)

Sous vide

Kragen Javier Sitaker, 2019-04-02 (2 minutes)

I want a temperature-controlled water tank for food preparation, including things like sous-vide cooking. In particular, it would be very valuable to me to be able to do the following:

• Cook eggs in the shell without risk of overcooking them, boiling away all the water, exploding the eggs.
• Heat water for mate to the appropriate mate temperature and maintain it there, rather than boiling water and then mixing it with an unknown amount of water.
• Controlling the temperature of yogurt so that variations between different batches of yogurt, and different jars in the same batch, are not due to variations in the temperature curve. This way I can experiment with different temperature curves. It would be nice if, as a bonus, I can make larger batches of yogurt.
• Cook plastic-wrap-wrapped vegetarian sausages at a known temperature without risk of burning the plastic (e.g., when steaming them in a pot there is the risk that the plastic comes in contact with the pot, which is at a higher temperature than the water, or that the water all boils away).
• Rapidly defrost hermetically-sealed frozen food with a rapid flow of warm water.

There are also fun tricks like cooking part of an egg, or normal sous-vide cooking of meat for long periods of time at low temperatures, but I think of these as much less valuable.

I think this device needs one or more thermometers, a heating element (maybe one of the cheap immersion heaters they sell at truck stops), a relay (or SSR) to control the heating element, a microcontroller, and some kind of water-circulating pump that can handle water at temperature. And, of course, a microcontroller with a well-tuned thermal control system to prevent overshoot.

This is somewhat related to my desire for a dehydrator (see files Notes on circuitry for the Nutra seed activator (p. 3099) and Home dehumidifier (p. 3131)) for food preparation and food waste handling, the main difference being the heat transport medium — water in the case of the sous-vide cooker, air in the case of the dehydrator. The dehydrator would also probably need to measure humidity.

Topics

Assembler bootstrapping

Kragen Javier Sitaker, 2019-07-18 (updated 2019-12-08) (16 minutes)

I've explored the question of bootstrapping computer systems from scratch in some detail, but one of the things I haven't touched on much is the design of the assembler — that is, the compiler from assembly language into machine code. This is somewhat curious, because assemblers have been quite critical in the bootstrapping process of actual computers, historically speaking, and often they have been highly extensible.

Suppose you had a minimal assembler that you could gradually extend into a reasonable high-level-language programming environment, starting with some tiny minimal core and gradually adding functionality. What would it look like?

(See Forth assembling (p. 940) for another take on this.)

Maybe we should start by looking at the history of assemblers.

History

Early computers were divided into the "first generation" of vacuum-tube computers (arguably starting with EDVAC and EDSAC in 1949), the "second generation" of transistorized computers (for example, the popular IBM 7090, first shipped in 1959), and the "third generation" of integrated-circuit computers (such as the System/360 model 85, 1969). Similarly, there were three "generations" of programming languages around the same time — the "first-generation programming languages" were binary or decimal machine code, the "second-generation programming languages" were assembly languages (originating on the EDSAC in 1948, before it actually operated, but becoming popular around 1955 with SOAP for the IBM 650), and the "third-generation programming languages" were high-level languages like COBOL, ALGOL, and FORTRAN, arguably starting in 1951 with Grace Hopper's A-0 compiler or in 1955 with her FLOW-MATIC, but not becoming popular until the 1960s.

Assemblers were adopted more rapidly on drum machines like the IBM 650 because the placement of the instructions was not sequential, and due to the nonuniform access time of the drum, even choosing the addresses for a straight-line sequence of instructions was a difficult optimization problem.

(Many subsequent "generations" have been touted by one or another person, most famously Moore's 1970s language FORTH, the fad for marketing new languages as "4GLs" in the 1990s, and the Japanese Fifth Generation Computing Project; their implicit promises of revolutionary improvement have not been fulfilled.)

From their initial popularity around 1960 until C finally made assemblers obsolete except for niche tasks around 1988, most computer systems software (including the assemblers themselves) was written in assembly language, and during much of this time, even much application software was written in assembly language. Among other very influential software, I believe that all of OS/360, MACLISP, TECO, BASIC-80, GW-BASIC, AppleSoft BASIC, TOPS-10, TENEX, TOPS-20, MS-DOS, SNOBOL4,

SKETCHPAD, WordStar, VisiCalc, Lotus 1-2-3, QuickDraw, nearly all NES games, and early versions of Turbo Pascal were all written in assembly language. Unix was very unusual in being an operating system *not* written in assembly starting around 1972 (although the early versions of Unix, before 1972, were indeed in assembly).

Well into the 1990s, computers were slow enough (and compilers bad enough) that programming in assembly language was occasionally necessary for any "serious" programmer — nearly all software ran into performance bottlenecks that could only be overcome with assembly language or custom hardware. So video game engines and transaction-processing software were routinely written in assembly. Nowadays, this is much less often the case, and a much smaller fraction of modern software is written in assembly, particularly since the bulk of computing power in modern personal computers has shifted to their Tera-like GPUs — now if you want more computing power, you reach for the GPU, not assembly language for the CPU.

Knuth (who wrote the software for his landmark book series The Art Of Computer Programming entirely in assembly language) has suggested that it's reasonable to write an assembler in an afternoon, for a computer that's not overly complicated, anyway. Modern binary executable formats add a certain amount of overhead to this, as do modern complicated addressing modes.

Macros

Serious assembly programming makes use of assembly-language macros, which are considerably more powerful than the macros we're used to in other programming languages. In httpdito, I defined some gas macros that use conditionals to do a little bit of size optimization, providing special shortcuts to set registers to 0, 1, 2, or themselves:

```
### Set dest = src.  Usually just `mov src, dest`, but sometimes
### there's a shorter way.
.macro be src, dest
        .ifnc \src,\dest
        .ifc \src,$0
        xor \dest,\dest
        .else
        .ifc \src,$1
        xor \dest,\dest
        inc \dest
        .else
        .ifc \src,$2
        xor \dest,\dest
        inc \dest
        inc \dest
        .else
        mov \src, \dest
        .endif
        .endif
        .endif
        .endif
.endm
```

This is used in macros like these:

```
.macro sys1 call_no, a
        be \a, %ebx
        sys0 \call_no
.endm

.macro sys0 call_no
        be \call_no, %eax
        ## There's a new, faster instruction for system calls, but I
        ## don't know how to use it yet.
        int $0x80
.endm
```

In tokthr, I used gas assembler macros to define bytecode including relative jumps:

```
2:      .byte b_c_at_inc, b_rot, b_c_at_inc, b_rot
        .byte b_sub, b_dup # - dup if
        fif 1f
        .byte b_unrot, b_twodrop, b_unloop, b_exit
1:      .byte b_drop, b_swap
        floop 2b
```

Here `b_sub`, `b_dup`, etc., are bytecodes, and `fif` and `floop` are macros that insert jump bytecodes with relative jump offsets encoded in the format tokthr's bytecode interpreter wants:

```
        .macro fif, target      # if, or end of while loop
        .byte b_branch_on_0, \target - . - 1
        .endm
        .macro floop, target    # do loop
        .byte b__loop, \target - . - 1
        .endm
        .macro felse, target    # else, unconditional jump
        .byte b_branch, \target - . - 1
        .endm
```

Unfortunately, gas assembly macros aren't quite powerful enough to define things like nested conditional control structures. The above code uses the "local" numeric labels `1f` and `2b`, an invention of Knuth's adopted by gas.

The `b_branch_on_0` bytecode is itself defined by a macro, in this case `defasm`:

```
        defasm branch_on_0, "(if)"
        pop %eax
        and %eax, %eax
        jz branch
        inc %esi                # skip 1-byte jump offset
        jmp next
```

`defasm` is defined as follows in terms of a more basic `define_bytecode` macro, which uses the process of adding data to a given section and subsection as a sort of counter variable: each newly defined bytecode

is assigned a unique address, and if the 256-entry table overflows, you get a compile-time error:

```
        .macro define_bytecode name, realname, origin
        .pushsection .data      # save current position, go to data section
        .subsection 1           # and subsection 1, where we put the addrs
        b_\name = (. - token_table) / 2 # define b_foo as the index of this ptr
        .ifeq b_\name - 256
        .error "\name got bytecode 256"
        .endif
        .short \name - \origin # insert offset which will be resolved next
        .popsection             # return to where we were, and
        .pushsection .dictionary
        countedstring "\realname"
        .popsection
\name:                          # define the name
        .endm
        .macro defasm name, realname
        define_bytecode \name, "\realname", machine_code_primitives
        .endm
```

But gas also has a `.set` pseudo-op that gives you variables that can be mutated arbitrarily during the process of assembly.

To generate unique labels, you can declare them as local to a particular macro (in gas with the `.altmacro` pseudo-op, for example), you can generate unique labels from a parameter (as above in `define_bytecode`), you can use local numeric labels, or you can generate unique labels in some other way, such as the following:

```
        .macro countedstring name
        .byte stringlength\@
1:      .ascii "\name"
        ## Here we count the length of the string --- computers
        ## are for counting bytes so people don't have to!
        stringlength\@ = . - 1b
        .endm
```

Here `\@` is a magic counter that gas increments for each macro expansion, providing enough randomness for the macros to generate unique symbols.

I also defined a `def` macro which takes an arbitrary number of arguments, in order to define bytecode subroutines that fit on a single line with straight-line control flow:

```
    def neg1, "-1", b_dolit_s8, -1   # ( -- -1 )
    def add, "+", b_umplus, b_drop  # ( a b -- a+b ) drop the carry
    def sub1, "1-", b_neg1, b_add    # ( n -- n-1 )
```

This uses the `vararg` feature of gas's macro system:

```
    .macro def name, realname, bytes:vararg
    defbytes \name, "\realname"
    .byte \bytes
    .byte b_exit
    .endm
```

In many assemblers, even the native machine instructions are defined as macros, rather than being built in to the assembler. Here's an excerpt from a disk image of the RDOS operating system for the Data General Nova (see Nova RDOS (p. 1724)):

```
; COPYRIGHT (C) DATA GENERAL CORPORATION 1977, 1978, 1979, 1980, 1981, 1982
; 1983, 1984
...
;INSTRUCTION DEFINITON FILE
...
;DEFINE MEMORY REFERENCE INSTRUCTIONS THAT DON'T REQUIRE AC'S
.DMR JMP=      000000
.DMR JSR=       004000
.DMR ISZ=       010000
.DMR DSZ=       014000

;DEFINE MEMORY REFERENCE INSTRUCTIONS THAT REQUIRE AC'S
.DMRA LDA=   020000
.DMRA STA=      040000

;DEFINE THE ALC INSTRUCTIONS
.DALC COM=      100000
.DALC NEG=      100400
.DALC MOV=      101000
```

However, I think `.DMR`, `.DMRA`, and `.DALC` are in fact built into the DG assembler.

Nowadays NASM is a more popular assembler than gas, in large part because its macro system is more capable; NASM's macro system , though it's still ad-hoc, has not only conditionals and macro-local labels but also a "context stack" mechanism that enables you to define macros for nesting control structures and the like by providing labels that are local to a "context" rather than a single macro expansion.

Stacks and expressions

A major weakness of assembly languages is that they don't have expressions, except for expressions evaluated at compile-time. In effect, you must name all your temporary variables. This makes assembly language verbosity and bug-prone, though it's not the only factor. Consider these two lines of C from dietlibc's `strtod` function; here `value` is a floating-point number:

```
while ( (unsigned int)(*p - '0') < 10u )
    value = value*10 + (*p++ - '0');
```

These get compiled to the following assembly and machine code, somewhat abbreviated and commented:

```
 74 0046 EB0C       jmp .L7
 76                .L8:
 78 0048 D80D0000   fmuls .LC2       ; multiply value by floating-point 10
 78      0000
 80 004e 47         incl %edi        ; p++
 82 004f 51         pushl %ecx       ; copy binary number to memory address
```

```
 83 0050 DA0424     fiaddl (%esp)    ; add in-memory binary number to float
 84 0053 5B         popl %ebx        ; fix the stack
 86                .L7:
 88 0054 8A07       movb (%edi),%al  ; *p
 89 0056 0FBEC8     movsbl %al,%ecx  ; sign-extend into %ecx
 90 0059 83E930     subl $48,%ecx    ; - '0' to convert from ASCII to bin
 91 005c 83F909     cmpl $9,%ecx     ; < 10?  i.e., <= 9?
 92 005f 76E7       jbe .L8          ; if so then loop!
255                .section .rodata.cst4,"aM",@progbits,4
256                .align 4
257                .LC2:
258 0000 00002041   .long 1092616192 ; floating-point 10, poorly represented
```

The particular thing I want to call attention to here is that the C code refers to only two variables, `p` and `value`, which are assigned to the assembly-level registers %edi and st(0), which is the top of the 8087 register stack that `fmuls` and `fiaddl` implicitly act on. The assembly code, however, additionally refers to "variables" %ecx, %esp, %ebx, %al, and .LC2, although .LC2 is just the literal constant 10. Of these, %al, %ecx, and (%esp) are used to hold temporary results that in the C code are left nameless, while %esp is used as a pointer to (%esp), and %ebx is used as a bit bucket.

You could write the code in a pretty closely analogous way in C, though I've taken the liberty of reordering the basic blocks so that we have an exit from the middle of the loop instead of an entry into it, and there's no C equivalent for squirrelling a value away in memory so the 8087 can see it:

```
for (;;) {
    char a = *p;
    unsigned c = a;
    c -= '0';
if (c > 9) break;
    const static float t = 10.0;
    value *= t;
    p++;
    value += c;
}
```

To my eye, at least, this is not an improvement; the indirection of dataflow through explicit variables, which are mutated, makes that dataflow (which is mostly tree-like) much harder to understand. If we call out the one place where the dataflow goes to more than one place by using a variable, it gets better, and I think a bit better even than the C original:

```
unsigned c;
while ((c = (unsigned)*p - '0') < 10) {
    value = value*10 + c;
    p++;
}
```

I think that, in some sense, this is the essence of stack-machine instruction set designs like the GreenArrays F18A core in the GA4 and GA144, and of FORTH: by adding an operand stack to assembly

language, they eliminate the need to assign temporary variables explicitly to intermediate nodes of tree-shaped rootwards dataflow. In FORTH the code precisely analogous to the above looks something like this:

```
variable c
...
begin  p @ c@  [char] 0 -  dup c !  10 u< while
    value f@ 10. d>f f*  c @ 0 d>f f+  value f!
    1 p +!
repeat
```

This is 27 run-time operations ([char] and begin are purely compile-time) which is just over twice the 11 instructions in the i386 code above. Such a 2:1 ratio is typical for stack-machine code and register-machine code.

It happens, though, that there's no need for the separate variable c; we can store it on the operand stack:

```
begin  p @ c@  [char] 0 -  dup  10 u< while
    d>f  value f@ 10. d>f f*  f+  value f!
    1 p +!
repeat drop
```

Or the return stack:

```
begin  p @ c@  [char] 0 -  dup >r  10 u< while
    value f@ 10. d>f f*  r> 0 d>f f+  value f!
    1 p +!
repeat rdrop
```

Nor do we need an explicit fvariable value; as the i386 code does, we can use the top item on the floating-point stack, which ANS FORTH allows to be the operand stack or not:

```
begin  p @ c@  [char] 0 -  dup >r  10 u< while
    10. d>f f*  r> 0 d>f  f+
    1 p +!
repeat rdrop
```

Forth also has compile-time evaluation, which we can use to avoid reconverting 10 to floating-point every time through the loop:

```
begin  p @ c@  [char] 0 -  dup >r  10 u< while
    [ 10. d>f ] fliteral f*  r> 0 d>f  f+
    1 p +!
repeat rdrop
```

That's only 21 operations, just under 2:1. However, perhaps more to the point, it's *dramatically* less code than the assembly version; it's close to the C version in verbosity. (On the other hand, I feel that I have more bugs writing stack code than register code; I succumb too easily to the temptation to keep lots of things on the stack, and then I lose track of where things are.)

The upshot of this is that if you have an operand stack sitting

around somewhere, then it becomes feasible to compose "expressions" by concatenating bits of executable code that communicate implicitly through that operand stack. This more or less gives you Forth.

Topics

Convolution applications

Kragen Javier Sitaker, 2015-09-07 (updated 2019-08-14) (9 minutes)

I think convolution, even the ordinary linear kind, is potentially an underappreciated abstraction for computer systems architecture. A well-chosen abstraction is a sort of "bowtie knot" in the dependency graph: the services it provides can, on one hand, be implemented in many different ways, and on the other, can be used for many different applications. This allows the indiscriminate and highly productive combination of any of *N* different implementations with any of *M* different applications. Successful large-scale examples include the Internet Protocol, the C programming language, the SQL query language, the i386 instruction set, the POSIX API, and more narrowly the filesystem interface.

This note focuses on underappreciated *applications* of convolution, but there are also underappreciated *implementations* of convolution. It doesn't touch on generalizations of convolution such as morphological operations or convolutional neural networks.

(See also the convolution section of More thoughts on powerful primitives for simplified computer systems architecture (p. 1895), the notes about mathematical morphology in Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216), and notes on efficient algorithms in Real-time bokeh algorithms, and other convolution tricks (p. 2661) and file `mcgraw-convolution`.)

Music

Convolution is of course widely used in computer music for things like reverb and frequency selection, but even broader applications are imaginable, especially once we consider time–frequency representations.

Consider synthesizing a xylophone melody. The score is a time-frequency representation of the music: at certain moments, there begins a note of a certain pitch with a certain amplitude. This is a scalar function of two independent variables, time and pitch. An instrument patch representing the xylophone can also be thought of as a scalar function of the same two independent variables, with the time in this case being the time since the beginning of the note and the independent variable being the PCM sample — say, the instantaneous air pressure. And if we reverse the instrument patch in time and frequency, its convolution with the score produces a new function from time and pitch to PCM samples. A timewise slice through that result at pitch=0 gives us the synthesized melody; timewise slices at other pitches, if we use equal temperament and the typical logarithmic scale measured in semitones, give us the melody synthesized in other keys.

There are some other dimensions to consider. If the notes in the "score" are not perfect impulses, but rather have some variation in their frequency content, they will simulate striking the xylophone keys differently; for example, impulses bandlimited to low frequencies will simulate striking them with a softer hammer; using a sharpening kernel instead of an impulse will drop out the lower

frequencies from that note. Of course, you can consider convolving the one-dimensional output slice with a reverberation, but this convolution can also be applied to individual notes in the "score" to add reverberation or softness (or especial sharpness) to particular notes.

But suppose we add a third dimension to the score, such as the note's position in its measure. Then we can convolve the score with a function that applies such color systematically to notes in particular positions in their measures before summing along the measure axis.

Consider also the problem of synthesizing a string sound, such as a violin. Karplus-Strong string synthesis traditionally works by feeding white noise into a digital delay line, which is an all-pole IIR filter with high-Q resonances, which amounts to convolving the (potentially very long) impulse response of the delay line with the input noise. If the noise is continuous and low-amplitude, this sounds a lot like a violin; the process of stimulating the violin string with the random roughness on the bow is somewhat similar, although there's some nonlinear amplification involved. But of course, modulo nonlinear effects like dispersion (e.g. from piano string stiffness), this is just convolving the input noise with the impulse response of the IIR filter.

If you take a "mother wavelet" that is a windowed sample of a continuous sound and convolve it with white noise in this way, the result will have the frequency content of the original continuous sound, frequency-broadened by the time restriction of the window, multiplied by white noise, which has little effect. This should allow you to provide arbitrary amplitude envelopes to an arbitrary continuous sound, within the limits set by the bandwidth-responsiveness tradeoff. Variations of this effect are widely used in music under the name of "vocoders", though usually they don't literally use a windowed sample of a continuous sound. (If your window is about 64 ms, your frequencies will be uncertain by around 15 Hz, and the sharpness of time-domain variation in the "carrier" sound will be diminished by about 64 ms.)

Graphics

Convolution is also widely used in graphics, but largely to apply visual effects like blurring and sharpening.

Text rendering

Consider rendering text on a screen. You can treat this as the convolution of a "window codepoints signal" and a "font signal", each three-dimensional. The "window codepoints signal" has an impulse at (x, y, c) precisely when the character with codepoint c should appear at (x, y); the "font signal" has the color at $(x, y, -c)$ of pixel (x, y) for the glyph of codepoint c. Convolving these two signals produces a three-dimensional signal whose $(x, y, 0)$ plane is the desired text display.

If the signal representing the original font is instead two-dimensional, you can merely space the glyphs farther apart than the width of the window; the convolution output is then a very large two-dimensional surface containing the text of the window and also a large number of scrambled copies of the text of the window. The original window codepoints signal is then a very sparse signal that is

almost equally large, with an impulse at each coordinate where it is desired to select a character.

Putting the glyph index instead into a third dimension avoids the need for the font encoding to depend on a maximum window size and instead produces the desired window as one spacewise slice across a glyph-index axis, with scrambled versions of the window in other spacewise slices.

You could imagine adding a fourth dimension of font style, which could be continuous.

This convolution approach to font glyph rendering can handle overstrikes, hinted glyph rendering, accents, blurring, and even coloring, but it falls down on rotation, multiple font sizes (including zooming), and even, surprisingly, text drop shadows. The drop-shadow problem is that where the solid letter is present, you would like it to opaquely obscure the shadow, not linearly add to it. You need some kind of opacity.

So suppose that, instead of directly producing the two-dimensional image, the convolution generates a three-dimensional opacity field, and we use perspective volume rendering techniques to render it into a 2-D image. Drop shadows can be physically *behind* the glyphs, and larger letters can simply be closer to the camera (with their X and Y coordinates scaled accordingly, of course).

With the addition of this final nonlinear stage of opacity and perspective, convolution becomes much more powerful: not only can it produce drop shadows and multiple font sizes, but it can also take over much of the work of rendering the glyph outlines. You can convolve stroke fonts such as Hershey fonts with a sphere or circle, for example, or more interestingly, with an ellipse at the traditional calligraphic 45° angle, or even a spiky ball to produce a shape with surface striations following the pen. You can represent a serif-font letterform as a sparse three-dimensional signal in (x, y, feature) space, then convolve it with a library of features such as serifs, stems, and bowls in order to generate overlapping three-dimensional forms that comprise the glyph; and you can modify this library to modify the font. Cursive pen-strokes might be signals in an (x, y, pressure) space, allowing you to vary the stroke width, opacity, or ink color by convolving with the appropriate pen. (And of course you can convolve with a small sphere or horizontal ellipsoid to thicken the outlines of outline fonts.)

3-D modeling

There's an existing body of knowledge in rendering convolutional surfaces, that is, implicit surfaces defined by level sets of the convolution of some fixed "kernel function" with a skeleton of a 3-D model. This involves an additional level of nonlinearity between the convolutional operation and the image you see: rather than smoothly varying between opaque and transparent, with the nonlinearity imposed by the tendency of nearby objects to nonlinearly obscure faraway ones.

This can perhaps be applied to the text-rendering problem.

Topics

The problem is not that people are not turning to real journalism anymore

Kragen Javier Sitaker, 2016-11-15 (8 minutes)

A dear, admirable friend of mine recently wrote:

> Well, [I agree that Facebook doesn't want to fix their fake news problem]. But the real problem [with the virality fake news] is people not turning to real journalism anymore.

There are several assumptions that I think underlie this statement, all of which are incorrect:

- Real journalism exists, or used to exist, as a thing that provides reliable information.
- Reliable information can make a person adequately informed about the world.
- Humans who are adequately informed about the world will make good political decisions.
- Good political decisions by voters will result in good government.

I may, of course, be misunderstanding his point of view; but, absent a better alternative for now, I will proceed to rebut my interpretation above.

If any one of these four is false, then the link from "people not turning to real journalism anymore" to good government is a false trail. Here I explain why all four of them are false, but if you find my arguments on one of these four points unconvincing, please have the patience to consider whether at least one of them receives its quietus here.

Journalism has never provided reliable information

Journalism, since its 17th-century inception, is done by people who are ignorant about the topic they are writing about; they learn enough about the topic to be able to explain it authoritatively to others, but their explanations are nearly always ridiculous to anyone knowledgeable about the topic. Worse, journalists are only allowed to explain things that are timely — which is the time when the least is known about them.

This has resulted in journalists clothing all manner of ridiculous lies in the solemn garments of Received Wisdom, with the lightest dusting of background information generally cribbed from some other recent article on the subject.

(Maybe you remember the period when dozens of articles mentioning the World Wide Web described it as "the graphical section of the Internet", when of course the internet does not and did not possess sections; the World Wide Web was not and is not graphical but primarily textual; and many graphical applications that are not Web browsers also ran on and run on the internet. The only way such nonsense could propagate is if each ignorant journalist

copied it from an earlier one.)

Journalism's standard of exceptionality as a factor of newsworthiness also undercuts its reliability, as is widely observed; famously, it has led many people to imagine that commercial airline flight in the US in the 1970s and later was much more dangerous than driving, because plane crashes are exceptional, while car crashes are not.

This makes journalism, even when it achieves its highest standards, as in Consumer Reports, extremely poor as a source for information. But journalism has never frequently achieved its highest standards. Most journalism has always been schlock. The Pulitzer Prize is named after a yellow journalist noted for his sensationalism and unconcern for veracity.

Furthermore, even when reliable information is available, reading it has never been very popular; novels have always been much more popular. This seriously limits the possible readership of a would-be provider of reliable information — the kind of people who think a 5,000-word article is a "long read" are never going to obtain much information about anything outside their direct personal experience.

This has gotten worse in recent years as newsrooms have cut their budgets, but it was always thus: in the 17th century when journalism began, when Hearst arguably launched the Spanish-American War, when reporters interviewed me in my childhood, when Crichton formulated his famous Gell-Mann Amnesia effect in 2002, when the New York Times shrank from calling torture "torture" for a decade, and today.

Reliable information is not enough to adequately inform you about the world

But suppose that the information you get is reliably correct. Instead of reading conceited ignorants parroting the misunderstood words of whatever expert they were able to talk to, you can instead read the brilliantly expressed, deeply knowledgeable analysis of the world's leading experts on a topic, which furthermore is carefully contextualized and prioritized so that you aren't left with big holes in your knowledge. This would be like reading a good textbook on a subject, instead of the low bar journalism aspires to reach and almost invariably fails at, because of the structural problems I described above.

And let's suppose, improbably for nearly all humans, that you actually do read the whole thing. Is this enough to make you informed?

No. If you read a textbook and do not do the exercises, you still do not gain the understanding that the textbook author attempted to convey to you. You cannot become informed about a topic merely by reading about it. You must practice it.

Adequately informed humans still often make terrible decisions, even in politics

The catalogue of human folly is limitless. Many humans smoke several cigarettes per day, thus making a decision several times a day that they know will likely cause them to die in agony while those

they love most watch helplessly. We have all seen humans neglect important parts of their lives until they fell apart — a marriage, a diabetic condition, a leaking roof, their own illiteracy. We all waste our money sometimes, from the poorest to the richest. Procrastination is a universal experience, though some are harder hit than others.

Humans do not decide their actions intellectually. They decide emotionally. Their thoughts affect their emotions, and vice versa. But their thought and beliefs do not determine their actions. Much of the time they invent justifications for their actions after the fact. Salesmen and other negotiators spend enormous attention on modeling the emotional reactions of their prey, even more than they spend on modeling their prey's incentive structures.

All of this applies to politics as well. Informing humans — even when it is possible to do so — often merely provides them with further justifications for their existing belief systems, which (especially for allistic humans) are determined mostly by their social milieu, not by their own attempts to think based on the information they have access to.

Good political decisions by voters are not adequate for good government

If the candidates are corrupt, the government will be corrupt, regardless of which one the voters choose. If the officials, once in office, are confronted by perverse incentive structures in their institutions, even good officials will produce bad government. And Arrow's Impossibility Theorem places strong limits on how good a voting system can be.

Institutions matter. Voting is not enough. Consultation matters. Expertise matters. Low-quality elites don't become high-quality elites by competing for votes.

None of this means voting and journalism are useless

The popular vote is still the most important safeguard against the kinds of disastrous policies that led India to famine after famine under colonial administration, that led the US to commit genocide against the Native Americans, and that kept a substantial fraction of the US's population in chattel slavery for generations. If Indians, Native Americans, and African Americans, respectively, had had the vote, these things would not have happened, as clearly shown by the end of slavery.

And even the very rotten kind of information that journalism provides at its best can still be valuable when it's about current events, where no better information is available.

But let's not mythologize "real journalism" or create unachievable expectations for it.

Posted at https://gist.github.com/anonymous/0d94910bffca58928ed1a2fe219cb0d4b

Topics

Toward a lightweight, high-performance software prototyping environment

Kragen Javier Sitaker, 2018-12-10 (15 minutes)

I was thinking that it might be worthwhile to prototype a client project, but I don't have a reasonable environment to do it in. The easiest way is to use a high-level language to write the high-level app logic, while using existing libraries for storage, cryptography, and networking; the necessary user interface can be provided portably by an embedded HTTP server.

I'm going to talk a lot about performance in here, along three axes: speed, memory usage, and package size. If you can do something ten times as fast, often that means you can do ten times as much of it, and in many contexts if you can do it in a tenth of the memory, you can also do ten times as much of it. A lot of computing resources are available in the form of fungible megabyte-seconds. Package size is probably less important, except as a proxy for implementation complexity, but there are cases where it matters because of either limited storage or limited network bandwidth. For example, if you're embedding an interpreter compiled with Emscripten or wasm in a web page, it's okay if it's 1 MB but probably not if it's 100 MB.

It turns out that software has advanced significantly, and there are several pieces of software out there that offer one to three orders of magnitude performance improvements over commonly-used alternatives.

High-level language: Lua

Lua is a safe, simple, easy-to-embed scripting language, with semantics similar to JS — but its whole grammar fits on one page of the 79-page reference manual. Its interpreter performance is better than that of other languages at a similar level, such as Python, Perl, Tcl, and even most implementations of Scheme, and there's a tracing JIT implementation called LuaJIT whose performance exceeds even that of best-in-class JS JIT implementations like V8 and even, on some scientific benchmarks, C.

More specifically, Lua is a dynamically-typed Algol-like lexically-scoped statement-oriented imperative language with a mark-and-sweep GC (incremental since version 5.1), closures, dynamically-growing hash tables, eval (but no apply), tail-call elimination, a reified global environment, exception handling (with stack traces by default), a metaobject protocol, dynamic method dispatch, reflection, lightweight cooperative threads, a generic iterator protocol, finalizers, weak references, operator overloading, multiple-value returns, multiline strings, goto, variadic functions, and an immutable 8-bit-clean encoding-agnostic string type.

Lua **does not have** classes, ML-style pattern-matching, inheritance (though you can implement it), integers (until 5.3), complex numbers, first-class continuations, built-in serialization, first-class tuples, function overloading, named or default arguments (though there is

syntactic sugar that comes close), preemptive threading, Prolog-style backtracking, lazy evaluation, sequence slicing, unwind-protect, UCS Unicode strings, macros, or much of a standard library.

Like PHP, Lua uses the same mutable type for sequences and finite maps, and a single value can have properties of both. Despite guaranteeing tail-call elimination, Lua's closure syntax is heavyweight enough to preclude using it to define custom control structures as in Smalltalk or Ruby, and it has no macro facility.

The standard Lua 5.1 interpreter is only 171 KiB on my machine, and LuaJIT 2.0.4 is only 443 KiB. The documentation says, "The virtual machine (VM) is API- and ABI-compatible to the standard Lua [5.1] interpreter and can be deployed as a drop-in replacement." LuaJIT is, unfortunately, orphaned, as is Lua 5.1, but its FFI is to die for (it includes a runtime parser for C) and its performance is seriously impressive. The current version of Lua is 5.3; its stock interpreter is 215KiB.

I'm not quite as comfortable in Lua as I am in Python, and I find that Lua is a bit more bug-prone and a bit more verbose. However, modern Python is now also extremely bug-prone due to serious mistakes in how Unicode support was added, and Python is becoming quite unwieldy; /usr/lib/python3.5 on my laptop contains 183,000 SLOC of Python code, and the interpreter itself is another roughly half-million lines of C, half of which is in extension modules. This is roughly 30× the size of the Lua 5.3 codebase. A Python installation is 100 MB; a Lua installation is 171 KiB, or 443 KiB if you use LuaJIT.

Lua is particularly appealing for high-concurrency applications like network servers because it supports "coroutines", which are really cooperative threads rather than coroutines; this is similar to Python's "generator" construct used in the asyncio library, though it differs in some significant details. Even more closely, it resembles the "greenlet" construct used in the now-orphaned Stackless Python EVE Online is written in. Coroutines allow programming of network protocols in a much more structured fashion than that permitted by promises in JS.

Lua is somewhat easier to extend with modules in C than Python or even Tcl, although its style is not to everyone's liking.

A freshly started Lua 5.1 virtual machine on my laptop has a resident set size of 2.7 MB. In modern terms this is exceedingly lightweight, some 300 times smaller than a browser tab with Slack open, but it's still large enough that this environment is not going to be usable for deeply-embedded processing (though NodeMCU provides a Lua 5.1 environment on an ESP8266, which has 96KiB of RAM — as of September 2018 it supports XIP for Lua code, so you can have 256KiB of Lua code and constants). LuaJIT is even smaller, at 908 KiB resident set size.

Software embedding Lua includes Grim Fandango, Escape from Monkey Island, Vim, awesome, Elinks, VLC, World of Warcraft, nmap, Wireshark, haproxy, Haka, sigrok, MediaWiki, LuaTeX, the Battle for Wesnoth, LÖVE2D, OpenResty, and Adobe Lightroom.

Bug-proneness

Above I said Lua was pretty bug-prone; I will elaborate on that here, because I think it's the main disadvantage of Lua, though one that's worth accepting in order to get the rather awesome features

described above. Eventually this bug-proneness seems likely to limit the fraction of your code that's worth writing in Lua.

Of course, Lua is dynamically typed, which isn't really a problem in itself, but does slightly exacerbate the other problems.

In several cases, it attempts to DWIM in ways that can cover up bugs; Lua does not believe that "errors should never pass silently", as the Zen of Python says. Specifically:

• Reads of nonexistent variables and table entries simply returns `nil` rather than raising an error;
• I/O errors do not raise errors by default when using the more general I/O library;
• worse, writing to nonexistent variables creates new global (!) variables;
• function argument list and return value adjustment similarly introduces nils, and also silently discards extra values.
• In Lua 5.3, which adds integers, implicit numeric coercion (int to float, and vice versa) is the rule, and integer math can produce different results from floating-point math;
• concatenating numbers to strings implicitly converts them to strings; and
• as in JS, writing to nonexistent sequence indices extends the length of the sequence.

In a few cases, the special nature of `nil` can create bugs analogous to SQL injection and blueboxing — `a[b] = c` will delete the table entry `a[b]` if `c` is unexpectedly nil; worse, if `b` was a number, that may unexpectedly change the length of the sequence `a`. Similarly, unintentionally returning a `nil` value will terminate an iterator early.

Lua's choice of indices for sequences — 1, 2, … n rather than the now-conventional 0, 1, … n-1 — is slightly more bug-prone, for precisely the reasons described by Dijkstra.

As in multitasking Forth systems, but unlike Python generators (or for that matter JS promise callbacks), any function invoked by a Lua coroutine has the possibility of yielding control. But because coroutines are resumed explicitly, rather than using an implicit global run queue, there is no locking mechanism that could block potentially interfering concurrent executions. Ierusalimschy claims this makes the coroutine mechanism "more powerful", which is certainly true, and precisely the problem. It's precisely analogous to unchecked exceptions, aspect-oriented programming, or dynamic method dispatch: by allowing a local change to have an effect that would have otherwise required a global change, this power means that to determine a certain property of the program that would have been local, instead a global search is needed.

Storage: LevelDB

LevelDB is a high-performance persistent bytestring key-value store by Jeff Dean and Sanjay Ghemawat, supporting ordered traversal and a limited form of transactions; on my laptop, it can handle about 300,000 key-value-pair insertions per second, about 10 to 100 times faster than Postgres and 2 to 20 times faster than SQLite. Unlike Berkeley DB, LevelDB remains fast when inserting many widely scattered keys into a large existing data store, even on high-capacity spinning-rust disks rather than lower-capacity SSDs,

using a data structure sometimes known as the "LSM-tree" or "log-structured merge tree".

The library itself is 359 KiB, but it depends on libsnappy, the high-speed compression library previously known as Zippy, which is another 30 KiB.

Other popular alternatives for this kind of thing include Berkeley DB, Redis, MongoDB, SQLite, or using some kind of serialization library (such as a JSON implementation, FlatBuffers, Protocol Buffers, or Thrift) to generate bytes that your code then manually writes to a file; and then there's RocksDB, which is a fork of LevelDB. Most of these are very large, very featureful, and very slow.

Redis and MongoDB involve running separate processes, and their authors are playing dishonest games to confuse people about free software.

Berkeley DB is 1.7 MiB and many times slower at bulk insertions; also, it's controlled by Oracle.

SQLite is 922 KiB and many times slower at everything except inserting large blobs and reading.

RocksDB was written as a fork of LevelDB with improved performance, but it's 3.1 MiB.

LevelDB is used by the official Ethereum client, formerly the official Bitcoin client, the high-performance distributed filesystem Ceph, Chrome, PouchDB, and Riak; Parse was built on RocksDB.

Cryptography and networking: libsodium

Libsodium is a better-packaged version of the highly-regarded NaCl networking and cryptography library, with some extra functionality added. Unlike other popular libraries such as OpenSSL, libsodium doesn't expose a wide variety of cryptographic primitives; instead, it provides a small number of functions that are easy to use securely, based on a small and conservatively chosen set of cryptographic primitives, including Salsa20, AES-256-GCM, SHA-256, SHA-512, ChaCha20, Poly1305, and Ed25519. In many cases, it includes the fastest available implementations of these primitives for many platforms.

The only plausible alternatives here are NaCl itself and monocypher, a fork of libsodium.

Compression: Snappy and zlib

LevelDB optionally compresses the data it writes using Snappy, since Snappy compression and especially decompression is significantly faster than spinning-rust disks (250MB/s per core for compression, twice that for decompression). Since the platform embeds LevelDB, it necessarily includes Snappy, so we might as well expose it at the Lua level.

However, zlib — the universally-used implementation of LZ77 compression — compresses sufficiently better than Snappy that it's worth including it as well. In particular, for compressing library code which is loaded at startup, zlib is a big win; it also permits implementing compressed HTTP and accessing zipfiles. zlib is only about 100 KiB, and the lua-zlib binding is 9 KiB.

As a quick test, I compressed a Lua source file I wrote a few years ago with Snappy and zlib. It compressed to 0.50 times its original size

with Snappy and 0.34 times its original size with zlib.

HTTP server

You can hack together an adequate HTTP/1.0 server in about 300 machine instructions on top of Linux sockets, or a similar or smaller number of lines of code in higher-level languages. (Often the worse performance of higher-level languages requires a bit more complexity to compensate, but even the fairly rich implementation in Python's `BaseHTTPServer`, `SimpleHTTPServer`, `urllib`, `urlparse`, and `cgi` modules only works out to about 2600 lines of code.)

There exists a fairly full-featured webserver in Lua called Xavante (142K, plus dependencies on coxpcall (46K), copas (99K), and luafilesystem (82K), for a total of 369K).

Embedding an HTTP server is by far the easiest way to provide a modern user interface, even on the local machine.

Miscellaneous libraries

The Lua standard library contains very little; even sockets are provided by the external "luasocket" package (563K, including implementations of HTTP, SMTP, and FTP), and although the built-in filesystem interface allows you to read and write files, it doesn't support directory creation or listing; until Lua 5.3, the language doesn't natively include an integer type or bitwise operations. The "luaposix" library is a smaller alternative (204K, plus bitop, a 75K dependency) to the luasocket and luafileystem libraries, providing the full POSIX API.

Total weight

The total virtual machine should be 0.45 MB of LuaJIT + 0.37 MB of LevelDB + 0.38 MB of libsodium + 0.03 MB of Snappy + 0.10 MB of zlib + 0.10 MB of other library code (mostly Lua), for a total of 1.43 MB, a floppy disk's worth.

However, *those are the uncompressed sizes*. The zlib-compressed sizes of the various pieces are as follows:

	KiB	KiB (gz)
LuaJIT	443	227
/usr/lib/x86_64-linux-gnu/libleveldb.so.1.18	359	151
/usr/lib/x86_64-linux-gnu/libsodium.so.18.0.1	376	165
/usr/lib/x86_64-linux-gnu/libsnappy.so.1.3.0	30	13
/lib/x86_64-linux-gnu/libz.so.1.2.8	102	55
misc		100

If we figure we need an uncompressed zlib to bootstrap uncompressing the rest of the platform, then the total is 758 KiB.

Other candidates for inclusion

I'd really like to include support for high-performance numerical computation, machine learning, windowing user interfaces, GPGPU, audio, ØMQ or similar, and FlatBuffers (or Cap'n Proto or SBE). Torch 7 has numerical array support for Lua; it's billed as "a scientific computing framework [for LuaJIT] with wide support for machine learning algorithms that puts GPUs first," and it also supports non-LuaJIT Lua 5.2; unfortunately it's orphaned in favor of a C++

replacement called “ATen”.

Topics

Differentiable neighborhood regression

Kragen Javier Sitaker, 2019-08-31 (15 minutes)

Neighbor predictors are a family of predictors in machine learning; they work by predicting, from a probe point, some kind of average of the points nearest to it. K-nearest-neighbors, or K-nn, with a constant number of nearest points, uniform weights, and the mode as the kind of average, is the most common, but there are also R-neighborhood algorithms which use all the points within some radius R.

When the variable being predicted is continuous, it's called *regression*, and it's common to use a weighting function or kernel so that points very near the probe point weigh more heavily in the average.

Commonly neighbor regression has discontinuities in their predictions as the probe point moves around, and this represents suboptimal performance, and makes them unusable for some purposes. For example, in Multitouch and accelerometer puppeteering (p. 1785) and $1 recognizer diagrams (p. 1264) I was looking for a way to morph smoothly between "keyframes" placed at arbitrary points in a two-dimensional state space. So I was looking for a way to fix discontinuity in neighbor regression, and one occurred to me. Maybe it's known.

Continuous neighbor regression

So, consider specifically the case where we're using a weighted mean: $\sum_i w(r, x, x(i))\ y(i)$, where $x(i)$ is the position of previously observed point i, x is the position of the newly observed point we're trying to predict, $w(r, a, b)$ is a weighting function which tells you how relevant a point at b is for predicting a point at a with a neighborhood of radius r, and $y(i)$ are the observations at the points $x(i)$ of the variable to be predicted. Commonly we normalize by dividing by $\sum_i w(r, x, x(i))$, particularly when the number of points in the neighborhood may vary, and usually the weighting function is a shift-invariant kernel of the form $w(r, a, b) = v(r, b - a)$.

Now, in this formulation, you could get discontinuities in three ways. First, the radius r could change discontinuously as you move continuously around the space. Second, the function v might have discontinuities, so that either a continuous change in $b - a$ or a continuous change in r might give rise to a discontinuity in v. Third, if a point has nonzero weight when it enters or exits the neighborhood, that will also give rise to a discontinuity, unless another point simultaneously exits or enters in a way that compensates for the discontinuity.

So the simplest way to get neighbor regression to be continuous is to use a constant r and the kernel function

$$v(r, c) = 1 - |c|/r$$

so that when points enter or exit the neighborhood window, they do so with zero weight, thus producing no discontinuity.

You can use a more elaborate kernel function; the only thing that's

important is that it be continuous and reach 0 when $r = |c|$. Usually you want it to be nonnegative, too, and rapid to compute.

Similarly, you can use a varying radius, as long as it varies continuously. The appeal of K-nearest-neighbors (as opposed to R-neighbors) is that it becomes more detailed without becoming less efficient in areas where you have more data; in Knn, the radius *also* varies continuously, as there is never a discontinuity in the distance to the Kth-nearest point.

Differentiable neighbor regression

The above cone kernel gives a continuous regression function, but its derivative has discontinuities when points enter and exit the kernel and when they cross its center.

If, additionally, the radius changes *differentiably* with your position in the space, the kernel function $v(r, c)$ is differentiable as r and c change differentiably; points enter and leave the window with not only zero weight but also zero weight *derivative* (with respect to differentiable movement of the probe point), then the resulting predictor is also differentiable.

If v is purely a function of $|c|$, making it rotationally symmetric, it needs to have zero gradient when $|c| = 0$; otherwise it will fail to be differentiable at that point.

The simplest way to satisfy this is to use a fixed r and a function such as the "zero-phase Hanning window"

$$v(r, c) = 0.5(1 + \cos(\pi r/|c|))$$

which is differentiable and nonnegative, and has zero derivative at $|c| = 0$ and $|c| = r$, value 1 at $c = 0$, and value 0 at $|c| = r$.

However, the Hanning window is a bit heavyweight, requiring as it does a transcendental function. A low-degree polynomial function would be more desirable; for example

$$v(r, c) = 1 - 3d^2 + 2d^3 \text{ (where } d = |c|/r)$$

This can be computed as $(2d - 3)d^2 + 1$, requiring a doubling, a subtraction, a squaring, a multiplication, and an increment. Like the Hanning window, it is differentiable and nonnegative, and has zero derivative at $|c| = 0$ and $|c| = r$, value 1 at $|c| = 0$, and value 0 at $|c| = r$. Visually it is almost impossible to tell the two windows apart; they differ by about 0.01 at maximum. (This function is in fact the unique degree-3 Hermite interpolation of the "Hanning window" at those two endpoints.)

This computational cost estimate is leaving something out, though: the cost of computing $|c|$, which requires a square root!

$$|c| = [\sum_i (x_i(j)) - x_i)^2]^{0.5}$$

So let's consider instead the following much cheaper kernel function, which has the same desirable properties but without requiring any irrational functions:

$$v(r, c) = 1 - 2R + R^2 \text{ (where } R = |c|^2/r)$$

This function, too, looks visually close to the Hanning window (though it differs from it by up to about 0.065), is differentiable and nonnegative, and has zero derivative at $|c| = 0$ and $|c| = r$, value 1 at $c = 0$, and value 0 at $c = r$. But it can be evaluated by just a doubling, a squaring, a subtraction, and an increment, once you have $|c|^2 = \sum_i (x_i(j)) - x_i)^2$. Not only does it avoid the square root, it doesn't even require a multiplier! (Except for the pesky r scale factor, that is.)

An alternative way to avoid square roots is to use a different Minkowski p-norm of c rather than the Euclidean L_2 norm used above, sacrificing pure rotational symmetry but avoiding the squaring operations necessary to calculate $|c|$ or even $|c|^2$. The L_1 norm and the L_∞ norm are both easier to compute, and in 1-D and 2-D they are equally good approximations; in higher dimensionalities the L_∞-norm ball touches the L_2-norm ball in more places than the L_1-norm ball does, which perhaps makes it a better approximation.

You can get a reasonably good approximation of the L_2 norm in a variety of cheaper ways. Taking the maximum of the L_∞ norm multiplied by $\sqrt{d}$ (where d is the number of dimensions) and the L_1 norm gives you a somewhat better approximation, as does the sum of the L_∞ and L_1 norms, scaled appropriately. At least in two dimensions, the sum of these two approximations, again scaled appropriately, is a better approximation still. (All of these approximations are also norms, as is easily verified.)

The level set of the L_∞ norm is a hypercube, while the level set of the L_1 norm is its dual, a "cross-polytope" or "orthoplex" such as an octahedron or 16-cell. The level sets of the other combinations described above are progressively rounder polytopes with their vertices all equidistant from the origin.

Most of the above discussion of efficiency concerns minimizing the bit operations necessary, as if you were designing circuitry. If you're using a modern CPU or GPU, the computation needed to dispatch an instruction absolutely dwarfs the computation needed to multiply two numbers or take a square root, so you should just use whatever takes the fewest instructions.

Differentiably variable neighborhood radii

The above is perfectly fine for a fixed radius, but at times a variable radius might be better, both to avoid deserts with no samples to work from (or one sample, resulting in a flat plateau) and to avoid well-covered regions where the kernel function blurs out almost all of the local detail and additionally demands a lot of work. But if we just use a radius for K nearest neighbors, we inevitably run into points where differentiability fails, as our expanding moving circle bumps into a new Kth point and starts to contract, leaving an old point behind.

So we need some way to compute a differentiably-changing radius that still includes about the same number of points.

One way to handle this is to use a fixed-radius kernel to find nearby points and see how many there are, or more precisely, what their $w(r, a, b)$ adds up to. This is a differentiable quantity, and except at zero, so are its reciprocal and the reciprocal of its square root. If it covers a region where the points are distributed more or less evenly, we can use that reciprocal square root as a kernel radius, and it will tend to cover a more or less consistent number of points.

It might be worthwhile to iterate this some fixed number of times, such as two: use the sample of points captured by kernel i to calculate a smaller radius at which to run kernel $i+1$, to calculate a smaller radius at which to run kernel $i+2$.

Division by 0 is a constant risk here. Perhaps for a given set of points you could calculate a minimum radius needed to always avoid it.

This approach is commonly called a "balloon estimator".

Interpolation versus regression

In machine learning it is typically assumed that the observed values are subject to some noise, so predicting them all precisely is symptomatic of overfitting and will lead to poor future results. But for some of the applications I have in mind, like animation keyframes in an abstract character state space, we really would like to exactly hit the given data point — we want to precisely interpolate a spline (in the sense of Levien, not de Boor) rather than fitting a curve to noisy data.

All the variants of neighbor regression described above, on a fixed set of probe locations and data points, is a sparse linear transformation of the data points it's regressing to — the weight values don't depend on the $y(i)$, just the $x(i)$. Moreover, I think it tends to be a reasonably-well-conditioned one, at least if your neighborhood widths are reasonable. So determining the set of ersatz observations $y(i)$ we would need for it to precisely predict the *actual* given observations $b(i)$ is simply a matter of solving the sparse linear system $\mathbf{A}y = b$, where the rows of the square matrix $\mathbf{A}$ are computed with w on the observation points $x(i)$. Interpolating between y and b allows you to choose the degree to which the surface has a little bit of freedom to suppress noisy points.

This turns neighbor-regression algorithms into N-dimensional surface-fitting algorithms, and they can be used thus as an alternative to NURBS.

It lacks some desirable properties for such surface-fitting algorithms; in particular, it lacks locality, in that a change to any input point will in general result in changes everywhere on the surface, not just near that point.

It should be mentioned that there's an existing standard way of doing this, used for example in the sklearn documentation, but it sucks. It is to use $r/|c|$ as the weight function, which diverges when $|c| = 0$, but in that case you're at a sample point and you can just use the value of that sample point. It's continuous close to the sample point, but it can have discontinuities when faraway points slip in and out of the window; also, it's far from differentiable, with sharp spikes at all the sample points.

Vector regression or interpolation

Above I've been speaking in terms of observations $y(i)$ and predicted values. It bears mentioning explicitly that these observations are not necessarily scalars such as the height of a surface; they might be, for example, vectors of 256 (x, y) pairs describing the path of a pen stroke, as in the application to $1 recognizer diagrams (p. 1264). All the math above works in the same way, since the only thing we're ever doing with the $y(i)$ is forming linear combinations of some of them weighted by the normalized weights from w — except for the linear-interpolation linear system to be solved in the interpolation-versus-regression section above, which should be solved separately for each component of the observation vector. For example, solve one $\mathbf{A}y = b$ problem for the x coordinate of the first point in each pen stroke, then a second $\mathbf{A}y = b$ for the y coordinate of the first point, then a third $\mathbf{A}y = b$ for the x coordinate of the second point, and so on.

A note on terminology

My terminology in the above has been fairly inconsistent, both with itself and with established terminology. Knn is normally called *k* -NN; the sample points are often called "training examples" rather than "samples" and the space they are located in (the independent variables) is called the "feature space" and sometimes the "search space" with its dimensions being "features"; the regression result is a "prediction" of a "property value" for the "object" or "query" or "query example" or "test point" (what I sometimes called the probe point above). I should probably go back and fix it.

Topics

- Programming (p. 3658) (286 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Linear algebra (p. 3551) (4 notes)
- Approximation (p. 3321) (2 notes)
- Norms

Nonlinear differential amplification

Kragen Javier Sitaker, 2016-12-14 (2 minutes)

Transistors, especially FETs, are lovely little devices. They have very high energy gain — effectively infinite in the FET case. They respond very quickly, and aside from that finite response speed, their response is historyless, which makes the electronic design and test problems considerably more tractable. In particular, circuits that are linear, noninverting, or non-amplifying are not capable of universal computation, and the transistor immediately takes care of all three of those barriers at once. (I'm fairly sure it's possible to imagine nonlinear, inverting, amplifying circuit elements that are still incapable of universal computation, but I don't know what they'd look like.)

Transistors do have a few problems, though. One is that their response speed, while good, could still be better; it's in the picoseconds range for sizes that are practical to fabricate. Another is that the transistors that we know how to fabricate are still somewhat tricky to fabricate, requiring exotic materials, materials not found in nature and of very high purity.

Braess's Paradox gives us a way to convert some kinds of noninverting nonlinearities into inversion; the well-known construction with resistors and a Zener diode is an example. In general, the bridge topology (used in the Braess circuit) provides a way to materialize a *difference* in your circuit, which gives you a way to get inversion and amplification, in some sense anyway. A change of 1% in the voltage across one of the "shores" of the bridge can easily result in a change of 10,000% in the voltage across the bridging element.

Nonlinear circuit behavior and behavior with memory is actually fairly universal in real circuit elements; it's just that normally we consider it a problem to solve, because especially in analog circuits, the distortions introduced by nonlinearity are difficult to characterize and predict. But, for example, capacitor databooks are full of information about the capacitors' deviations from linearity, which often reach 20% or more despite their best engineering efforts to remove them. What if we could take advantage of such everyday nonlinearities to implement digital logic?

Topics

- Electronics (p. 3430) (138 notes)
- Physical computation (p. 3631) (26 notes)
- Braess

Bootstrapping rope bridges and other tensile structures with UHMWPE-bearing drones

Kragen Javier Sitaker, 2019-11-25 (5 minutes)

To build a suspension bridge across a river, you hang it from a heavy cable. To pull the heavy cable across, you use a rope. To pull the rope across, you use a thin cord. To pull the thin cord across, you use a fine thread. But how do you pull the fine thread across? Well, according to stories, you shoot an arrow. But nowadays maybe a small quadcopter would be better.

How fast can you do this kind of bootstrapping? To simplify the problem, consider the problem of running a heavy cable 100 meters straight up, looping it over something smoothish with no significant friction, and running it back down. Instead of using discrete sizes, let's suppose the cable tapers exponentially, and that it is made of UHMWPE with 2.4 GPa yield stress (see Dyneema (p. 123)) and 0.97 g/cc, and that our initial flight can lift 10 g.

But let's save the initial flight for later.

Once we're in steady state of cable embiggening, the weight of the 100 m of up-going hanging cable needs to be supportable by the thinner cable looped over the top, and (more demandingly) the *difference* in the weight of the up-going cable and the down-going cable needs to be supportable by the still thinner cable we're reeling in at ground level.

Before solving the problem exactly, let's consider a crude approximation: the cable reeling in at the ground needs to be able to support the 100 m of up-going cable, which is all the thickness of the cable leaving the ground, 200 m in the future. So, over 200 m, the cable can only get thicker by a factor such that the cable 200 m ago is still thick enough to support 100 m of it.

100 m of Dyneema weighs 97 g/cm², which is 97 tonnes/m², or 950 kPa. This is 2500 times less than the yield strength, so we're safe as long as the cross-sectional-area growth over 200 m is a factor of 2500 or less, or a factor of 50 or less in diameter. So, by pulling on a 100-micron-thick fiber, you can lift a 5-mm-thick cord on the other side, and by pulling on the 5-mm-thick cord when it gets to you, you can lift a 250-mm-thick cable on the other side.

But that's actually ridiculously conservative, because at cable position t you're not actually lifting 100m $f(t - 200\text{m})$, where $f(s)$ is the linear density of the cable at point s, but rather $\int_a^b f(u)\,du - \int_b^t f(u)\,du$, where $a = t - 200\text{m}$ and $b = t - 100\text{m}$, and the second integral is insignificantly small. Solving this exactly is pretty easy.

Even using the simple approximation, though, 10 g over 100 m at 0.97 g/cc is 0.103 square millimeters, so the initial thread you send up to loop over the top can be 0.103 square mm (320 microns in diameter) at the bottom of the tower. That, too, is ridiculously conservative if you taper it.

In practice it is probably difficult and inconvenient to handle fibers of less than 10 microns in diameter; even though UHMWPE is

highly biocompatible, inhaling tiny rigid fiber fragments may cause mechanical damage. (By comparison, spider silk fibers are 2.5 to 4 microns in diameter.) They're also likely to run into significant trouble with wind, especially given UHMWPE's brittle nature. 10-micron UHMWPE fibers can, if we extrapolate the mass yield stress downwards, withstand 0.19 N of force, the weight of 19 grams. 10-micron-diameter UHMWPE weighs 76 micrograms per meter, which is 76 mg per km.

Sending such tapered ultrastrong fibers around on microminiaturized drones or balloons may be a quick way to get hold of things that are far away, build enormous tensile structures between buildings and mountains, and snarl up rotating machinery, such as helicopter blades, although for such impact applications it might be useful to knit or zigzag the fibers in some way to reduce the acceleration shock, for example weaving them into a yielding net which, when hit, draws up heavier fibers from the ground.

Topics

• Physics (p. 3632) (119 notes)
• Materials (p. 3560) (112 notes)
• UHMWPE (p. 3762) (11 notes)

Parallel register file

Kragen Javier Sitaker, 2018-11-27 (2 minutes)

A thought that occurred to me as I read about the LMI K-machine 's duplicated register file:

Suppose you are executing a 32-bit 3-address instruction with three 8-bit register fields. As a possible alternative to having 256 registers (or a smaller register field), you could have 8 registers, one bit in each register field identifying some subset of the registers to use. For example, you could specify 00001000_2 to indicate register 3, or 01000000_2 to indicate register 6. In the case where you specify more than one register to write to, the results are written to all specified registers; 11111111_2, for example, writes to all 8 registers, and 00000000_2 discards the result.

A perhaps more reasonable design here is to have an output register field that is twice the length of the input register fields, each of which can address only one half of the register space, thus eliminating the necessity for a multi-ported register file. For example, you could have 6 bits per input register field and 12 bits of output register field. If the two halves of the output register field were always identical, it would look like a machine with 6 registers and the usual dual-ported register file, just with an inefficient instruction encoding.

Presumably the results when you select more than one input register should be specified; for example, wired-AND or wired-OR would be reasonable. Wired-OR has the advantage that 000000_2 produces 0, which is more commonly useful than the -1 that would be produced in the wired-AND case. This also, of course, eliminates any necessity for an OR instruction.

12 registers in this form would probably be about as comfortable as 8 registers in the more usual form.

Topics

Distinguishing natural languages with 3-grams of characters

Kragen Javier Sitaker, 2013-05-17 (updated 2013-05-20) (7 minutes)

In outgoing-notes-mail-01.

I thought a fun tiny program would be something to identify languages from training data. Some few N-grams are highly distinctive of particular languages; it should be possible to use a table of a few such N-grams to distinguish. Stuffing an entire byte 3-gram into a machine register, a simple C program can tabulate about five megabytes of 3-grams per second on my netbook:

```
#include <stdio.h>
#include <ctype.h>

enum { n_threegrams = 256*256*256 };
int threegrams[n_threegrams];

static inline int
update(int threegram, char c)
{
  unsigned char uc = c;
  threegram <<= 8;
  threegram |= uc;
  threegram &= 0xffFFff;
  return threegram;
}

int main(int argc, char **argv)
{
  int threegram = 0;
  int out_of_word = 1;
  char cc;
  int ii;
  threegram = update(threegram, '-');
  threegram = update(threegram, '-');
  threegram = update(threegram, '-');
  while (fread(&cc, 1, 1, stdin)) {
    if (isalnum(cc) || cc == '\'') {
      out_of_word = 0;
      threegram = update(threegram, cc);
      threegrams[threegram]++;
    } else if (out_of_word) {
      /* nothing */
    } else {
      out_of_word = 1;
      threegram = update(threegram, ' ');
      threegrams[threegram]++;
    }
  }

  for (ii = 0; ii < n_threegrams; ii++) {
```

```
    if (!threegrams[ii]) continue;
    printf("%d %c%c%c\n", threegrams[ii],
           ii >> 16 & 0xff,
           ii >> 8 & 0xff,
           ii >> 0 & 0xff);
  }

  return 0;
}
```

The top 3-grams my program finds for Spanish in my Spanish dictionary file are:

```
4000 dor
4225  de
4336 a c
4609  n
4936 te
5006 nte
5537 ra
5710 ado
6742 ent
8842 ar
```

For English from the KJV bible:

```
22273 d t
24569 to
35631 of
36735  of
43476  an
45407 and
59014 nd
74898 he
96843 the
121874  th
```

In the KJV, 'ar ' was at 3638, some 40 times less common than ' th', while 'ado' was at 181, some 673 times less common; 'ra ' was less common still.

Running it against the 20MB spanishText_10000_15000 from the Spanish WikiCorpus v1.0 yields somewhat different results:

```
109439  co
112939 en
134011 el
136181 es
136930 la
138318 as
151942  la
195466 os
251117 de
319299  de
```

In the KJV results, I got ' de' 4575 times, 'de ' 2851 times, and 'os '

only 45 times. ' th' occurrs only 1354 times in the Spanish WikiCorpus text.

So in Spanish text, 'os ' is 195466/1354 = 144 times as common as ' th', while in English text, ' th' is 121874/45 = 2708 times as common as 'os '.

So it seems reasonable to guess that a text containing more 'os ' than ' th' is Spanish if it's one of Spanish and English, and vice versa; and both are sufficiently common in their respective languages that even a very short sample of one of these languages is likely to contain an instance. 'os ' occurred about once every 100 bytes in Spanish, while ' th' occurred about once every 40 bytes in English.

So you can probably do a reasonable job, on x86, of discriminating between these two languages as follows:

```
enum language { lang_en, lang_es };
enum { sp_th = ' ' | 't' << 8 | 'h' << 16,
       os_sp = 'o' | 's' << 8 | ' ' << 16 };
enum language __attribute__((regparm(2)))
lang_id(char *text, int len) {
  int englishness = 0;
  for (; len; text++, len--) {
    int threegram = *(int*)text & 0xffFFff;
    if (threegram == sp_th) englishness++;
    else if (threegram == os_sp) englishness--;
  }
  return englishness > 0 ? lang_en : lang_es;
}
```

This works well, and compiles (with -Os -fomit-frame-pointer) to 22 instructions, 53 bytes:

```
08048504 <lang_id>:
 8048504:       53                      push   %ebx
 8048505:       31 c9                   xor    %ecx,%ecx
 8048507:       eb 23                   jmp    804852c <lang_id+0x28>
 8048509:       8b 18                   mov    (%eax),%ebx
 804850b:       81 e3 ff ff ff 00       and    $0xffffff,%ebx
 8048511:       81 fb 20 74 68 00       cmp    $0x687420,%ebx
 8048517:       75 03                   jne    804851c <lang_id+0x18>
 8048519:       41                      inc    %ecx
 804851a:       eb 0e                   jmp    804852a <lang_id+0x26>
 804851c:       81 fb 6f 73 20 00       cmp    $0x20736f,%ebx
 8048522:       0f 94 c3                sete   %bl
 8048525:       0f b6 db                movzbl %bl,%ebx
 8048528:       29 d9                   sub    %ebx,%ecx
 804852a:       40                      inc    %eax
 804852b:       4a                      dec    %edx
 804852c:       85 d2                   test   %edx,%edx
 804852e:       75 d9                   jne    8048509 <lang_id+0x5>
 8048530:       31 c0                   xor    %eax,%eax
 8048532:       85 c9                   test   %ecx,%ecx
 8048534:       0f 9e c0                setle  %al
 8048537:       5b                      pop    %ebx
 8048538:       c3                      ret
```

You could squish this down quite a bit more; the eight-byte `sete;movzbl;sub` sequence is there to avoid a three-byte `jne;dec` sequence, if you swapped the functions of `%edx` and `%ecx`, you could use the two-byte x86 `loop` instruction instead of the five-byte `dec;test;jne` version; and you can probably skip the handling of the empty string with the unconditional jump to the end of the loop. The untested 45-byte result is:

```
        ## Distinguish English from Spanish in a text buffer.

        .globl langid
langid:
        push   %ebx
        mov    %edx, %ecx
        xor    %edx, %edx
loop:   mov    (%eax), %ebx
        and    $0xffffff, %ebx
        cmp    $(' ' | 't' << 8 | 'h' << 16), %ebx
        jne    test2
        inc    %edx
        jmp    incr
test2:  cmp    $('o' | 's' << 8 | ' ' << 16), %ebx
        jne    incr
        dec    %edx
incr:   inc    %eax
        loop   loop
        xor    %eax, %eax
        test   %edx, %edx
        setle  %al
        pop    %ebx
        ret
```

By factoring out the N-grams into a data structure (`for threegram, idx in features { if threegram == here { counts[idx]++ } }`) , you could probably extend this with another 4 bytes or so per language, up to a dozen or so languages, with reasonably good results, but you'd need to choose the N-grams with reference to all the languages; 'os ' and 'de ', for example, turn out to be common in a number of Romance languages, so you might end up using some other, less common N-gram; and as a result you might have to use more than one N-gram per language.

As an example, here are the top 3-grams from 12 megabytes of Catalan (also from WikiCorpus 1.0):

```
56343 que
58327 en
62273 ent
62702  el
67923  la
76707 el
80647 la
110601 de
126960 es
169507  de
```

Compared to Spanish:

```
109439  co
112939 en
134011 el
136181 es
136930 la
138318 as
151942  la
195466 os
251117 de
319299  de
```

In the Catalan corpus, 'os ' occurs 15674 times, about once every 800 bytes --- one-eighth as common as in Spanish, but common enough that you should probably pick a different 3-gram to distinguish between Spanish and Catalan. ' i ' occurs much more frequently in Catalan than in Spanish (and ' i ' occurs not at all in the KJV) but I'm not quite sure what occurs much more frequently in Spanish than in Catalan.

If you found a reasonable set of 3-grams (or even 2-grams or 4-grams) that distinguished the different languages in your set, you could perhaps search for a machine-code hash function that maps one or two of the desirable 3-grams into each of a few buckets. This might take less space than storing the 3-grams themselves, since you could choose a set of 3-grams that happened to have a compact machine-code representation.

Topics

Options for bootstrapping a compiler from a tiny compiler using Brainfuck

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

What's the simplest instruction set for a compiler that can compile itself? I think you probably need some kind of looping construct, some kind of way to read in bytes, some kind of way to output bytes, and probably sequencing of statements. You can probably avoid much parsing by making the input instructions be single bytes, but you probably still need some kind of arithmetic to calculate the jump offsets.

Mats Linander's Awib is a multi-target optimizing BF compiler written in BF, with six backends (it can output Linux i386 ELF executables, C, Tcl, Golang, Ruby, and Java), so BF is clearly a sufficiently powerful language to write a self-compiling compiler in. (It also contains an ASCII-art portrait of Meriday in the morning and can be compiled as Tcl, bash, or C, as well as BF. Truly impressive.) It's about 43 kilobytes, and it would presumably run under Urban Müller's original 240-byte AmigaOS BF compiler or Brian Raiter's 199-byte Linux version: http://www.muppetlabs.com/~breadbox/software/tiny/bf.asm.txt

Daniel B. Cristofani has written a much more minimal BF self-compiler, targeting C; I found a copy at http://esoteric.sange.fi/brainfuck/impl/compilers/dbf2c.b, and he has a copy on his own web site at http://www.hevanet.com/cristofd/brainfuck/dbf2c.b. It seems to work; anyway, I compiled it with itself, verified that it produced the same output when compiled with the self-compiled version of itself, compiled Linus Åkesson's Game of Life with it, and played Life successfully on the result. Cristofani's self-compiler for BF is 1183 bytes, but running Erik Bosman's bfstrip utility from http://esoteric.sange.fi/brainfuck/utils/bf-tools/bfstrip.c on it reduces it to 904 bytes.

So, one approach to bootstrapping things from BF would be to compile programs from other languages into BF, and then run them with one of these BF interpreters. But I think there's a much more interesting approach available, which is to add some instructions to a self-compiling BF implementation, recompile it with itself, and then update it to use the new instructions.

Topics

Solar-powered portable computers

Kragen Javier Sitaker, 2016-09-17 (updated 2018-10-28) (15 minutes)

Today, with off-the-shelf microcontrollers, you could build a computer, somewhat superior in speed and screen size to a Sun 4, that runs off a solar cell from a solar calculator. Memory is somewhat trickier. The hardware BOM cost is about US$30 or US$40.

Computational power would not be the power bottleneck, even with a fairly powerful CPU; rather, memory power (to erase Flash especially) and display power are likely to outweigh the CPU.

Suns

First, a word on the class of machine that desktop software was designed for.

The Unix workstation revolution was sparked by what Raj Reddy called the "3M computers": one MIPS, one megapixel, one megabyte of RAM, for under a megapenny. This was what you needed for a PARC-style GUI. The SUN (later called the Sun-1) was one of these, as were the NeXT, the Apollo, and the doomed PERQ. These machines all came with dozens of megabytes of disk.

The diskless Sun-3/60 I used to use as an X terminal is about 3 MIPS (InfoWorld, 1988-11-14, "Sun Shows Tempest Versions of Sun-3/4 Workstations", by Scott Mace) with an 1152×900 display (I think it has a bwtwo and a cgfour in it, but I think I was using a monochrome monitor). By the end of the 1980s, various Sun 4 models (based on the 1987 SPARC family) exceeded 10 MIPS; the contemporary DECStation 3100 was 20 MIPS, with DEC forcing its competitors down to a price of a thousand dollars per MIPS. Some DECStations at UNM provided internet access to 50 or 100 concurrent users when I started using them around 1994.

By that time, much more than a megabyte of RAM was commonplace on these workstations. There was a memory-price bubble from about 1992 to about 1996 during which a RAM cartel held prices steady at around US$40 per megabyte before finally collapsing to the Moore's-law trend line at about US$10 per megabyte; before that, memory was halving in price every couple of years, so, in 1990, it was around US$80 per megabyte. My Sun-3/60 from 1988 has, IIRC, a full complement of 48 megs of RAM, which at US$160 per megabyte would have cost Sun about US$8000, a bit out of line with the cost of the rest of the machine — but I suspect that RAM was added later.

But 16 megs would be an entirely reasonable amount.

So, our target is 10 MIPS, 1 megapixel, 16 megabytes of RAM, and let's say a gigabyte of disk.

Displays

An E-ink display needs about 1 mW on average to keep it updated for continuous comfortable reading. (This number comes from the Keyboard-powered computers (p. 2220), which says that a 6" E-Ink display needs about 750 mW during a 120 ms screen update, for 122×91 mm at 167 dpi, 190 nJ/pixel, 8100 nJ/mm²; 8 20-em lines of 7-point text work out to 49 mm × 20 mm, or 8 mJ per update, and

about 50 words, so about 0.16 mJ per word; at 350 words per minute that's almost a milliwatt. Larger text is more power-hungry.)

They can be a little hard to come by; the Seeed Studio one I found on DX is AR$390 (US$26) and only 200×96 pixels at 111 dpi (57 mm × 29 mm), enough for 12 lines of 40 characters. Old reflective passive-matrix feature phone replacement displays might be a viable alternative, and they are very cheap if you can find them (US$3, say). The Nokia 2730 display costs AR$78 (US$5) on MercadoLibre at the moment; it's QVGA resolution (which I guess is 320×240) and does 200 dpi 18-bit color. More recent smartphone displays are probably too power-hungry.

Microcontrollers

Various kinds of off-the-shelf low-power microcontrollers consume a bit less than a nanojoule per instruction, which has been stable for a decade or more and might finally start trending down again.

Processing power can scale down smoothly with available energy up to a point. Atmel picoPower ARMs, which go up to 48MHz (at 1 insn/cycle), do 1–10 µW in standby (and only 250 pJ/insn) but take 4–20 µs to wake up. Some MSP430s are below 0.3 µW idle, while others are in the same range as the Atmel chips, but wake up much faster. (These numbers come from file low-power-micros in this repo.)

ARMs are ubiquitous these days, and they're power-efficient and do a lot of work per clock cycle and per instruction. However, it might make bootstrapping quicker to use a different processor architecture, one that has existing self-hosted assemblers, compilers, and development environments available as relics from the homebrew personal computer era that have some chance of running in the very limited RAM on a microcontroller. That basically means 6502, 8080, or 8086, and nowadays that basically means 8080, specifically Z80.

Zilog — now a fabless semiconductor house — still makes Z80s (now the "eZ80" and "ZGATE" lines) for microcontroller applications. Toshiba has a Z80-compatible microcontroller line called the 870/C. Unfortunately all of these are fairly overpriced and underpowered. A typical part is the EZ80F93AZ020SG, which costs US$6.14, runs at 20MHz, and has 64K of Flash and 4K of on-chip RAM, but supports off-chip RAM too. It's not designed for low-power operation — it sucks 180 milliwatts when not sleeping, working out to a rather hefty 9 nanojoules per instruction, assuming one instruction per cycle. Even Zilog is switching to ARMs nowadays.

For a while, many MP3 players used Z80-derivative DSP chips like the ATJ2085; some of them were called "S1mp3", and an alternative firmware was developed for them. Digi-Key doesn't carry the ATJ2085.

Nowadays it probably makes more sense to try to run a Z80 emulator in software on an ARM core than to run an actual Z80. An LPC4310FET100.551 runs you US$7.74 from Digi-Key, has 168 kB of RAM, is ROMless (it has a bootloader in ROM that loads code from an external NOR Flash or other device at startup), runs its two ARM cores at 204MHz, and runs down to 2.2 volts, and even then it uses only 160 milliwatts, less than the EZ80F93AZ020SG.

Solar calculators

I walked by a cheap-shit electronics store today and saw a bunch of small solar calculators for AR$40 (US$2.60). It occurred to me to wonder how much computational power you could run off one of their solar cells.

I think they use inefficient amorphous silicon photovoltaic cells rather than the standard 16%-efficient polycrystalline silicon used for rooftop and utility-scale solar panels, and the cells look like they're about 9 mm × 36 mm. This means about 320 milliwatts of sunlight falls on them even in direct sun. If we suppose it's 9% efficient, as Wikipedia's "Thin-film cell" article says of production thin-film amorphous (silicon?) cells, then that's 29 mW, which at the 1.7-V a-Si bandgap voltage would be 17 mA.

At a nanojoule per instruction, 29 mW is 29 MIPS!

```
You have: 9 mm * 36 mm * 1000 W/m^2 * 9% / nanojoule
You want: MHz
        * 29.16
```

Energy storage in capacitors

Can you extend the usefulness of a wimpy solar cell with some solid-state energy storage?

Off-the-shelf ceramic capacitors can store in the range of 1–4 mJ; you can get a 1 mJ 6.3 V 47 µF ceramic capacitor, holding 1 mJ, for US$0.12. If you were to try to stuff the calculator with US$2.60 worth of these, you would have 21 mJ of storage, enough for about 21 million instructions (or 84 million on the picoPower chips). This is enough for a minute or so of CPU word processing, but it's not enough to update the display.

Garden lights and batteries

I bought some solar garden lights at the supermarket a few months ago for AR$40 (about US$2.90 I think; XXX check this). They have a 300 mAh 1.2 V NiCd AA cell inside powering a white LED and a polycrystalline solar cell on top that's about 40 mm × 40 mm; they claim it provides light for about 8 hours, although I haven't tested how much current it takes. It probably doesn't have a MPPT controller (the only component visible on the tiny PCB is a through-hole ⅛ W resistor, although there's a tiny blob of epoxy that might be hiding something). But it probably doesn't need one, in the sense that the cell should produce about a quarter watt in full sun, which would fully charge the cell in 1.4 hours:

```
You have: (300 milliamp hours 1.2 volts) / (40 mm * 40 mm * 1000 W/m^2 * 16%)
You want: hours
        * 1.40625
```

So it's probably limited by the battery, not the solar cell, so it probably fully charges the battery to its full charge of 1.3 kJ (!!), which is 1.3 trillion nanojoules and thus about 1.3 trillion instructions, which would be 1.3 million seconds (about two weeks) at 1 MIPS, or about two weeks of continuous E-ink reading. ¼ W at 1 nJ/insn is of course 250 MIPS.

Memory

Unfortunately, the microcontrollers I mentioned above are plenty fast, have very little memory. A typical example is the NXP MKW01Z128 mentioned in License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162): a 48 MHz 32-bit RISC in-order Cortex-M0 CPU with a built-in 600 kbps license-free RF transceiver, with 128 KiB of Flash and 16 KiB of RAM. (It uses about 1.2 nanojoules per instruction if it's receiving data at the same time.)

For many year, it has been normal for a desktop computer to have about as much RAM as it can access in a second, which works out to have been very roughly a megabyte per megahertz. You will note that this microcontroller is nowhere close to this balance, with 48 MHz (and about 48 MIPS) but 0.144 megabytes, a factor of some 300 away from the balance for a desktop computer. This is a very common problem with current microcontrollers.

At this point, old 36-pin and 72-pin SIMMs are e-waste that can be freely scavenged or bought for prices in the neighborhood of a dollar or less, typically ranging from 4 MB to 32 MB. But these run on 5 V and typically have standby currents of a few milliamps and operating currents of over an amp, which means they use on the order of 5000 mW running at full speed and tens of milliwatts just to retain data. So they aren't really an option.

Chips to look at:

https://www.digikey.com/product-detail/en/issi-integrated-silicon-solution-inc/IS42S16100H-7TL/706-1446-ND/5683868
https://www.digikey.com/product-detail/en/issi-integrated-silicon-solution-inc/IS62WV1288BLL-55HLI/706-1046-ND/1555419
https://www.digikey.com/product-detail/en/microchip-technology/23LC1024-I-SN/23LC1024-I-SN-ND/3543084
https://www.digikey.com/product-detail/en/microchip-technology/23LCV1024-I-SN/23LCV1024-I-SN-ND/3543093
https://www.digikey.com/product-detail/en/on-semiconductor/N01S830HAT22I/N01S830HAT22IOS-ND/6166720
https://www.digikey.com/product-detail/en/fujitsu-electronics-america-inc/MB85RS1MTPNF-G-JNERE1/865-1255-1-ND/4022688
https://www.digikey.com/product-detail/en/rohm-semiconductor/MSM51V17405F-60T3-K/MSM51V17405F-60T3-K-ND/2695010
https://www.digikey.com/product-detail/en/adesto-technologies/AT25SF041-SSHD-T/1265-1131-1-ND/4824165
https://www.digikey.com/product-detail/en/issi-integrated-silicon-solution-inc/IS25LQ080-JNLE-TR/706-1463-1-ND/5872437
https://www.digikey.com/product-detail/en/fremont-micro-devices-usa/FT25H04S-RT/1219-1190-1-ND/5875686
https://www.digikey.com/product-detail/en/winbond-electronics/W25X40CLZPIG/W25X40CLZPIG-ND/3008616
https://en.wikipedia.org/wiki/Programmable_metallization_cell
https://www.pjrc.com/mp3/simm/simm.html

There are reasonably inexpensive SRAM chips out there that can provide a megabyte or two of RAM for a few dollars, but nothing bigger. DRAM is thousands of times cheaper (right now Pricewatch lists DDR3-1600 DIMMs at US$30 for 8 GiB, under US$4/GiB, which would be 6¼¢/16 MiB) but very power- hungry. Even just retaining data in DRAM uses a lot of power.

(You will note that US$4/GiB today in 2016 and US$40/MiB in 1992 is a factor of ten thousand in 24 years, which is a bit over 13 doublings — a doubling time of about 22 months, reasonably close to the Moore's trend line I mentioned.)

Flash memory, by contrast, can typically retain data for ten years while turned off and dissipating only tiny amounts of energy it had stored while turned on, and always at least a few months. And it's even cheaper than DRAM: Pricewatch lists 64GB SDHC cards at US$18, or US$0.28 per GB, almost 15 times cheaper than DRAM. It even uses a little less power to read than SRAM or DRAM do. But erasing Flash is enormously more power-hungry than erasing SRAM or DRAM.

(Other kinds of nonvolatile memory — FeRAM/FRAM, MRAM, CBRAM, PCM RAM — have broadly similar characteristics to Flash.)

It's tempting to think of Flash as having similar access characteristics to a hard disk, but that's only really true for erasing. Reading written data is very fast, and can typically be done a byte at a time only a few times slower than reading entire blocks. This means that with the appropriate data structures, Flash can replace much of the RAM in our 1990 workstation, as well as its disk.

Erasing and writing data to Flash is about a thousand times slower than reading it.

Writing to erased blocks is somewhat fast, Even when you have to write by pages, they're 256-byte or 264-byte pages, and writing them takes about 700 microseconds — you can do 1400 pages per second on a single cheap Flash chip, compared to the 50 random spinning-rust disk writes you could do in 1990, or the 120 you can do nowadays. Sequentially, you could write a megabyte per second (2048 pages) to pre-erased disk at the time, or a third of a megabyte per second (1400 pages) to pre-erased Flash now. And these Flash chips are so cheap that it makes sense to use several of them, multiplying your bandwidth and allowing block erases to be concurrent with reads and writes.

Many Flash chips have bit-serial interfaces, which makes them much easier to hook up to a microcontroller. None of the SRAM or DRAM chips I've been able to find on Digi-Key have bit-serial interfaces, which means that driving them from a cheap microcontroller is going to require hooking up some kind of shift register to control the address bus.

So maybe we can get by with only a megabyte of real RAM and, say, 32 gigabytes of Flash.

There's a distinction I'm ignoring here, which is that the Flash that's fast to read a byte from is NOR, and a megabyte of it costs about US$0.46. So 32 gigabytes of it would be about US$15000, which is significantly more than the US$9 of NAND Flash. In practice you probably want all three types: maybe a couple megabytes of real RAM (US$8), 16 megabytes of NOR Flash (US$7), and 32 gigabytes of NAND Flash (another US$7).

Final bill of materials

Topics

A brief note on autonomous cyclic fabrication systems from inorganic raw materials

Kragen Javier Sitaker, 2018-04-27 (1 minute)

"Cyclic fabrication system" is a term due to Moses, Yamaguchi, and Chirikjian for, roughly, a fabrication system that can reproduce itself — a self-replicating robot. Such systems, once they exist, will eliminate scarcity of many or most material goods, make interplanetary mining an annoyingly ubiquitous reality, and convert problems like global warming and asteroid strikes from existential risks into manageable problems.

So how do we build one? This depends in part on the environment (terrestrial, space?) and the available materials (iridium, platinum, water, aluminum oxide, carbon, iron, nickel, oxygen?).

I'm going to restrict my attention here to inorganic raw materials, even though this body is typing this in an environment with a locally high density of available organic materials with very nice engineering material properties.

Fired clay

Fired clay is the fundamental technology for human-driven (non-autonomous) terrestrial cyclic fabrication. Although tools of stone figure prominently in the human geological record, going back a hundred and fifty thousand generations, we developed pottery some fifteen hundred generations ago, and this is currently the basis for all of human industry.

Clay has

Topics

The Adafruit Feather

Kragen Javier Sitaker, 2018-06-30 (1 minute)

The Adafruit Feather 32U4 seems like the modern successor to the Arduino. It has a slightly smaller form factor (22.9 mm × 50.8 mm), comes with a LiPo battery-charging circuit, weighs 4.8 g, and it's based on the ATmega32U4 at 8 MHz at 3.3V. This is a step down in computational power, but also in power consumed. But it has 20 GPIOs, 7 PWM pins, 10 analog inputs, a prototyping area, and its pin spacing is breadboard- and perfboard-compatible; and it supports USB directly, so you can directly do things like keyboards.

And of course it's software-compatible with Arduino, although it's running at half the cock speed.

Instead of "shields", the Feather has "wings", which plug into optional female headers you can solder on. There are a few dozen wings available, including things like Wi-Fi, OLED displays, etc. I²C is the primary means of communication between the Feather and its Wings in order to get a limited amount of stackability.

There are also Feather base boards with other CPUs, apparently including a Cortex-M0 with Wi-Fi (an ATSAMD21 and ATWINC1500, US$35), a Cortex-M4, an ESP32 ("Huzzah32", US$20), an ESP8266 ("Huzzah", US$17), and an STM32F205 with Wi-Fi ("WICED", US$35).

Topics

• Electronics (p. 3430) (138 notes)
• Microcontrollers (p. 3580) (29 notes)
• AVR microcontrollers (p. 3337) (20 notes)

Parallel NFA evaluation

Kragen Javier Sitaker, 2015-09-03 (updated 2015-10-01) (8 minutes)

How can you parallelize or incrementalize the evaluation of a finite state machine? Although this has been considered in the past in the context of parallelizing text search queries like WAIS across Thinking Machines, this is a particularly important problem right now, because tokenization is almost always the performance bottleneck of simple compilers and parsers, and parallelization is the key to performance on modern computers.

Background

All of this is well known; it's included in Blelloch's 1993 review of parallel prefix sum computation but was almost published by Ladner and Fischer in 1980, and then was published by Hillis and Steele in 1986.

Explanation

A simple and general approach which yields logarithmic running time for an NFA on a sufficiently large parallel machine is as follows. Assume WOLOG that the text length n is a power of 2. Begin with an ε-free NFA. For each character `c[i]` for `i` from 0 to n–1 of the input text, in parallel, compute a relation `t[i, i+1]` as the set of edges in the NFA that transition on `c[i]`. Then, in lg n parallel steps, coalesce adjacent intervals in `t` into larger intervals with this compose operation: compose(a, b), where a and b are relations, is the ordinary relational composition operation, which produces the relation (`s[j]`, `e[j]`) for the maximal set of `s[j]`, `e[j]` such that for each `j` there is at least one `m` such that (`s[j]`, `m`) is in a, and (`m`, `e[j]`) is in b. In each parallel step h, we are coalescing intervals of size 2^{h-1} with this composition operation.

That is, in step $h > 0$, for each `i` divisible by 2^h, we compute

`t[i, i+`2^h`] = compose(t[i, i+`2^{h-1}`], t[i+`2^{h-1}`, i+`2^h`])`

using the values we computed in step $h - 1$.

The size of the relation for any given interval of any size is capped at the square of the number of states in the NFA, so in the worst case, it does not increase with the length of the text, so for a given NFA, this takes Ω(lg n) runtime in the worst case. Average-case analysis is trickier, in part because it depends both on what an "average" NFA is and on what an "average" text is, but my intuition is that, in practical cases, the relation size for a text span of any length more than a few characters will be under, say, 10.

Sooner or later you need to constrain the leftmost state to be the NFA's initial state; it is probably best to do this by a special case in computing `t[0]`, since that will save work in all the nodes that include it.

Having done this, you can propagate the results back down the tree of intervals in another lg n steps, providing you with the exact reachable set of NFA states at each character boundary in the initial input.

This is the standard Blelloch prefix-sum algorithm, merely using

the relational composition operator as the reduction operation rather than numerical addition. (Apparently Kogge–Stone is more common in current codebases?)

In theory, you can get about a 5% improvement in total runtime by coalescing triples of adjacent intervals in each step rather than pairs. In practice, depending on the relative costs of communication and computation, the optimal branching factor may be something different from 2, 3, or 4. For example, if communication has high latency, it may be 32 or 1024. (This also depends on the characteristics of the NFA: a higher branching factor may reduce the size of the intermediate-result relations substantially.)

Reducing an NFA to a DFA is useful for many computational models, but in this case, the larger number of states in the DFA is likely to be a handicap. You probably want to be coalescing your intervals using the smallest alphabet of states you can get away with.

In practice, you probably want to do most of the computation serially; if you have 8192 cores, the best you can do is an 8192× speedup (ignoring superlinear speedups from caching and the like) so you might as well start out generating 8192 `t[i, i+m]` values on the 8192 cores by linearly running the finite automaton over each of 8192 chunks (probably as a lazily-materialized DFA rather than as an NFA), followed by, say, 13 steps of alternating communication and coalescence.

Applications

Using Bjoern Hoehrmann's new parsing algorithm "parselov", which conservatively approximates the language of a context-free grammar by compiling a stack-limited conservative approximation of its PDA into a finite automaton, it should be possible to perform nearly all of the work of parsing in parallel using this algorithm. However, I'm not entirely sure how big parselov's finite automaton is, and if it has a sufficiently large set of intermediate states, this might not work.

Google Code Search used to provide a public regular expression search over all public code. For normal kinds of regular expression searches, where you aren't interested in matches bigger than a few megabytes, parallelization on a cluster is probably not useful. If you do need larger matches, for example on genome or proteome searches, parallelism may be useful.

Using Levenshtein automata, you can generate reasonable-sized NFAs to find strings within a small Levenshtein distance of a given string. (Baeza-Yates and Gonnet, I think, came up with a different way to do this that takes advantage of bit-parallelism in modern CPUs and is therefore usually faster in the serial case.) However, if the text being searched is static enough to be indexed, it is probably faster to run Levenshtein automata on a suffix array of the text, especially now that we have compressed indexing and practical linear-time suffix-array construction algorithms.

You can use a variant of this algorithm to incrementally update finite-state-machine results — the construction is the usual construction of an incremental algorithm from a parallel algorithm, where the high concurrency of the dependency graph of the values computed during the execution of the algorithm translates into being able to change some of the input without invalidating much of the

total data flow graph. Concretely, if one of the input blocks at the leaves of the tree changes, you recompute its mapping of starting parse states to ending parse states, then propagate that change back up and down the tree, potentially affecting the parses of all subsequent blocks.

You can also use this approach to parse a *substring* of a sentence of a regular language, simply by starting the parse of the leftmost block with the "superposition" of all states, like the other blocks, rather than the usual initial state. This is potentially useful, for example, for syntax highlighting in a text editor (or in a code snippet in a web page) or for parse-error recovery in a compiler. Moreover, the substring parse can then be extended in either direction by inspecting a larger part of the sentence.

Topics

The details of the GPU in this laptop

Kragen Javier Sitaker, 2018-10-29 (2 minutes)

So this laptop has an NVIDIA Quadro K1000M, according to `lshw`:

```
       *-display
            description: VGA compatible controller
            product: GK107 [Quadro K1000M]
            vendor: NVIDIA Corporation
            physical id: 0
            bus info: pci@0000:01:00.0
            version: a1
            width: 64 bits
            clock: 33MHz

            capabilities: pm msi pciexpress vga_controller bus_master cap_list ro
om
            configuration: driver=nouveau latency=0

            resources: irq:16 memory:f2000000-f2ffffff memory:e0000000-efffffff m
emory:f0000000-f1ffffff ioport:5000(size=128) memory:f3080000-f30fffff
       *-multimedia
            description: Audio device
            product: NVIDIA Corporation
            vendor: NVIDIA Corporation
            physical id: 0.1
            bus info: pci@0000:01:00.1
            version: a1
            width: 32 bits
            clock: 33MHz
            capabilities: pm msi pciexpress bus_master cap_list
            configuration: driver=snd_hda_intel latency=0
            resources: irq:17 memory:f3000000-f3003fff
```

This card is reputed to have 192 Kepler-architecture shader cores (plus another 192 that are locked), and its performance is similar to the GeForce 630M. It runs at 850 MHz and has 2GiB of RAM, "16 texture mapping units and 16 ROPs".

Pixel Rate: 3.400 GPixel/s
Texture Rate: 13.60 GTexel/s
FP32 (float) performance: 326.4 GFLOPS
FP64 (double) performance: 13.60 GFLOPS (1:24)
DirectXL: 12.0 (11_0)
OpenGL: 4.6
OpenCL: 1.2
Vulkan: 1.1.82
CUDA: 3.0
Shader Model: 5.1

326.4 gigaflops sounds like a lot. The Intel GPU in my ultrabook (see Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033)) is 51.2 gigaflops, and its CPU is 25.6. So it's a bit more

than 4× the speed of the ultrabook.

It's from 2012 and still sold for US$60 in 2017.

The memory system is 900 MHz DDR3 and 128 bits wide, so it can do 1800 million 128-bit transactions per second, for a total bandwidth of 28.8 gigabytes per second.

The GeForce 600 series page on Wikipedia has further details. The GeForce 630 is listed as "Entry level".

By contrast, a current NVIDIA Volta card is the US$9000 Nvidia Quadro GV100, with 14800 gigaflops, plus tensor processing units that do 4×4 FP16 matrix multiply-accumulates.

Topics

Notes on QR code capabilities on typical Android hand computers

Kragen Javier Sitaker, 2018-09-10 (2 minutes)

Alejandra's cellphone automatically scans QR codes when they show up in the camera app, popping up a translucent notification. If they are a Mecard (or presumably a VCARD) it offers to add a contact, attaching the photo from which the QR code was snagged. If they are text, it displays about the first 27 characters of it, displaying newlines as spaces, and offers a chance to copy it to the clipboard. If they begin with "http:" (I guess?) it has a "chain link" button to follow the link. Gzipped data shows up as a question mark in a diamond, the "substitute" symbol.

Presumably it also supports vCalendar and Wi-Fi network codes.

Encoding life.py (1987 bytes) resulted in a QR code it failed to recognize. However, angleadd.py (1060 bytes) resulted in a scannable QR code, which resulted in text that could be pasted into a notepad, containing the full Python program; however, this doesn't work reliably. With -s 1 and the default PNG type in qrencode, I get a 117×117 barcode. fraktur.py (566 bytes, 89×89) worked somewhat more reliably, including at 2×2 pixels per module (which I guess means 50 modules per inch?); the UTF-8 decoded properly but most of the Fraktur glyphs are missing from the phone's fontset.

Alejandra installed an QR-code scanner from F-Droid, and it was able to scan the Mecard, but it didn't recognize it as a Mecard.

https://github.com/zxing/zxing/wiki/Barcode-Contents goes into some more details, though from a 2016 perspective.

https://qrworld.wordpress.com/2011/06/16/how-to-create-qr-codesoo-for-business-cards/ talks about Mecard vs. VCARD in 2011. Apparently by then Mecard was actually more widely supported than VCARD.

Aaron Toponce went hardcore for his business card: https://pthree.org/2010/01/07/qr-code-with-mecard-and-hcard/

Topics

- Hand computers (p. 3492) (10 notes)
- Business cards (p. 3355) (2 notes)

State of the world 2016

Kragen Javier Sitaker, 2016-09-05 (10 minutes)

An apocalyptic religious cult has captured an area the size of Syria in an effort to bring about the end of the world. It conducts mass executions of "apostates" about once a month, and enslaves many of the people it conquers. This year, it says, the Mahdi will appear; he will lead them to victory against Rome's crusader armies in Dabiq.

The United States routinely uses tele-operated flying military robots to incinerate people suspected of opposing it in areas far from any battlefield. A repentant US robot operator has just gone public with his story after personally killing some 1600 people, many of them innocent. A repentant CIA agent has just been released from prison, where he was serving a sentence for blowing the whistle on the CIA's illegal torture program. No other participants in the torture program have yet been charged with crimes by a court.

A resurgent undemocratic Russia has just revealed that, for the last fourteen years, rogue intelligence agents in the United States has been inserting undetected spying software into computers around the globe, including reprogramming the controllers in their hard disk drives.

A year and a half ago, a soft-spoken ex-CIA employee fled the US, first to Hong Kong and then to Russia, having released a massive collection of classified documents showing the extent of the US's illegal spying programs at home and abroad. He is still in hiding in Russia. Among other things, he showed that US spies had broken into all of the data centers of many US technology companies and stolen their customers' information. Now, US tech companies are in open conflict with the intelligence agencies. Many of them are redesigning their products and systems to frustrate intelligence agencies.

At about the same time, the founder of Russia's biggest online social networking system, through which hundreds of millions of users maintain contact with their friends, fled to the US after legal harassment by the Russian government. Today he is working on an encrypted messaging system.

Meanwhile, in a room in the Ecuadorean embassy in London, there lives an Australian hacker and award-winning journalist whose hair went white in his thirties. He has not left the embassy for years. The Ecuadorean government granted him political asylum three years ago when the British government sought to extradite him to Sweden, from which he claims he would be extradited to the US. A secret US grand jury investigation of his journalism started years ago, after it published a large collection of classified US embassy cables which have been a crucial source for journalists and scholars of international relations ever since.

The political assassination of nearly the entire staff of a satirical weekly newspaper in France has resulted in a wave of anti-Muslim violence throughout the country and new restrictions on speech and freedom of association throughout the continent.

In Argentina, where the currency is collapsing, the president has just been indicted by a state's attorney for covering up an Iranian

terrorist attack on a Jewish building. The previous state's attorney was found dead in his apartment the day before he was to present the charges against her.

The Pope and the president of Russia both announced last year that the Third World War had begun.

A volatile, pseudonymous virtual currency built on cryptography, implemented by software written by a mysterious figure known only by the pseudonym Satoshi Nakamoto, has a US$3 billion market capitalization (down from US$17 billion) and half a billion dollars a month in trading volume. Many countries have outlawed it. Custom chips designed only to process this virtual currency now populate multi-megawatt data centers throughout the world.

Meanwhile, prominent technologists warn that artificial intelligence is likely to end the human race, and that automation will destroy most jobs even sooner.

US citizens seeking solace in traditional belief systems have begun to shun vaccination, and diseases such as measles that were eradicated generations ago have begun to return to the nation. Polio begins to spread internationally in areas afflicted by armed insurrection, especially Pakistan, where many polio vaccination workers have been killed since the US hunted down jihadist leader Osama bin Laden in 2011 using a fake vaccination program.

A US movie studio shut down a couple of months ago for a period of time and pulled a satirical movie about North Korea after all its internal emails were stolen and released. The US government blames the attack on North Korea, but an anonymous group of bored teenagers seems more likely.

A worldwide conspiracy of anonymous bored teenagers known as Anonymous has been temporarily politically neutralized by, primarily, US law enforcement, after they began to uncover and publish information about illegal conspiracies within large US companies. The US has one journalist imprisoned for attempting to report on the group. A splinter group numbering a few hundred thousand has been honing harassment tactics against US feminists, particularly those who criticize video games.

Uruguay, Colorado, Washington, Alaska, Oregon, and Washington, DC legalize recreational marijuana. Belgium legalizes euthanasia for the terminally ill, and the Queen of Belgium, who has been sick for years, dies.

A year ago, Ukraine exploded in protests that overthrow its pro-Russian government; Russia annexes part of Ukraine. US elites are debating the merits of arming Ukraine. Ukrainian nationalists are using hand-launched artisanal flying robots made in Dnepropetrovsk to surveil pro-Russian forces from the air, but not yet to kill them. Ukrainian children collect fresh remnants of artillery shells.

Switzerland unpegs the Swiss franc from the Euro, immediately bankrupting several major foreign-exchange brokerage firms who had offered their customers extreme leverage with which to speculate.

Greece, suffering a worse economic downturn than the Great Depression, elects a party that claims to be a coalition of the radical left, which immediately begins to reverse privatization and government downsizing, and plan to shut down the concentration camps into which the previous government had herded asylum seekers. The new government hires a video game company's

Economist-In-Residence as its finance minister, and the continent is full of speculation that Greece will withdraw from the Euro. Greek banks seem to be collapsing as a result.

Australia, by contrast, continues to herd asylum seekers into concentration camps, where they are imprisoned for the rest of their lives for the crime of not being Australian.

The US continues to lead the world in imprisoning its own citizens with 1% of its adults in prison, rivaling the Stalin-era USSR on a per-capita basis, and remains the only country in the Americas to execute them. Although it continues not to publish any official statistics on the matter, its police continue to extrajudicially execute several times as many US citizens (Wikipedia counts 593 cases in 2014) as are executed legally (Wikipedia counts 39 cases in 2013), a practice that has provoked widespread protests for the last several months, centering around the execution of an unarmed teenager in Ferguson, Missouri. China, however, leads the world in executions, executing thousands of people per year, more than the rest of the world combined.

Amidst all of this, the total installed volume of photovoltaic panels (almost entirely made in China, which is now the world's biggest economy) is increasing by 30% per year, and oil prices have fallen by more than half to their lowest price in years, plunging Venezuela into a serious economic crisis and putting economic pressure on Russia. After adopting controversial new mining techniques that cause frequent earthquakes in previously seismically stable areas, the US has returned its oil production to their peak levels in the 1970s, and now its oil production is exceeded only by Saudi Arabia and Russia.

Meanwhile, atmospheric CO_2 has shot past 400 ppm, higher than it has been in 4.5 million years, 120 ppm higher than before the Industrial Revolution, due to burning fossil fuels. As a consequence, 2014 was the hottest year in recorded history; NASA reports that the southwestern US is almost certain to face "megadroughts" lasting decades, droughts unmatched in a millennium.

This is the political landscape of the early 20th century.

A quarter of the world's population has Linux-based personal computers in their pockets, running at microwave frequencies and always wirelessly connected to the internet. This number doubles every year. These computers are not secure, and their users do not have root, so they can be used to spy on the users without their knowledge. The US commonly does this to target people for assassination by flying robot.

We have a free encyclopedia universally available to anyone on the internet, and that anyone can edit. It continues to grow linearly, reaching almost 5 billion articles, the equivalent of about 2000 volumes. Against all odds, it continues to remain more reliable than any previously written encyclopedia. Encyclopedia Britannica has ceased print publication. Many newspapers have also ceased publication.

All new luxury cars now come with onboard supercomputers that record GPS traces of their travels. This data is sent back to the manufacturer to analyze.

Topics

Vector instructions

Kragen Javier Sitaker, 2017-07-19 (2 minutes)

Old Crays had vector instructions. These used a "vector length" register and a "vector mask" register to specify which items in the vector to process.

On the Cray Y-MP C90, there were eight vector registers, V0 to V7, each containing a 128-element vector of 64-bit values; a vector instruction would process corresponding elements of two of these registers, two at a time, and deposit the results in another vector register. But you didn't have to process all 128; the "vector length" register could terminate the process early if you didn't have that many.

These vector instructions could be pipelined in the sense that the result from one could be fed incrementally to another, as long as they used different functional units.

Vector registers were loaded from and stored to central memory using a "block transfer" with a first word address, an increment or decrement (stride), and a vector length; and this could participate in the pipelining. Thus a sequence of vector instructions could construct a flow graph that loaded some sequences of values from main memory, processed them, and wrote them back to main memory, all in a pipeline, but the viewpoint of the assembly program was that it was conducting a series of in-order operations on large vectors. As the manual says, "The CRAY Y-MP C90 computer system allows a vector register reserved for results to become the operand register of a succeeding instruction."

There were also scalar registers, which could be used with vector instructions as arithmetic operands.

The only vector integer operations provided were sum, difference, and, or, xor, and leading zero count. In floating-point, you had the usual operations, except that instead of division, you had only reciprocal.

The "vector mask" register could be used to select elements from one vector or another, or to replace selected elements with zero or a scalar register. (In related selection operations, you had a "register shift" group which I don't understand.) The vector mask could be set by numerical tests on a vector register: [<=>≤≥≠]0. There was also a variant of the VM-setting instruction that puts the indices of the matching elements into another vector register, although I don't think there was a way to use those indices except one at a time.

Topics

GPT-2 sets the scene

Kragen Javier Sitaker, 2019-11-22 (updated 2019-12-01) (22 minutes)

I thought I'd try out GPT-2's abilities at generating some adjective-heavy descriptive prose, using Talk To Transformer.

I think this kind of thing could be dangerous for humans with an inclination towards psychosis; they could easily start to think these texts were messages for them from God or demons or the CIA or something. Even humans not prone to psychosis are vulnerable to the Eliza Effect, where by pareidolia they read all kinds of meaning into any text with even the slightest semblance of coherence. And this text has more than the slightest semblance.

Yawning in the echoing forest

My prompt:

As the red sun rose over the mists of the echoing forest, I awoke to a spiderweb sparkling in the fog, yawned and stretched, and smelled the musty-sweet smell of the bog.

GPT-2 completions:

• I yawned again and stretched, and ate breakfast. I ate a little more, and yawned some more. I was very hungry, but I couldn't figure out what my new surroundings were, except that there were new memories that I didn't want to lose. After lunch I walked down into the bog, pulling my shoes onto my feet and shivering.
• It reminded me of hot stewed serpents and sweet jellies and it terrified me, like at any moment I might step onto wits one of those monsters of my dreams and get jolted like Milla Jovovich. I emerged from the forest, using a talisman to summon my trusty Glaive. The
• Like a cheap copy of Gray's Anatomy, a billowing furry body yawned "I love you", it said in my ear. I woke again, and gazed at the black waters of the Old River. "You're still here", I croaked, and let a limp hand break free of my splintered wrist. "Don
• I crawled to the wick of the bulbous flame, snuffed it out, and brought it to the campfire. [hippies including Wes Craven! Craven doesn't actually use the term "personas", the last thing I want to do is abuse a trend in film criticism that some people cherish. "Person
• When I remembered this I laughed, and took a moment to look around. Only one person on the park had been sleeping that night, a slim man with a thin mustache and a big purple head, who had not attended to his own business but was watching me with that perplexed look of his.

"I sleep

Well, that's interesting. Sleepy amnesiac exploring nature; terror fantasy; extreme furry S&M porn; film criticism, and some kind of creepy sci-fi.

Trumpet flower moths in the night

My prompt:

The trumpet flowers opened at dusk and released their intoxicating scent, drawing giant clearwing moths from near and far, who fluttered like doves as they came to sip the sweet nectar.

GPT-2 completions:

• A stormy sky blanketed the land, thick with thunder and lightning. A sea of rubble had amassed where the town of Mazarkis had nodded in unison. A massive oak tower had caught the lightning that struck the town and released sparks that cut the air like torches. "It is here," the rain outside his window
• Until the lights were gone, the guards seemed to sit there for hours
On the smooth-skating marble atrium, chattering idly in their
Pause one, out of sight, as if just as anxious to get back to their
Equipment as to allow happy times for themselves.
At noon, another guard came by with a bou
• They would land and start up the sky lane a few times in an hour, or perhaps take a nap. While women in my family were the avid bakers, the fondness of my father for stone roasted, braised and popcorn tofu grilled chicken barbecue johndoes not even begin to cover what a goddamn delight he found each of these
• The night continued to be dark and bright. About that time the sound of water from a well almost drowned us all. Ten feet down the cave was a basin, in which two men and two women were bathing.
But I wondered why they were there, and found it very puzzling. We were none of us dumb enough to
• The birds were gone by daybreak, and the scent they left was said to be iridescent in the cold morning air. When the night arrived, the blossoms burst out into the night sky, reaching skyward for a few moments before lowering on either side. By the time dawn emerged, most of the blossoms had withered into black globes

Again, a great variety: a suspenseful fantasy scene (what happened to the town of Mazarkis? Did the lightning reduce it to rubble?), poetry about guards on (?) an atrium, weird family cooking recollections, some kind of group adventure story, rapid cycling of dark and light with "iridescent scent". I love it! It's better than a cut-up!

Boca Juniors celebration

My prompt:

Safe within my massive artificial mountain of glued-together pebbles and sand, I listened to the wild festivities celebrating our victory over Boca Juniors without, as the revelers danced in the streets and beat their wild drums.

GPT-2 completions:

• But I was too saddened by our defeat for this. I too was beat down by feelings of regret and remorse that no-one, not even George Ross, could stop from allowing it to happen. Why? Because he knew, without a doubt, that there was always an impossible task
• Not to be outdone, the team took to the field to advance the story of the year by beating Santos on match day and earning the franchise's first win over the most hated team in Liga MX. After a rematch, Montagliani presented us with the trophy, giving his team the trophy

• So I lay back, relaxing and savoring the moment. The lovely Filipino lady came in behind me and as she rubbed my shoulders I began to lighten up a little.
She drew my hand towards her breasts and kissed me like I had

never been kissed before and she wanted
• My sun-kissed skin flowed in waves along my pebbled thighs as I
• What a war these factions have fought for a week. Boca Juniors, I fear not their armies, nor their luck, nor their rich coffers, but what would a victorious opponent for them? Would they tolerate our spirit, our style of play, and our joy of life? Surely no

This is hilarious! A tortured self-reflection, sports nonsense, regular porn, very weird but short porn, and some kind of political agitprop against a soccer team.

Swordfight scene

My prompt:

Gray struck him again, harder, with the edge of his shining scimitar, but the Mazarkin's armor held. With a great cry, he leapt into the air and kicked at Gray's head, but Gray had already ducked, and as the other landed on his feet, Gray whirled to face him, battered shield up.

GPT-2 completions:

• "Better protect my lord!" he shouted, "If you kill me, your queen dies!" To Gray's surprise, Brown's face went white, and his spear dropped. The elves
• Gray only had his blade at his side. "I'm through with you, Garro ! "

"No," says Garro, looking up at Gray with eyes hard as stones. "
• It took half a dozen blows before Gray was soundly knocked to the ground. Once on the ground, Gray saw that the Mazarkin had seemingly dodged everything he could think
• "You! A coward. You must have been blind, talking Your Grace. But not brave. A coward. You cannot give up so easily. This will make you strong."
• With one hand, he banged the edge of the shield upon Gray's head again, causing blood to come spilling from the wound. Then he lifted his great sword again and

So, pretty consistent in that they're all continuations of the fight scene, but several are dialogue, and one has elves. They're shorter than the earlier continuations and thus less interesting, which I think is a limitation of the Talk To Transformer site.

Vampire predation

My prompt:

As her hot blood filled my mouth, she trembled in helpless delight. I felt her fiery life filling my veins, dispelling for a little while my ghostly pallor, as I held her delicate form firmly

• in my arms. I swallowed my hard-earned seed. I was still feeling pretty drained when she finally came down from her thrilling, and very intense, release.

When I had finally finally calmed down, I sat back and looked at what I had just accomplished. Her delicate flesh was bulging from her shapeless suit. Even the crisp dressing on her soft flesh was
• by my dainty hands. I willed her to fall into deep sleep, and so drove her to all her. Lips, and to her great desires.

I willed all in me now to throw my strength into renewing this last struggle, even that I might not fail to make her submit to my

advances. Soon, the heavy suck
• in my arms.
"Haah, I'm getting too old for this now."
"Do not be, the life energy is a nourishment that is best given when married. However, because we married earlier, we would need to make it look like our love is growing as if it were a flower that you throw out into the wind. With this, we
• in my embrace. With a tongue as rough as the hungry pit on her throat, my lips caressed her beautiful jade-cut lips, and softly nibbled her hard tender nips. Turning her head to face me, she whispered: "You have captured my spirit, and I am yours forever."
To read part one of our story click here. To read part
• in my arms.
My hands had been in rough contact with her soft and delicate torso, and I could feel a pair of soft yet firm buttocks growing between my fingers. With no other thought than that she needed what I wanted, I guided my now numb
Okay, so, normal if somewhat surreal porn, incoherent vampire story, more coherent vampire story, romantic porn with links, and very weird porn that I actually cut off here because it got more sexually explicit than I want to include here. "Firm buttocks growing between my fingers" is a very surreal image.

An agony of grief

My prompt:

I crumpled to the ground in agony. "Why?" I asked. "Why him? Why now?" Suddenly knowing I would never again in my life see his dear, sweet face, never again touch his cheek, filled my chest with an unbearable tightness.

GPT-2 completions:

• "Why?" "What can it mean?" "Do you know how I hate you, O God?" "Then answer me. And she has been here, who knows not, never seen the place? "Oh, Christ
• I cried aloud, weeping with all my yet unearthly force, as though my heart had been broken, my blood were I heard some one moaning, muffled somewhere deep in the woods, the roar of an unknown beast. N
• I ran back to the big tree and tried to put out the fire. I had to wash my face before I could get a word out. Even the pebbles in my face and from the tree trail made my arm ac
• "Why this at last ?" I mumbled. "Why, what's the matter?" One minute. . . . . and then. "Shame on the hotel!" It was 9:30 P.M. Why? Why
• My palms had been moistened and my cheeks throbbed when I touched him, but this new feeling struck me too and all my physical and psychological strength fled. As he passed, I went to one corner of the room and sat quietly

This one is harder. Emotionally, I mean. The first is some kind of bargaining with a very interactive sort of God; the second continues in the theme of grief and weeping ("cried aloud" would have been better than "asked" in my prompt, I think); the third is some kind of emergency response; the fourth is incongruously pedestrian; and the fifth is about accompanying a loved one in their death.

FOREST NEEDS MOAR MODIFIERS

My prompt:

The ancient, verdant forest sighed sweetly and fragrantly in the warm, soft breeze. Excitable young bluebirds flitted brightly between the thick, secretly radioactive branches,

GPT-2 completions:

• happily humming their song and laughing as they threw confetti into the air. The hum was barely noticeable to her; her already radiant mood greatly improved. The towering cypress trees looked grand in the background of the sparkling white lakes and sparkling white waters.

'Beautiful, isn't it? Wonderful!'

'Oh, this is just like our precious home.'

'Our home

• poking and pecking each other, and eyeing the red-headed teacher intently. It was a moment of peace before all hell broke loose...

The Vermilion Girls: Softly Sharp Knives of Laughter

From the Sword Art Online anime is currently airing a one-shot manga, and by my count there is at least 6 different chapters in this series. In each volume,

• their vibrant, brilliant blue feathers just as bright as those of a current-day hummingbird. It was like one long day spent reading a long, windy passage of literature, or smelling a powerful perfume.

What-what-what-what? Sniffle.

'Waaaaaaaah'

As if somehow overcome by pleasure and a lingering sense of despair, Erin

• a new specimen emerging from the shadows and spying on me. I approached the bird and gave a quick squawk. "Hah, alert! You're too early. A feeder came by just before the first meal!" I was almost there. "Ugh, sorry." "You won't catch a feeder there." "Uh-huh. Not any time

• and those who weren't looking for secrets quickly went in to the forest in a hurry. After searching through dozens of trees and examining their contents, they finally realized something: it was a lockpick.

How to hide a lockpick? It was easy, the trees did not have to be smothered with soot and they did not need to be cut down to have easy access to the

This was an experiment in going overboard on adjectives and adverbs, and in some cases it was successfully transferred to the output, to the point of self-parody at times; "the sparkling white lakes and sparkling white waters," really?

The first is some kind of party-nostalgic scene (bluebirds throwing confetti!); the second one is some kind of blurb for a lesbian porn manga or something (whose awesome title is original); the third one is some kind of fiction but I don't know what; the fourth has anthropomorphic birds seeking "feeders"; and the fifth is some kind of demented adventure.

No modifiers, just a helicopter

My prompt:

Rayentray slid through the door half a second before it slammed, belting herself

into the helicopter's copilot seat. "Ready!" she grinned, as the blades deafened the neighborhood and they began to rise.

GPT-2 completions:

• Angie shuddered and slammed the door shut behind them, as the primary ignition roared to life. The helicopter's mated engines roared to life, as the aircraft soared above the rooftops. The two pilots, each carrying one M107 Thunderchief heavy machine gun, looked down
• "Well, I'm expecting lots of calling, but no need to hurry. It will be taken care of. I could be here any minute. Come on!" Right at the moment their throats were becoming over-taxed, Sabertooth I stepped into view through the door,
• "We're a military-transport aircraft, and I'll get into any a simulation where I can beat you any time!" she laughed. Soon, the aircraft had climbed so high in the sky, the top-shelf F-111s that were based at Roswell with their Predator-2s were unable
• she spoke into the radio. "Crew elevator deck 703, VCHADE 703, emergency closing."

She flipped the switch and as they hurtled upwards the top canopy opened, revealing a woman with pigtails and a headband.

"Shall we do this?" Judy smiled, gripping her pistol tighter
• "You want to stop? Here we come."

The exterior walls of the home were immaculately maintained. The interior might as well have been a spaceship. The home belonged to Georgette and her small brood, who lived out their afternoons in comfort, ignorance and opulence. It was

So, lots of military stuff; an interesting thing is that three other characters appear, and all three are also women, like Rayentray. Unless Sabertooth is a person.

The fake military terms are really impressive. The "M107 Thunderchief heavy machine gun" sounds so convincing you'd think it must have been lifted wholesale from somewhere, but in fact the M107 is a sniper rifle, one person can't carry a heavy machine gun, and the Thunderchief was a US fighter-bomber aircraft. Similarly, the F-111 was retired in 1998, before the Predator really entered use, and the Predator-2 is a handheld toy quadcopter; you wouldn't use either a Predator or a Predator 2 with an F-111 even if they were in service at the same time.

This was an experiment in going to the other extreme: *no* adjectives or adverbs (except, I guess, "copilot seat" uses "copilot" as an adjective). To compensate, I tried to channel my inner thesaurus, Tom-Swifting things a bit.

#2 is pretty incoherent.

Invented Wikipedia articles

The Arbëreshë are a real people, Albanians who settled in Italy several centuries ago, but essentially everything GPT-2 says about them is false. They generally belong to the Orthodox Christian Church.

My prompt:

[edit]
Religiously, most Arbëreshë belong to

GPT-2 completes:

• the Church of the 'Everlasting God' (Qanat, a god eternally active, and the offspring of all other gods). Indeed, they consider the Dark Lord of the universe to be a reincarnation of the god(s) originally at war with Elantris and are constantly warning others to stop him. Their dread of the Mortal World greatly offsets their obsession with magic, which often turns them astray and warps their thinking. They don't fear death, even at the hands of
• the Church of Sarben, which has its headquarters in Anarres. The belief in the omnipotence and total justice of the Gods and Goddesses is generally the basis of most Arbëreshë beliefs, which sometimes come to the fore in disputes or personal conflicts. Even if one does not follow any specific religion, Arbëreshë still consider themselves to be descendants of the gods and goddesses of their forefathers. For example, Velyar 'cause it is his beloved
• several different Indo-European groups (see Wikipedia:Indo-European), with Ashirasë, the highest, being associated with the Nordics (see Norse, Bulgars).

In their position of the highest divine state, the Shaidar-i-Asrar or Shaidar-i-Ištar were historically known as the primary divine of all the Arbëshes, and the more exalted of the divine-favored inhabitants of the world. Ashiras
• the Old Fëan religion. Most have strong ties to Athelâr, and it is through that religion that most Arbëreshë worship and take their oaths. There are some more distant Arbëreshë, especially those who worship other deities, who do not belong to a single religion. It is rare, however, for Arbëreshë to do so without also taking an oath from them.[1][2]

Some Arbëreshë form their
• a tradition that dates back to the founding of the city by Tulu (also called Tulu-túru) in the year 498 Eru. Since that time, it has flourished, but still today the chief religion is still that of Tulu-túru. During the present times it was preached, practiced, and followed by most people in the region, who consider the church to be the ancestor of the current Tulu-túru, and especially its priests and lector

I think the first of these is wonderfully creative: a god that is the *offspring* of all the other gods. The second refers to Anarres, the fictional impoverished anarchist moon in Le Guin's *The Dispossessed*.

Topics

• Humor (p. 3511) (9 notes)
• Fiction (p. 3454) (7 notes)
• Pompous (p. 3641) (6 notes)

Gauzy shit

Kragen Javier Sitaker, 2018-11-04 (4 minutes)

Trying to calculate caustics (see files Caustics (p. 1619) and Caustic simulation (p. 1454)) and "Windy" Oona Raisanen's oscilloscope traces and whatnot, it occurred to me that there's a very simple solution.

Let's think about caustics first. You have some uniform light density ρ on some original surface, you know, a caustic generator, and that density gets splayed across some other surface in some nonuniform but computable and continuous way. If you want to find the density at a pixel of the caustic, the simple solution is to take the *inverse* of that computable, continuous mapping, and map the pixel center point through it; it will give you zero or more points on the generator, since the inverse is not, in general, a function. You need to sum the density from those zero or more points.

The density from each point, however, depends on the "focusing" from that point on the generator (which may be infinite). Consider a small parcel of area ε around the point on the generator. This parcel, with total brightness ρε, gets spread over some area on the caustic, which is some function of the partial derivative matrix of location on the caustic according to location on the generator. I think maybe it's the determinant. Anyway, then you take the reciprocal of that, and you get the brightness contribution at that point. (You may want to add some small value to the absolute value of the reciprocal to prevent singularities, analogous to how the finite spatial coherency of light in space prevents singularities in the electromagnetic field.)

I think that you can do this more efficiently in the usual cases by computing not just this brightness but also its gradient in the neighborhood of the sample pixel. Then you can do the full calculation of the inverse and its gradient at some subset of points, like 1/16 or 1/64 or something, and just interpolate at other points.

If your sample points are evenly spaced, this is a lot like the algorithm for computing Perlin noise, except that not just the gradient but also the base brightness value is nonzero at the sampled point. And of course it's not random.

However, I think you can do better than this with an approach similar to raymarching with signed distance functions. You don't need very many samples in areas with very uniform gradients, and in areas where the gradients are changing rapidly, you need a lot of samples. You can try to approximate this by sampling the derivative matrix of the gradient (is that the Jacobian?) or you can compute the gradient over an area using interval arithmetic, rather than at a point using ordinary arithmetic.

(Yes, that's the Jacobian, and the Jacobian determinant gives you the brightness, or the reciprocal brightness, depending on which Jacobian you're considering.)

In the case of Raisanen's oscilloscope traces, we have to solve the same problem once for each column of oscilloscope pixels. Within that time interval, our brightness is spread out uniformly over a single dimension of time, but there may be very many points in time that map to the same vertical pixel. We can sum the reciprocals of the

(absolute plus epsilon) derivative of the waveform at these points to get the total brightness deposited at the pixel. And, as before, if the second derivative is very small, we don't need to sample very densely.

In both cases, of course, we could avoid the inverse problem and just iterate over parcels of the generator. But I think that's almost certain to lead to sampling some areas far too densely while sampling other areas far too sparsely.

Or will it? Approximating the inverse function, in the general case, seems like it'll mostly depend on that same kind of sampling, so it may not actually gain us anything to start from the pixels on the caustic rather than the generator.

Topics

• Algorithms (p. 3310) (123 notes)
• Graphics (p. 3483) (91 notes)
• Caustics (p. 3368) (6 notes)

A stack of coordinate contexts

Kragen Javier Sitaker, 2007 to 2009 (9 minutes)

I thought I had a novel way of thinking about automatic APL-style lifting of pointwise operators, and it yielded some interesting suggestions, but I think it turns out it kind of sucks as a generic way of thinking about APL operators. Here I document my blind alley in case other people find it interesting.

The Problem

In APL, you can write 4 + 5 and get 9, or 4 + 3 4 5 6 and get 7 8 9 10, or 4 5 6 + 2 3 8 and get 6 8 14. And you can similarly add a vector to a matrix, and each element of the vector will be added to an entire row (or column, depending on your point of view) of the matrix. And so on.

The Basic Approach

So here's a way to think about that.

Suppose we have a (possibly finite) stack of coordinate contexts, each of which contains one coordinate, and each of our values is a function of this stack. Scalars don't look at the stack at all. Vectors look at the top item on the stack to figure out which scalar to return. Matrices look at the top two. Now we consider our language's e.g. multiplication to be pointwise multiplication over these functions.

Suppose we use + to denote mathematical addition and +. to denote the addition operator of our language. Then we can say

```
x + y = \ C . x(C) +. y(C)  where C is a coordinate stack.
```

So if x and y are both scalars, then so is the result; if one is a vector and the other is a scalar, then the result will be a vector, in that it will depend on the top item of the coordinate stack for one side of the addition; and so on. This automatically provides the desired behavior for binary operators on vectors of different ranks.

Unintuitively, the indices that we normally think of as changing fastest are further down the stack.

Can we define other APL functions this way?

Transpose and Compose

If we want to define the transposition operator that transposes the first two indices, we can define it in terms of cons, car, and cdr functions on coordinate stacks. car returns the top index off the stack; cdr returns a coordinate stack missing that top index; and cons returns a coordinate stack with a new car pushed on top of what was previously there.

```
transpose x = \ C . x(cons (car (cdr C)) (cons (car C) (cdr (cdr C))))
```

That's just the Forth SWAP or the PostScript exch operator. It's probably clearer if I write it with Haskell's right-associative infix cons operator ':' and pattern-matching:

```
transpose x = \a:b:z . x (b:a:z)
```

If we define transpose this way, then we can apply it to a scalar --- but the resulting value will require at least two stack levels that it actually ignores, i.e. will be an infinite matrix. I could add error-checking cases to transpose to handle this.

(XXX this is wrong. What you want is something that transposes the two leftmost indices, the ones that change slowest, not the ones that chnage fastest.)

There's a useful operator that Numeric calls "take", and which APL implements by indexing one array with another (e.g. 4 5 6[1 2 1] yields 4 5 4, and 4 5 6[2 3 rho 1 2 2 1 3 1] yields 2 3 rho 4 5 5 4 6 4).

In its most basic form it only applies to a one-dimensional array as its first argument. I'll call it "compose". Can we define this "compose" operator?

```
compose(lookuptable, indices) = \C.lookuptable(cons(indices(C), nil))
```

That is, we create a new coordinate stack to use to index the lookuptable, consisting only of the values from indices. That's OK for one-dimensional lookuptables, but maybe we can generalize it so that it does the right thing if the elements of the lookuptable are vectors or more complicated things.

To do that, we need to provide the leftover elements of the coordinate stack --- the ones not needed to index into indices --- to lookuptable in place of the nil.

Taking Rank Into Account

So now we need to consider each array as more than just a function from coordinate stacks to scalars; now we also care about its rank (although that was obvious from the start, since without knowing its rank and also its size in each dimension, we can't do simple things like display it on the screen).

So now I am losing interest, because I think this way of thinking about it doesn't actually simplify things any over the traditional view (where an array consists of a vector of dimensions and a vector of contents). But I'll pursue it just a little bit longer. Let's call the contents function of x "x.f", and its rank function, which tells how many items it cares about on the coordinate stack, "x.r", and write { f = x.f, r = x.r } to express a new one identical to x. So we have

```
x + y = { f = \ C . x.f(C) +. y.f(C), r = max(x.r, y.r) }
transpose x = { f = \a:b:z . x.f(b:a:z), r = x.r }
compose(lookuptable, indices) = {
    f = \ C . lookuptable.f(cons(indices.f(C), dropn(C, indices.r))),
    r = indices.r + lookuptable.r - 1
} where dropn(X, 0) = X and dropn(X, N>0) = cdr dropn(X, N-1)
```

The more general form of the "+" case for scalar binary operations is

```
scalarop(op, x, y) = { f = \C . op(x.f(C), y.f(C)), r = max(x.r, y.r) }
```

You could treat arbitrary unary scalar operations as if they were

vectors, then apply them with compose:

```
opvalue(op) = { f = \C . op(car C), r = 1 }
```

Or you could have a scalarunop:

```
scalarunop(op, x) = { f = \C . op(x C), r = x.r }
```

(It seems like you ought to be able to derive the "r"s mechanically from the expressions for the stacks to which you're applying the "f"s. In the transpose case, the stack is still the same height, so the rank is the same; in the compose case, we drop indices.r items from the stack and then add one, so we need an extra indices.r - 1 items. But I'm not quite sure how to do that yet.)

The other basic Forth stack manipulation operators also have interesting functions. DUP, diminishing the rank by 1, takes the diagonal of a matrix; DROP, increasing the rank by 1, turns a vector into a matrix so that it can be applied column-wise. For now I'm not going to think about OVER (diagonal between first and third dimensions?), ROT (rearranging the order of the top three coordinates?), NIP, TUCK, 2DUP, and 2DROP.

Iota

One-dimensional monadic iota is fairly straightforward; its content function doesn't depend at all on its argument. (Only its size does, and we haven't talked about size yet.)

```
iota = { f = car, r = 1 }
```

APL's N-dimensional monadic iota is kind of stupid; it takes the values produced by the one-dimensional iota --- as many as needed --- and then reshapes them into the requested shape.

Numerical Python has a somewhat more general operation called "indices". Applied to a vector describing an N-dimensional shape, it returns N sets of indices, each of which has the shape requested, and contains the Nth index. For example:

```
>>> Numeric.indices((2, 3))
array([[[0, 0, 0],
        [1, 1, 1]],
       [[0, 1, 2],
        [0, 1, 2]]])
```

So the first top-level item has the first coordinate of each location, the second item has the second coordinate, etc. That is, Numeric.indices((a,b))[0][x][y] == x, and Numeric.indices((a,b))[1][x][y] == y. This sounds stupid but is very useful if you want to tabulate values of N-dimensional functions.

Unfortunately it's a little clumsy to work with in the one-dimensional case, because iota (which it calls "arange") is wrapped in another layer of nesting. It's a little clumsy to define in this system; although its pointwise value is, like iota, just one of the coordinates, the number that specifies which coordinate is hidden deep in the stack, and counts backward from there.

```
indices(dims) = {
    f(C) = C[length(dims) - C[length(dims)] - 1],
    r = length(dims) + 1
}
```

The "- 1" is there to make the coordinates zero-based.

Reshape

Reshape (dyadic rho) is extremely simple in the standard representation, and rather a dog's breakfast in this one.

Reduce

Reduce ought to work by default along the axis that changes slowest, and that's kind of ugly here.

Conclusions

It was a cute idea --- especially the Forth stack manipulations turning into operations --- but I think it makes things more complicated, not less.

Topics

- Programming (p. 3658) (286 notes)
- Facepalm (p. 3450) (24 notes)
- Stacks (p. 3730) (21 notes)
- Arrays (p. 3326) (17 notes)
- APL (p. 3320) (9 notes)

Obscurity platform

Kragen Javier Sitaker, 2018-04-27 (1 minute)

One useful measure for security through obscurity might be running on a platform that doesn't have a lot of out-of-the-box exploits available for it. GEF supports x86, ARM, AARCH64, MIPS, PowerPC, and SPARC, but GDB also supports ARC, MicroBlaze, m68k, NDS32, Nios II, S/390, and TMS320C6x; of these, m68k is pretty comfortable and has a lot of tooling available, so might be a good choice.

The J1A or RISC-V might also be pretty reasonable platforms, but a bit bleeding-edge.

Linux on m68k seems to be pretty dead (web page not updated since 2000; last Debian release Etch) and FreeBSD is gone too, but NetBSD still supports m68k. Also HP-PA, SuperH, and VAX. These are "Tier II" ports. There are a number of JIT 68000 emulators for popular machines; Basilisk II (GPL) successfully runs old versions of MacOS, and there's a reasonable amount of interest in QEMU support for it; QEMU at one point in 2014 got to being able to boot Linux/m68k on an emulated Mac, barely. These unfortunately require Macintosh ROMs.

GCC also, crucially, supports m68k, up to at least GCC 4.9 and maybe GCC 5, I'm not sure.

(Unicorn also, unfortunately, supports m68k.)

Topics

Maybe Counting Characters in UTF-8 Strings Isn't Fast After All!

Kragen Javier Sitaker, 2007 to 2009 (15 minutes)

These are responses to Reddit comments on http://canonical.org/~kragen/strlen-utf8.html.

I Think I Was Wrong

From reading the Reddit comments, I now think that the results I got didn't justify the conclusion I drew, and the evidence now suggests that iterating over the code points in a UTF-8 string *is* significantly slower than iterating over the code points in an ASCII string. Thanks, guys! I should dig into it a bit more and see if I can learn more.

It looks like my results were really skewed by using `gcc -O` instead of pretty much any other level of optimization. By ill chance, I happened to test things on the single optimization level that would give the results I got.

"bonzinip" writes:

> crap. no one in their right mind would use lodsb on a modern processor. my results are
>
> ```
> 3:                  strlen(string) =   33554431: 0.013685
> 3:                my_strlen(string) =   33554431: 0.024342
> 3:             my_strlen_s(string) =   33554431: 0.099565
> 3:         ap_strlen_utf8_s(string) =          0: 0.102122
> 3:         my_strlen_utf8_c(string) =          0: 0.058268
> 3:         my_strlen_utf8_s(string) =          0: 0.099565
> ```
>
> more or less the same for all three benchmarks

Well, I pointed out in the page that `lodsb` was a bad idea, even on an old processor; on old processors, you should use `scasb` instead, as `gcc -O` does. XXX aristotle used lodsb

The really interesting thing about your results, though, is that `my_strlen` is more than *twice as fast* as any of the UTF-8 versions, and *four times as fast* as what the C compiler used in this case. If that's the best we can do for UTF-8, then counting characters in UTF-8 strings *isn't* fast. It's slow!

I'll try to reproduce your results; what processor are you using, what compiler and options, and what is the generated assembly code?

"Porges" writes:

> strlen is also consistently fastest for me
>
> ```
> 1: all 'a':
> 1:                  strlen(string) =   33554431: 0.019204
> 1:                my_strlen(string) =   33554431: 0.035398
> 1:         ap_strlen_utf8_s(string) =   33554431: 0.068897
> 1:         my_strlen_utf8_c(string) =   33554431: 0.072316
> 1:              my_strlen_s(string) =   33554431: 0.120852
> 1:         my_strlen_utf8_s(string) =   33554431: 0.137072
> 2: all '\xe3':
> 2:                  strlen(string) =   33554431: 0.019043
> ```

```
2:               my_strlen(string) =   33554431: 0.035056
2:        ap_strlen_utf8_s(string) =   33554431: 0.068909
2:        my_strlen_utf8_c(string) =   33554431: 0.071979
2:             my_strlen_s(string) =   33554431: 0.120309
2:        my_strlen_utf8_s(string) =   33554431: 0.154263
3: all '\x81':
3:                  strlen(string) =   33554431: 0.019083
3:               my_strlen(string) =   33554431: 0.034908
3:        ap_strlen_utf8_s(string) =          0: 0.068871
3:        my_strlen_utf8_s(string) =          0: 0.069123
3:        my_strlen_utf8_c(string) =          0: 0.071848
3:             my_strlen_s(string) =   33554431: 0.120325
```

Edit: I should note this is -O2, not -O as in the post.
... I've posted a followup to this. http://reddit.com/info/6moej/comments/

As with bonzinip's results, the UTF-8 counters are half as fast as the C-coded byte counters, and about one fourth as fast as `strlen`. What processor is this, and what is the emitted assembly?

"splidge" comments on "Porges'" post:

Yes, for me running with -O gives results similar to the original post whereas -O3 gives results similar to yours.

In fact, running without -O gives pretty much the same results for the default strlen() as -O3, with -O coming out significantly slower. The other C implementations improve from no optimisation to -O to -O3 as you would expect.

(I don't have anything to say about this, except that it's kind of depressing, but I thought it was important to archive.)

"rolfr" writes:

This guy's using obsolete performance measurements. Number of instructions in the inner loop hasn't been important for ages; it's all about the pipeline characteristics of said instructions.

Yes, you are right. I'm pretty ignorant about assembly-language optimization on modern processors, or, for that matter, on non-modern processors. Knowing I was ignorant, I only used instruction counts as a heuristic and relied on observed runtime measurements for my actual conclusions.

"Wavicle" writes:

To be fair, he bases his final conclusions on the observed runtime over large strings *of the same value repeated*. Thus he trains the branch predictor to always make the same optimization each time.

That's a fair objection. I wonder if it makes a large difference.

The Point

I didn't write that page to prove some predetermined point. I wrote it to document my (our) exploration of a hypothesis of Aristotle's, namely that iterating over the code points in a UTF-8 string was approximately as fast as iterating over the code points of an ASCII string.

Then, at the end, I listed some things I thought I'd learned in the process, including this #3:

Aristotle was essentially correct: the penalty for counting UTF-8 characters, or indexing into or iterating over the characters of a UTF-8 string, is very small.

As a result of this structure, the Reddit comments contained a lot of speculation about what "the point" of the page was, and a bunch of people missing it.

"ochuuzu1" writes:

> Also, plus, WTF: no one in their right mind would use `strlen()` in a performance-critical inner loop of any real application.
>
> You can tweak `strlen()` to make it run as fast as you possibly can, but it will never be faster than not calling `strlen()` in the first place.

Of course you are right. `strlen` is just a proxy here for the speed of iterating over the characters in a string, which is indeed found in performance-critical inner loops of many real applications.

"cracki" writes:

> haha. nullterminated strings are stupid. **counting** characters is stupid. it costs you nothing to keep the length around.

Same comment.

UTF-8 as an Internal Representation

"GolemXIV" writes:

> I hope that I should not have to use UTF as in memory representation at all (with exceptions of course). It would be so nice if I could leave UTF* to places where characters are stored or streamed.
>
> UTF/UCS combining marks means that programs cannot treat one code point as being the same as one unit for editing even when you use UTF-4 (UTF-32). That sucks small planets when not streaming or storing strings.
>
> Is there string libraries on programming languages where character type refers to a base character together with all the combining characters that are attached to it? I would like to work with vectors of character objects, not with code points.

I agree that often it would be much better to have vectors, or at least sequences, of character objects rather than code points or bytes or UTF-16 bytepairs. I don't know of any string libraries that work this way, but I'm pretty ignorant about Unicode, so maybe there are some.

It's a good point that converting UTF-8 to UCS-4 still doesn't save you as much hassle as one might naively hope, because of things like combining characters.

Occasionally, though, I've heard assertions that UTF-8 is very inefficient, because finding the Nth code point of a UTF-8 string is O(N). The hypothesis I started with was that (a) iterating over the code points is more important than indexing them randomly and (b) iterating over them is as fast as iterating over ASCII bytes. If that's correct, then while you might decide that UTF-8 is a bad internal representation for some reason or other, it shouldn't be because you're afraid it's slow. And GolemXIV makes a good point that even in UCS-4, you can have arbitrarily many code points that display in a single spot on the display.

"cracki" writes:

> besides, it's an encoding. you're meant to decode it if you need to work with the contents. hardly anybody has any excuse for not expanding utf-8 to 32 bit characters in memory. then, even indexing is constant time.

Well, regardless of what you're *meant* to do, you should decode it if that leads to a system that makes people happier --- say, running acceptably fast while being simpler and overall easier to debug and modify. A lot of times, finding a simpler way involves exploring a lot of ideas that sound kind of crazy, like not decoding your UTF-8. And most of them *are* crazy, and this one probably is too, but you have to explore them in order to find out.

Miscellaneous

"silon" writes:

> Please do not call the function strlen

I didn't; I called it things like `my_strlen_utf8_s`.

"bonzinip" writes:

> because the string instructions are very slow and go through the microcode sequencer. i would just use normal `mov` and `inc` instructions.

You'll notice that that's what the C version of `strlen` compiled to, and that it was faster than my dumb `lodsb` version and GCC's dumb `rep scasb` version.

"bart2019" writes:

> The storage of an integer is ridiculously little compared to the storage of the string contents itself.

I suppose that depends on how many of your strings are less than 4 or 8 bytes, doesn't it? I've written a number of programs nearly all of whose strings were that short.

(That said, I basically agree that null-terminated strings are a bad idea.)

Carl Friedrich Bolz ("cfbolz") writes:

> Implementing ropes well is not trivial. A while ago I tried to to implement the Python string type in such a way as to internally use ropes [in] PyPy. I never managed to get ropes perform significantly better than array-based strings for real-life benchmarks (although I admit I didn't try anything extreme with 2GB of text or so).

It's true that ropes, like other clever data structures with beautiful big-O numbers, tend to have higher constant factors. But I don't think they're all that much higher. (I seem to recall that the SGI STL includes a C++ rope implementation called "cord", and the Boehm garbage collector includes a rope implementation called "rope"; I might have gotten the names backwards, though.)

One of the trouble with real-life benchmarks is that people write real-life code to perform acceptably on the implementations they have; so they tend to heavily lean towards the things that were very efficient on the old implementation, and away from the things that you hope you're improving. Dick Gabriel wrote about this back in the 1980s in p.3, section 1.1 of Performance and Evaluation of Lisp Systems:

> There is a range of methodologies for determining the speed of an implementation. The most basic methodology is to examine the machine instructions that are used to implement constructs in the language, to look up in the hardware manual the timings for these instructions, and then to add up the times needed. Another methodology is to propose a sequence of relatively small benchmarks and to time each one under the conditions that are important to the investigator (under typical load average, with expected working-set sizes, etc). Finally, real (naturally occurring) code can be used for the benchmarks.
>
> Unfortunately, each of these representative methodologies has problems. The simple instruction-counting methodology does not adequately take into account the effects of cache memories, system services (such as disk service), and other interactions within the machine and operating system. The middle, small-benchmark methodology is susceptible to 'edge' effects: that is, the small size of the benchmark may cause it to straddle a boundary of some sort and this leads to unrepresentative results. For instance, a small benchmark may be partly on one page and partly on another, which may cause many page faults. Finally, the real-code methodology, while accurately measuring a particular implementation (namely, the implementation on which the program was developed), is not necessarily accurate when comparing implementations. For example, programmers, knowing the performance profile of their machine and implementation, will typically bias their style of programming on that piece of

code. Hence, had an expert on another system attempted to program the same algorithms, a different program might have resulted.

He goes into more detail in section 1.4.3, p. 28:

One way to measure those characteristics relevant to a particular audience is to benchmark large programs that are of interest to that audience and that are large enough so that the combinational aspects of the problem domain are reasonably unified. For example, part of an algebra simplification or symbolic integration system might be an appropriate benchmark for a group of users implementing and using a MACSYMA-like system.

The problems with using a large system for benchmarking are that the same Lisp code may or may not run on the various Lisp systems or the obvious translation might not be the best implementation of the benchmark for a different Lisp system. For instance, a Lisp without multidimensional arrays might choose to implement them as arrays whose elements are other arrays, or it might use lists of lists if the only operations on the multidimensional array involve scanning through the elements in a predetermined order. A reasoning program that uses floating-point numbers 0--1 on one system might use fixed-point arithmetic with numbers 0--1000 on another.

Bolz makes the same point in his blog post on the PyPy ropes:

Using ropes to implement strings has some interesting effects. The most obvious one is that string concatenation, slicing and repetition is really fast (I suspect that it is amortized O(1), but haven't proved it). This is probably not helping most existing Python programs because people tend to code in such a way that these operations are not done too often.

More to take into account:
http://www.reddit.com/r/programming/info/6lv0y/comments the original post http://www.reddit.com/info/6m0ej/comments/ reddit post of porges' response
http://porg.es/blog/counting-characters-in-utf-8-strings-is-faster porges' response (George Pollard)
http://www.daemonology.net/blog/2008-06-05-faster-utf8-strlen.html Colin Percival's response
http://porg.es/blog/ridiculous-utf-8-character-counting George Pollard's comment on Colin Percival's approach
http://www.reddit.com/r/programming/info/6m5yg/ reddit comments on Colin Percival's approach, which includes the lovely comment:

What would be even more useful, would be if people attempted to make a "fastest" random-access indexed lookup of a full Unicode codepoint in a stream or buffer of UTF8. If that could be made speedy enough, then there would be less of a need to store 16 or 32 bit codepoints internally, and always need conversions.

Topics

- Performance (p. 3621) (149 notes)
- Assembly language (p. 3328) (25 notes)
- Utf 8
- Unicode
- Strings

XCHG: An Archival Swap Machine

Kragen Javier Sitaker, 2014-06-29 (7 minutes)

Move machines suffer one big problem as a replacement for Brainfuck: the register operands are necessarily in pairs, which halves code density in some sense and means that reading the code can easily suffer from framing errors. $&@ might mean write to $ and then transfer & to @, or transfer $ to & and then read from @, depending on where we are.

As a fix to this, I suggest a variant move machine with 26 directly nameable registers, one for each English letter, and two instructions for each one: a capital letter to read it into the accumulator, or perhaps subtract it, and a lowercase letter to write the accumulator to it. So Ab would copy A to B, and Abc would copy A to both B and C.

A further convenience for compiled code would be to define the digits such as 4 as "multiply accumulator by 10 and add 4". This allows constants to be provided in a reasonable way.

Subleq is a universal OISC. So we can get all arithmetic from just subtraction; a single subtracting accumulator is thus a sufficient interface for all arithmetic. You could memory-map it with a single location, which gives the current total when read or subtracts when written.

Subleq, however, also includes a signed comparison and conditional jump. And to program anything more than state machines on it, you need self-modifying code, unless you make its output and one of its inputs indirectly addressed.

To get around the state machine limitation, you can memory-map a pointer register and its pointee, like the PIC. And getting a jump requires memory-mapping the program counter. The combination almost gives you a conditional jump; all that's left is a way to get a Boolean to index with, like a < instruction that fills the accumulator with its sign bit. That way you can index to either the program counter or the word after it.

This memory-mapping the PC thing means that your memory cells need to be more than a byte, so figure 32 bits.

One final thing needed to make the machine reasonable: a subroutine return mechanism. Let's say that when writing to the program counter directly (not as a pointee) we swap the would-be new program counter value with the accumulator. Then the callee can save it upon entry for return later.

Could that work as a general mechanism? It would eliminate the read-write distinction, but you would need a memory-mapped copy/discard device in addition to the subtractor, the pointer, the pointee, and the program counter. But you would get 80-some printable registers instead of 26. As a special bonus, you could claim the machine was "linear" or "reversible".

You also need input and output devices. The traditional Brainfuck ., interface seems adequate, but should use the accumulator contents, and can reliably report end of file.

I recommend writing code for this machine on fixed-width lines of 100 characters in order to make subroutine calls work reasonably.

This seems like a machine that should be nearly as easy to implement on most machines as Brainfuck, but much easier to generate code for. Let's take the always-swap option:

; always reads as the last value written to :, discarding written data
, discards written data and reads as the next byte of input
. writes output and reads zero
0123456789 multiply accumulator by ten and add a digit
- reads the subtracted total and also subtracts the current accumulator from it
' is a register pointing to memory
" is an alias for the place ' points to
< sets the accumulator to -1 if it is negative, 0 otherwise
^ is the program counter

9 instructions and memory-mapped locations, not much worse than Brainfuck's 8 instructions. Alternatively, without the cutesy swap thing, except for jumps:

Aa read and write bytes of input and output
0123456789 multiply by ten and add a digit
B reads the subtracted total
b subtracts the accumulator from it
C reads the value at the address written to D (maybe use B instead?)
c writes the value at the address written to D
Dd read and write a register that is normal except for controlling C
E reads the address of the next instruction
e sets the address of the next instruction, causing a jump, and also sets the accumulator to what the next instruction would have been
< fills the accumulator with its sign bit
Other letters read and write general-purpose registers.

This is 7 to 10 instructions depending on how you analyze it. It seems to me that these two formulations are close to the same complexity to implement, but the first one is much more novel, and either might turn out to be easier to program with — by which I mostly mean write compiler backends targeting it. You have the capacity to do subroutines/functions, multiple threads, dynamic dispatch, array indexing, and so on.

x = y + z is, with the swap machine, y-z--:;-;--x, I think, starting with a zero subtractor. We subtract y and z from zero to get the negated sum. We store that in : to use it twice to make it positive, then read the sum and store it in x.

Zeroing the subtractor is a challenge, because zeroing is irreversible. We need to subtract not only the subtractor's original contents, but also the accumulator contents that we folded into it in the process. After the sequence --, the subtractor has had its previous contents canceled, but it still has the negated original accumulator contents in it. :;--; saves these original accumulator contents in :, then wipes out previous : contents with ;. After --; we have wiped out the old - contents too, leaving the original accumulator contents negated in - and unnegated in : and the accumulator. Another - gives us the negated original in - and twice the negated original in -; now if we save the negated original in :, we can subtract it twice: :;-;- or in full :;--;-:;-;-. There may be a simpler way to zero the subtractor, but I think that way works.

Topics

Interesting features of the GNU assembler Gas

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

Interesting features of gas:

• multiple subsections: `.data 0`, `.data 1`, etc. You can switch around between them during assembly, and then they get concatenated when assembly is done. There's some ambiguity in the manual about how this interacts with the "." symbol.
• The "." symbol, which is like Intel's $, and which can be reassigned to move around and assemble things in funny places. I haven't yet tried using it to assemble code someplace we've already been (e.g. for a counted string). The documentation for `.org` says this won't work.
• `.fill` would be useful for filling an area with a pattern (despite its bizarre semantics.)
• there is of course a full conditional compilation system.
• there is of course a full macro facility, with default arguments, named arguments, recursive macros, string interpolation, the ability to redefine existing symbols, and optionally local variables.
• `.incbin` lets you suck in an external file, or part of an external file, as data.
• It uses Knuth's `1f`, `1b`, `2f`, `2b` local labels. These work remarkably well for something so archaic-looking.
• The `-a` option generates an assembly listing. There's a bunch of formatting stuff for the listings: `.eject`, `.title`, `.sbttl`, `.nolist`, `.list`, `.psize`.
• It supports bignum math and emitting 8- and 16-byte integers from them.
• There's a "section stack" so you can assemble into a different section for a while, then pop back to where you were, or even swap back and forth between two sections without knowing which ones they are. Unfortunately, at the moment, the section stack is only partly implemented as documented.
• `.print` prints stuff out.
• ELF apparently has a bunch of section flags: "allocatable", writable, executable, "mergeable", and a few more
• there's sort of support for defining struct fields with `.struct`
• `.intel_syntax` supports Intel assembler syntax.

Topics

DHT bulletin board

Kragen Javier Sitaker, 2016-09-07 (7 minutes)

I think you can build a DHT bulletin board system duplicating the functionality of Usenet fairly simply.

Usenet

Usenet, a peer-to-peer communications system started in 1980, provides a bulletin-board or threaded-discussion system. It consists or consisted of a bunch of "articles", each of which was identified by a "message-ID", categorized in one or more "newsgroups", and "referenced" zero or more articles that it was a "follow-up" to, typically the transitive closure of the "references" relation on the article it was replying to, with the ID of the message being replied to last. Typically, a Usenet news server would provide access to articles recently posted to some finite set of newsgroups, replicated to other news servers using gossip.

Typically, Usenet newsreaders would display a tree of the followup relation while displaying the body of a single article, which would often include pieces of other articles it was commenting on. JWZ devised the standard algorithm for deriving the tree from the "references" headers. Experimental visualizations such as Yee's threadmap (paper) have been tried, echoing the two-dimensional hypertext layout used in the Talmud for millennia.

I'm speaking in the past tense here because Usenet has largely succumbed to spam; the gossip protocol's lack of scalability and the needs of spam filtering resulted in a progressive centralization of Usenet servers; and web-based systems have largely supplanted it for day-to-day discussion.

Making message-IDs unique was always a bit of a challenge, in particular since the gossip protocol in NNTP used message-IDs to determine whether it already had an article, so you could halt the spread of an undesirable message by posting a competing article with the same message-ID, if you got wind of it soon enough. Modern peer-to-peer systems like BitTorrent, Git, Bitcoin, and Tahoe-LAFS usually use self-certifying names (such as secure hashes of documents or of public keys) in order to avoid such collisions.

System design

The system consists of a DHT storage system for immutable data and feeds for mutable data.

The DHT, storage, messages, and newsgroups

The system's immutable data is stored, potentially permanently, in a DHT on the internet; the DHT's basic interface consists of `key = PUT(data)` and `data = GET(key)`. The `key` is a 192-bit unique ID, which is 24 bytes in binary or 32 bytes in base64, such as "oruMktRkd127bWh6av4tXhJXiEscirL/" or "SUrY99PC96aUjJttKFjmvWTFFqkNXmt1"; it consists of a randomly generated 96-bit symmetric encryption key to use to decrypt the article with an authenticated mode such as GCM, and a 96-bit secure hash of the encrypted data chunk (for example, the first

96 bits of SHA-256). Only the secure hash is sent over the network to the DHT storage nodes in the GET operation.

The DHT is only able to store fairly small chunks of data, up to 16 kibibytes or so, so a single article may be split across many DHT chunks. One of these chunks is the "root chunk" of the article, whose key is the message-ID, containing the following data, together with its typical sizes:

• an "in-reply-to" field, with the message-IDs of zero or more articles to which it is a followup (24 bytes);
• optionally, a "supersedes" field, with the message-ID of a article that this one is intended to replace, which must be signed by one of the same authors (24 bytes);
• zero or more signatures identifying the authors of the article (64 bytes), who may be bitcoins;
• other human-useful metadata such as the subject/title, timestamp, byte count, and tags (128 bytes);
• the key to the root block of a Merkle hash tree containing the article contents, including possibly other metadata (24 bytes). Typically this tree will contain a single node, which is the body itself.

This is a total of about 264 bytes, and it's roughly what a Usenet "overviews" entry would contain for the article.

The Merkle hash tree in this case is made out of 192-bit keys, so it can support about 682-way branching; a single level of branching gets you to message bodies of 11 megabytes, two levels of branching get you to message bodies of 7.6 gigabytes, and three levels get you to 12.7 terabytes.

There are two kinds of "newsgroups" in this system: an "unmoderated newsgroup" consisting of a tag query over whatever articles you happen to know about, and a "moderated newsgroup" which is a special message content-type containing the concatenation of many message-IDs, each with the "root chunk" of the message appended to it in encrypted form, just as the DHT node would send it to you, so that you don't have to query the DHT to get each message's overview.

With this structure, a moderated newsgroup with a history of 100 000 messages would be a message of about 26 megabytes. New versions of the newsgroup, as long as the moderator doesn't reorder the overviews of the existing messages, will generally only alter the last chunk in the Merkle tree — so you can update your view of the moderated newsgroup by downloading only the root chunk, two levels of branching chunks, and the newly-updated chunk. (At the cost of revealing, perhaps, how much of the newsgroup you had before.)

Purging/expiry

DHT nodes need to adopt some policy for purging data that nobody will ever query once their storage media begin to get full; otherwise, it will become impossible to post new articles, and this could even be carried out as a denial-of-service attack by a . Inevitably, the system as a whole will expire data as old DHT nodes go offline forever and new ones replace them. (Unless the system is, at the time, dying, like Usenet.) So all of the data that ought to be preserved, in anybody's opinion, has to get "refreshed" by being

stored again in the DHT, at intervals.

Feeds

None of the above allows you to go from old data to new data, because old data can only contain the keys of older data. uery

Topics

- Programming (p. 3658) (286 notes)
- Systems architecture (p. 3691) (48 notes)
- Protocols (p. 3668) (21 notes)
- Decentralization (p. 3404) (13 notes)
- Distributed hash tables (p. 3410) (2 notes)
- Computer-mediated communication systems (p. 3379) (2 notes)

The Problem: Writing With One Access Pattern, Reading With Another

Kragen Javier Sitaker, 2007 to 2009 (19 minutes)

So I was talking to someone today about a network monitoring database problem I'd run into previously. A lot of network monitoring systems store many gigabytes of historical data about, say, bandwidth used per device, in order to support queries later on. Common queries include things like:

• what was the bandwidth used on device X in every five-minute interval over the last week?
• what was the aggregate bandwidth used by each of the devices in such-and-such a group?
• what were the five devices that used the most bandwidth over such-and-such a period?

It's nice to have queries like these be answered very quickly, but existing tools like Postgres and RRDTool don't seem to perform very well.

To be concrete, suppose that you have 100 000 devices, and you poll each one for input bytes, output bytes, input packets, output packets, input errors, and output errors once every minute. Each of these numbers may get too big to record in 32 bits, so it's good to use 64. So that's 8 * 6 * 100 000 = 4.8 megabytes per minute, or 7 gigabytes per day, or 200 gigabytes per month.

Disk Model

My standard model of a disk is that a disk is something that takes 8–10ms to start reading or writing at a random spot, and then reads or writes at 40MB/sec as long as it's reading or writing sequentially. Some disks (or disk arrays) are five times faster (bandwidth-wise) and some are five times slower, and most have faster seeks under some special circumstances, but that's generally the right ballpark.

An interesting number that comes out of this is the bandwidth-delay product, which is about 10ms * 40MB/s = 400kB. If you're alternating between reading or writing a chunk of some fixed size, and seeking to a new location, this is the chunk size at which you spend only half of your time seeking. Reading or writing in chunks much smaller than this means the disk spends most of its time seeking; reading or writing in chunks much larger means the disk spends most of its time transferring data.

Bandwidth varies between disks more than seek time, so bandwidth-delay products vary quite a bit.

Approaches to Solving It

The usual way of dealing with this is to use RRDTool, which stores each counter for each device in a separate file, and thins out old data so as to keep a fixed upper limit on the total number of samples in any one file. RRDTool makes queries about one or a small number

of devices quite fast, but updating of a large number of counters is slow, since each one is in a different file.

Because the data is so large, it's a bit expensive to keep it all in RAM, so it would be nice if both updates and queries had reasonable locality of reference. If you're updating 7 gigabytes of RRDTool files (11520 bytes per counter, to store 1-minute data for a day), you pretty much have to read and then write 7 gigabytes of data, which is going to take at least three minutes on a single disk; but we're hypothesizing that the actual amount of new data being written is maybe 4.8 megabytes. So the RRDTool approach imposes largish costs on mass updates due to poor locality of reference.

You could go to the other extreme and optimize for writes: write all the new counter data in one big 5-meg blob. But then reading the 1440 one-minute samples for a single counter over a day's time requires 1440 disk reads, 5 megabytes apart, and perhaps 10 seconds. This is not acceptable either.

My Proposed Solution: Tiling

This points at a solution I had suggested in "r-tree indices for database table clustered indices", http://lists.canonical.org/pipermail/kragen-tol/2004-January/000740 08.html which is to sort of divide the data into "tiles" of, say, 64 minutes by 1024 counters, each occupying a contiguous half-megabyte. We can assume (for now) that getting from any tile to any other tile requires a random disk seek. So if you're recording six new 64-bit readings for each of 100 000 devices, those 600 000 readings get broken up into 600 groups of 1024, and each of those 600 groups gets written into a separate tile. If each requires a separate seek, this should take about six seconds instead of six minutes. And if you're reading the 1440 one-minute samples for a single counter over a day's time, those will be spread across 23 tiles, so will require about 1/2 second.

That's an improvement, but there are several more directions of optimization possible: side files, tile ordering, thinned data, and grouping counters by type.

Side Files

First, we can initially append updates to a "side file" instead of sending them directly into their final locations, then eventually copy the data to the tiles where it will ultimately live. To start with, every query must read the entire side file, so you don't want it to get too big, and it cuts the theoretical write bandwidth of the database by a factor of three or four, since every update must first be written to the side file (with metadata), then read from the side file and written again to a new location.

But now writing 600 000 readings --- 5 megabytes without the metadata that tells what they are, and perhaps 10 megabytes with it --- takes a quarter of a second instead of six seconds, which seems like it's better than an order of magnitude speedup. However, the eventual writes to the tiles will still take time, but as explained below, we can accumulate several updates for each tile, and deliver them in the same number of seeks.

If we batch up four updates in the side file before flushing them out to the tiles, the side file will get to 40MB, which is maybe one second

of disk bandwidth. Copying it to the 600 tiles will take a second and a half of bandwidth and 600 seeks, so about 7–8 seconds to do the copying and 8–9 seconds in all --- about 30% of the total disk traffic the four updates would need without the side file. (This copying can be done incrementally rather than all at once, which allows you to batch up eight updates in the side file at a constant size of 40MB, or four at a constant size of 20MB. That's assuming it's practical to reorder stuff in the side file as some of it gets flushed out.)

A 40MB side file would take about a second to read off the disk, and in the absence of any disk buffering, every query would require that additional second. (The assumption is that since it's ordered by the time at which the updates were made, queries won't have particularly good locality of reference.) This would be a good reason to keep the side file small.

But, actually, you can probably keep the side file in RAM until it's a quarter of a gigabyte or more, which in this scenario would allow it to batch up roughly 500 full tiles --- and that's getting close to the point of diminishing returns, where maintaining a bigger side file doesn't actually save you any more disk seeks. Keeping the side file in RAM means that your queries don't suffer from this additional second of disk access time, although they may still have to access the data.

So, in this limit, if you buffer up all the new readings in memory, while also appending them to a side file (in case of a crash), and then writing them out to 600 new tiles when those tiles are full, then every 64 minutes, you write out 600 megabytes of data to the side file (sequentially --- about 15 seconds) and 300 megabytes of new tiles (with seeks in between, so 3 seconds of seeks and 7 seconds of data traffic, for 10 seconds of disk traffic). That's 25 seconds, or a little under half a second of disk traffic per update.

So the very approximate time taken for an update, including the amortized time to eventually write it to the tiles: *Without side files: 6 seconds* With a 40MB batchy side file: 2 seconds *With a 40MB streaming side file: 1 second* With an in-memory batchy 300MB side file: 0.5 seconds

Tile Ordering

So far, we've proceeded on the pessimistic assumption that all the tiles were a whole random seek apart. But, actually, they have to be laid out on disk in some sequence or other, so some of them will actually be sequential with one another. For example, we could lay out most or all of the tiles for a particular group of counters (mostly) sequentially on disk, while tiles for different times are laid out in different parts of the disk.

This doesn't make anything worse than our previous assumptions, but it can make them better. In particular, the 23 tiles in which a single counter can be found throughout a single day will generally be a single 10-megabyte read rather than 23 half-megabyte reads, so will take 1/4 second instead of 1/2 second to read. An entire week will take almost 2 seconds.

This optimization can't help by more than about a factor of 2 over the above design because I picked the tile size to guarantee that we don't spend more than half of our time seeking in the worst case. If you make the tiles a bit smaller, you can improve the results for reading in the direction the tiles are contiguous in, at the expense of

reads and writes in the other direction. For example, if we make our tiles 32 minutes by 512 counters, then they will be an eighth of a megabyte each, and reading the 1440 points for a single counter over a day will require reading 45 tiles totaling 6 contiguous megabytes, or about 0.16s, rather than 0.25. But reading a full column (say, to find out which devices used the most bandwidth over a certain period of time) would then require reading 1200 discontiguous tiles, for 1200 disk seeks (about 10 seconds) and 150 megabytes (about 4 seconds), for a total of about 14 seconds, rather than 600 disk seeks and 300 megabytes, for a total of about 12 seconds. (See below about grouping counters by type to improve this.)

Thinned Data

You probably want to add additional tiles containing subsets of the data --- perhaps a data point for each counter every five minutes, or every half-hour, or every hour, or every day. If you want to graph the performance of a particular network device for an entire month, you probably don't want more than 1000 or 2000 data points, and a data point for every 20 minutes adds up to 2000 data points in a month.

This way, you can have, say, 2016 points of five-minute data for a week, in a sequence of 32 contiguous tiles --- 16 megabytes, or 0.4 seconds of disk traffic --- so that you can generate weekly graphs quickly. Monthly or yearly graphs are an even bigger savings.

The thinned data will be small compared to the full-size dataset, so its size probably isn't that important.

RRDTool erases the full-resolution data after making up thinned versions, so that the database always remains the same size.

Grouping Counters By Type

In the section about tile ordering above, I mentioned that figuring out which devices used the most bandwidth over a certain period of time would require a really unreasonable query time, since it requires reading all the tiles for two particular times --- the beginning and the end of that period. There are about 600 tiles covering each of those times, totaling about 600 megabytes, which takes 15 seconds of disk I/O, plus 1200 disk seeks (10 seconds more), if you need to read them all.

But the reason we have 600 000 counters is that we're recording 6 different counters per device. Most queries, like the one suggested above, probably only touch one or two of the counters --- so if each tile only contains one kind of counter, we can improve this substantially.

If the desired results are available in a single counter type, we only need 200 tiles. Furthermore, if we have a thinned data set that happens to place the ends of the period in question in the same column of tiles, then we only need 100 tiles. This means we only need to read 50 megabytes of data and seek 100 times, so we can do the query in a little over 2 seconds instead of 25.

Queries that need data from every type of counter are very rare.

Delta-Compression

As proposed above, each counter has 64 values in a particular tile,

which total 512 bytes if they're 8 bytes each. But most of the time, we can probably get away with an 8-byte initial value and a sequence of changes from the previous value, each of which will usually be 8, 16, or 32 bits. (0 is probably the most common value.) If they're 16 bits on average, then we could fit about four times as many values into the same half-megabyte block, which means that everything needs a lot less I/O bandwidth and many fewer seeks.

In particular: - we only need 1.2 megabytes per minute of writes, or 1.7 GB per day, or 51 GB per month; - each tile can hold 128 minutes of 2048 counters; - the updates for each timestamp get written to 300 tiles; - the 1440 one-minute samples for a single counter are spread over 12 tiles, which can be read in 0.11 seconds if they're contiguous; - the 300 tiles currently being updated can live in an in-memory side file of 150 megabytes maximum or 75 megabytes steady-state (assuming steady-state is possible); this is assuming we keep things compressed in RAM as well; - writing out those 300 tiles (every 128 minutes) takes 4 seconds of write bandwidth and 3 seconds of seeks, or about 0.05 seconds (amortized) per update, plus the time to write to the disk version of the side file, which may still be bulky. - thinned data will probably comprise a larger fraction of the file, since its deltas will be larger; - reading 100 000 counters at each of two timestamps will probably require reading 50 tiles, which will be 25 megabytes of data, so it should be possible to find out which devices used the most bandwidth over some arbitrary period in under a second of disk time.

Further Redundancy

If you could afford it, then instead of tiling, you could just store the data twice --- once such that all the data for each counter is mostly contiguous on disk (i.e. broken up into chunks mostly bigger than the bandwidth-delay product), and again such that all the data for each timestamp is mostly contiguous on disk. Side files would batch updates as before to allow efficient updates.

This would allow queries that only care about a few timestamps or a few counters to run with very little I/O, and delta-compression of counter data might be feasible for the contiguous-by-counter data; it probably isn't feasible for the contiguous-by-timestamp data.

Network Bandwidth Requirements

If you were doing this monitoring using SNMPv2, you could probably do a bulk-get from each of, say, 10 000 devices, once a minute, and get back a 1000-byte-or-so response from each one. That's 10 megabytes per minute, or 1.3 megabits per second in each direction. You can get that kind of performance from ARCNet, or 1978-era Ethernet, or 802.11b, or old 4Mbps Token Ring cards that cost US$5 on eBay in 1997. I'm assuming you can poll switches or routers or something, one per ten devices, rather than having to talk to all 100 000 network devices directly.

An SNMP library that can handle talking to 10 000 agents within a minute --- perhaps 200 or 300 at any given time, if you have a reasonable timeout --- may be a little more difficult to come by. It's not technically very difficult to do, but you have to design your SNMP library to do it, and not require a separate thread for each concurrent request.

In Summary

It should be inexpensive to log and query several vital statistics of each device on a large corporate network with a single five-year-old laptop, maintaining historical data with hourly granularity indefinitely, using the following techniques.

Batch updates in memory, logging them in a file, flushing them to disk when necessary.

Each datum is addressed by a tuple (countertype, deviceid, timestamp). I recommend storing data in physically contiguous tiles of around half a meg (times or divided by four) addressed by tuples (tilerow, tilecol), where tilerow is a function of countertype and deviceid and tilecol is a function of timestamp, such that data from different countertypes are assigned to different tilerows, and each tilecol corresponds to a contiguous interval of timestamps adjacent to the intervals its neighboring tilecols correspond to. Then, tiles of the same tilerow and consecutive tilecols should be stored consecutively on disk most of the time.

Variable-length delta-compression of the data for a particular counter within each tile should provide very substantial benefits in responsiveness.

Data must be stored redundantly in two ways: - the file of logged updates eventually becomes redundant with the data stored in the tiles; - thinned-out data tiles contain selected timestamps from other tiles to facilitate queries that cover longer time intervals.

Topics

• Programming (p. 3658) (286 notes)
• Performance (p. 3621) (149 notes)
• Databases (p. 3400) (20 notes)

Smoky day

Kragen Javier Sitaker, 2008-04-19 (4 minutes)

(I wrote this 2008-04-19, when Buenos Aires was still under a blanket of heavy smoke.)

I went out in the smoke tonight, Saturday night, to try to get food from Chinatown, despite Beatrice's protestations that 20:00 was too late.

As I slowly walked the few blocks to the route 107 bus stop, three 107 buses passed me. I waited at the bus stop as two 107 buses passed going the other way; while I waited, standing in the street, other would-be passengers accumulated: a bald man with gray hair cuddling and kissing with his middle-aged girlfriend as they stood in the street behind me, and two teenagers.

Eventually I gave up on the bus and hailed a passing taxi, which I took to a Citibank near Chinatown, where I extracted money from my bank account via an ATM.

The sidewalk cafes in the commercial district were full of people, despite the smoke blanketing the city; I recognized an acquaintance waitressing at the the restaurant "1810", where we first tasted Argentine empanadas. A few blocks away, as I walked in the direction of the 107 bus route and Chinatown, I found a long line of mostly old people. I asked a young man standing in line what the line was for. He didn't answer for a moment, and then without meeting my eyes, he explained that it was for bread.

I walked along what I thought was the 107 bus route, but I arrived in Chinatown before seeing any more 107 buses. The store I had hoped to go to had closed at 20:30; I walked around looking for an open store, so I could buy peanut butter, ginger root, and packaged ramen. (Ramen only costs $2 a package there.)

I passed a couple of young men with small shopping carts full to the brim of 1.5-liter Quilmes beer bottles, waiting to be let into an apartment complex; elsewhere I passed one or another sentry waiting at a door, presumably to let in people who had gone out.

After walking about six blocks through almost all of Chinatown, I never found an open grocery store, so I went to Todos Contentos and ordered a couple of dishes to take home to Beatrice.

As I waited, I read some of the sports section of the paper. It had a list of the rugby and football games that had been canceled because of the smoke, although it explained that the air "wasn't toxic", just irritating and allergenic. Maybe "tóxico" means something different in Spanish than in English.

As I carried my order from the restaurant to the 107 bus stop, I stopped by "Dashi", a sushi restaurant near the Buddha Bar. The newspaper blurbs outside the door explained that the chef had spent a long time in Perú and had studied in California, so I hoped that perhaps they might have some of the sushi flavors I've been missing here in Argentina: maguro, uni, natto, unagi, ama-ebi, inari, and so on. I, went in to read the menu. Although it had several pages listing an impressive number of different kinds of sushi, more careful reading revealed that they were made from a small number of basic ingredients that did not include any of the above. I was a little

disappointed but not surprised.

I walked on. A couple sitting on some steps asked me what my mask was for --- I explained it was for the smoke. Wordlessly the man grinned and lifted his cigarette to his lips and took a long drag, filling his lungs with much denser smoke. I laughed.

I eventually caught the 107 home. Strangely, when I got on, the bus was empty.

Topics

Cartesian product storage

Kragen Javier Sitaker, 2017-03-20 (3 minutes)

A common way we organize storage in computer programs is as a Cartesian product of some set of entities and some set of attributes that pertain to those entities. We might represent entities as, for example, struct base addresses or array indices, and attributes as, respectively, offsets into a struct and array base addresses.

In both of these cases, at the machine level, we are representing both entities and items as machine words and combining them with binary addition, possibly composed with a bit shift or even multiplication by a constant. In some cases, you can use a simpler operation, dividing the memory address into one field for entities and another for attributes, which could save you an adder at the hardware level; this allocates memory space of a power of 2 to each entity and each attribute. Combining the fields in hardware can be done with some wires.

What happens if you try to use some other operation to combine the identifiers of entities and attributes? An easy example is to use a limited-width adder: given a 16-bit entity address and an 8-bit struct offset, use a 10-bit adder rather than a 16-bit adder to combine them, and then don't allocate structs that cross 1024-byte page boundaries. This is intermediate both in restrictions and in implementation complexity between the field-division approach and the usual approach.

In the same context, if you use XOR instead of an adder, you have the same memory-allocation restrictions as the field-division approach, but now you have the ability to reverse the order of the list of entities (or of attributes, or of power-of-2-sized blocks of either) by using an address with 1 bits in the field for the other kind of thing.

If you use OR or AND instead of XOR, you have the same restriction again — but now stray bits (ones in the OR case, zeroes in the AND case) allow a sort of "inheritance". Instead of reversing the order of the entities, or of entities in some subgroup (or correspondingly attributes), they allow an attribute to be shared among all entities, or all entities in a power-of-2 group; or, correspondingly, to allow an entity to have the same value for all attributes, or all attributes in a power-of-2 group.

This seems like it could be useful under some circumstances; you could imagine, for example, allocating memory for a potentially buffer-local variable in every buffer, then making the variable buffer-local by changing its "offset".

(The other possible 13 bitwise operations all seem to fall into the same category as one of OR/AND and XOR or be inapplicable. Operations with truth tables 0000 and 1111 discard both inputs, and 0011, 1100, 0101, and 1010 discard one of their inputs. 1000 (NOR) and 1110 (NAND) are simply AND and OR (respectively) with both inputs inverted; 1001 (XNOR) is XOR with one input inverted; 0010 and 0100 (implication and reverse implication) are AND with one input inverted; 1101 and 1011 (negated implication and negated reverse implication) are OR with one input inverted.)

Topics

A REST interface to a software transactional memory

Kragen Javier Sitaker, 2017-06-21 (2 minutes)

Suppose you design a REST system in which every GET or PUT includes a transaction ID, which is a URL, assigned by some kind of transaction serialization service. Any piece of code that wants to make changes to resources must allocate a transaction ID from the transaction serialization service.

Some resources are merely files, while others are synthetic, with their representations produced on demand from other resources. When you GET the URL of a file, the file notes your transaction ID, and if it changes in the future while your transaction is still active, it will post a change notification to your transaction ID. When you PUT a file, the new state is at first only visible to GETs with your transaction ID.

When a transaction is completed, you tell the transaction service to commit your transaction. It first verifies that none of the files you GETted have had changes committed to them by a previously committed transaction; if this is true, it tells all the files you PUTted to commit your change.

(It's unclear to me whether the transaction service has to do all of this dependency tracking or whether the file servers can do some of it.)

If, instead, one or more of the files have changed, then the transaction service refuses to commit your transaction, instead rolling it back. Then you can retry the whole transaction from the beginning, or give up and report failure.

Topics

Arduino safety

Kragen Javier Sitaker, 2018-12-10 (4 minutes)

I just watched a video on the YouTube channel "GreatScott" in which the author builds a modified-square-wave inverter using a MOSFET H-bridge driven through a couple of TC4427s.

I was somewhat horrified at the way he wrote the software; he writes `off()`, `lowon()`, and `highon()` functions which each set the transistors to the appropriate states. If you call the `off()` function often enough, everything is fine, but if you call `highon()` after calling `lowon()` without an intervening `off()`, you get shoot-through; the first thing it does is bring Arduino pin 9 LOW, which pulls pin 1, the gate of the P-channel MOSFET on the high side of the left of the H-bridge, to ground, turning on the MOSFET. But Arduino pin 8 is still HIGH, pulling pin 3, the gate of the N-channel MOSFET to ground on the left side of the H-bridge, to 12V, also turning it on. So you get shoot-through and a potentially damaging current spike through your MOSFETs. Similarly, you get shoot-through on the right side if you call `lowon()` after calling `highon()` without an intervening `off()`, because it brings pin 10 LOW, turning on the P-channel MOSFET on the high side of the right side of the H-bridge, before bringing pin 7 LOW, turning off the N-channel MOSFET on the low side of the right side.

So it is very important to make sure that you don't call these functions without calling `off()` in between. Furthermore, you would like to ensure that enough time elapses in between the call to `off()` and the subsequent function for the high-side MOSFETs to be entirely turned off, since in general they have a small RC delay, though in this case it's probably less than a nanosecond. (And for this reason, just rewriting the `lowon()` and `highon()` functions to turn things off before turning other things on is not necessarily adequate, though it would probably be a good idea.)

You'd like to be able to encapsulate this timing safety logic in some kind of tiny capsule of code that is small enough to be sure that it says what you think it does, then validate that it's called correctly at compile-time, since after you've already plugged your refrigerator into your homemade inverter, it's too late to report a timing constraints violation.

You could make this almost work in Arduino's dialect of C++ by adding a little bit of state and checking the current time when these functions are called. But what do you do when it's too early? You could delay until the appropriate time, but that hangs the whole system, which is really unfortunate if it's doing something else like updating an LCD. Or you could just return without doing anything, which would result in missing an output pulse in this case, which might cause real problems.

A slightly less shitty solution would be to allocate a timer to MOSFET control (as GreatScott does) and use it to delay the state transition until a later time, using an ISR. This approach doesn't block everything else on the device, but it does use up a timer; generalizing this to share a timer between multiple users essentially produces a real-time operating system. But it doesn't guarantee you

get the timing you asked for, just that the timing you get doesn't cause shoot-through and explode your MOSFETs.

To really encapsulate the timing constraint so that it's not spread across your entire codebase, you probably need some kind of theorem prover that can prove things about function execution times.

Topics

Charge transfer servo

Kragen Javier Sitaker, 2013-05-17 (2 minutes)

I just tore down a broken DVD player I found on the street. It has three motors in it, two of which are geared to some kind of linear-motion slide; but all three of them seem to be just DC motors with no builtin servo.

Accurately positioning a linear-motion slide is a big part of what you need for various kinds of machinery, so I immediately got to thinking about how you could do it.

Aside from the most obvious things (the optical tape sensor from a printer, using a webcam, using the inverse square law with LED and photodiode) it occurred to me that the plastic slide itself probably has a relative permittivity of two or three, and so if you could arrange metal plates fixed in place on each side of it, you'd have a variable capacitor. Better, you could put one metal plate on its surface and another one fixed in place above or below it, with a layer of plastic between them keeping them from touching as they slide against each other. This would give you a capacitance varying from the femtofarad range to the 1000-picofarad range with the motion of the slide.

Charge-transfer sensing supposedly allows inexpensive measurement of variable capacitances to precisions of fractions of a femtofarad. If that's really true in this case, it should allow the inexpensive measurement of such a variable capacitor to some six significant figures, or 20 bits, of precision. If this were to be fully transferred to the positional precision of the slide — that is, if nothing else affected the capacitance than the position of the slide — then we would be capable of measuring the 13-centimeter motion of the slide to within 130 nanometers, about 1300 atoms. Sufficiently intelligent motor control, then, might be capable of *positioning* the slide to within 130 nanometers, although due to the many sources of slack and vibration in the system, surely not arbitrary motion paths with such precision.

Topics

Complex linear regression (in the field ℂ of complex numbers)

Kragen Javier Sitaker, 2019-08-17 (updated 2019-08-18) (9 minutes)

In $1 recognizer diagrams (p. 1264) I speculated that applying linear regression to vectors of complex numbers might be a good way to match user interface gestures such as strokes to templates, since a linear function in ℂ amounts to a translation, rotation, and scaling. But I can't find any discussion of doing linear regression on complex numbers.

So I'm going to solve that problem here.

The stroke-matching problem

Suppose we have a sequence of points representing a stroke the user has drawn with their pen or a finger on a pen computer — whether something like the Fly Pen, something like the HP Itsy, something like a Wacom graphics tablet, or something like a modern Android computer. We want to determine how far this sequence is from being "the same as" a stored template representing, say, the letter "B", so we can see which of several stored templates it's closest to and thus determine the user's intent.

Let's suppose the sequence of points has already been resampled to regular spatial intervals, with the same number of points as the template (128 or something), so pairing up corresponding points is trivial.

The simplest thing that could possibly work is to take the sum of absolute differences of coordinates between corresponding points: $\Sigma_i |x_{ti} - x_{si}| + |y_{ti} - y_{si}|$, where i ranges over the number of points, x_t and y_t are the coordinates of the template points, and x_s and y_s are the coordinates of the stroke points. This loss function tells us how far the stroke is from being a "B" or whatever.

This may work under some circumstances, but it has some problems:

• The user may have written the stroke in a different *position* than the template, introducing *translational error* into all of the components of the loss function.
• The user may have written the stroke in a different *orientation*, introducing *rotational error* into most of the components of the loss function, except those points in the center.
• The user may have written the stroke at a different *size*, introducing *scaling error* into most of the components of the loss function, again, except for those in the center. Moreover the scaling may be nonuniform — the user may have stretched the stroke horizontally, vertically, or diagonally without intending to do so — introducing *skewing error*.
• Using the absolute difference means that an error of, say, 5 pixels diagonally, will count as 7 pixels; this introduces a *anisotropic bias* into the loss function which makes it harder to cleanly separate, for example, Bs from non-Bs.
• Using the absolute difference means that an error of 20 pixels counts

as just four times a 5-pixel error. But, generally, if the user really did intend to draw a B, four 5-pixel errors are much more probable than a single 20-pixel error. This, similarly, makes it harder to cleanly separate Bs from non-Bs. Using the absolute difference represents an implied probability distribution of errors that is exponential.

The "$1 recognizer" paper I was writing about has some solutions to these: translate the stroke so its centroid is at the origin to eliminate translational error, rotate it so that the start point is at a fixed angle to eliminate rotational error, rescale it nonuniformly horizontally and vertically so that its bounding box has size 1 in each dimension to eliminate scaling and skewing error, and use the sum of Euclidean distances rather than the sum of absolute differences to eliminate anisotropic bias. (They still have problem #5, unless I'm imagining things and it isn't really a problem.) As I wrote in $1 recognizer diagrams (p. 1264), this procedure seems unduly sensitive to noise.

But it occurred to me that this was very similar to the problem of linear regression, only with complex numbers. If we represent each point (x, y) as a complex number $(x + jy)$, then rotation and *uniform* scaling around the origin are merely multiplication by a complex number, and translation is merely adding a complex number. So if the stroke is precisely a translated, uniformly scaled rotation of the template, then there exist some complex numbers m and b such that $\forall i$: $x_{si} + jy_{si} = m(x_{ti} + jy_{ti}) + b$. Let's abbreviate the stroke point i as $s_i = x_{si} + jy_{si}$ and the template point i as $t_i = x_{ti} + jy_{ti}$, so that this becomes just $\forall i$: $s_i = mt_i + b$. And if the points of the stroke are perturbed slightly from those positions, then there exist complex numbers m and b that give a small sum $\Sigma_i |mt_i + b - s_i|^2$ (the L^2 norm of the vector $m\vec{t} + \vec{b} - \vec{s}$, where all the components of vector $\vec{b}$ are equal); that residual sum tells us how much error there is in the approximation, and by finding m and b to minimize that sum, we can find something that is in some sense the best fit.

This is precisely the everyday problem of linear regression, but in the field $\mathbb{C}$ of complex numbers, rather than the usual field $\mathbb{R}$ of real numbers. The squared modulus solves problems #4 (anisotropy) and #5 (nonuniform weighting), implying a Gaussian distribution of errors, which is probably a reasonable approximation even if not precisely correct; and, at least in $\mathbb{R}$, it makes the optimization problem easy by making it convex and differentiable in closed form.

If we can find the ideal rotation, scaling, and translation in this way, we can conceivably find a better fit than the one the "$1 recognizer" finds, and maybe we can find it more efficiently, too, especially if we can calculate it in closed form rather than using golden-section search to heuristically approximate the optimal rotation. We'd still need to stretch the stroke up front to correct for nonuniform scaling (skewing error), perhaps by calculating the x–y covariance matrix of its points.

This is easier than the more general problem of trying to match up two sets of possibly rotated, translated, and scaled points, such as star tracking, because the correspondence between the points is provided up front — it comes from the temporal sequence of points in the stroke.

Existing work on complex linear regression

Abdul Ghapor Hussin, Norli Anida Abdullah and Ibrahim Mohamed wrote a 2010 paper called "A Complex Regression Model" about using linear regression with complex variables to predict "circular variables" such as the direction from which waves are hitting a buoy. I don't really understand their derivation.

Math.stackexchange.com users Naetmul and hans have written up the general solution for finding a least-squares-optimal approximate solution to the linear system $\mathbf{A}\vec{x} = \vec{b}$ in complex numbers. I don't think I can transform the linear regression problem into this problem, because the output has too many degrees of freedom — I'm looking for a pair (m, b) regardless of how many input points there are, while that problem takes the matrix $\mathbf{A}$ and the vector $\vec{b}$ as given, then tries to find the vector $\vec{x}$ that gets you closest to it in the subspace defined by $\mathbf{A}$. This amounts to projecting $\vec{b}$ onto that subspace and then figuring out where you are in it.

Aha! John Cowan referred me to whuber's answer on the Cross Validated Stack Exchange, where they explain that the answer is

$\beta\hat{} = (X^*X)^{-1}X^*z$

along with R code to implement it and everything. Not sure it's rigorously proven to be the correct answer, but there are "appears to work" arguments.

Finding the least-squares solution to the univariate complex linear regression problem

So we want to find $\operatorname{argmin}_{m,\, b \in \mathbb{C}^2} \Sigma_i \, |mt_i + b - s_i|^2$. Although the complex modulus $|\cdot|$ isn't differentiable everywhere, the modulus squared $|\cdot|^2$ is, and therefore so is the whole sum. So we should be able to find all of its extrema by finding where its partial derivatives with respect to m and b are zero. And, if the situation is like the situation in $\mathbb{R}$, the derivative will only have a single zero, and it will be the minimum; intuitively, a similar situation ought to obtain here.

Let's define $\Delta_i = mt_i + b - s_i$, so we're trying to minimize $\Sigma_i \, |\Delta_i|^2$, so we set $\nabla[\Sigma_i \, |\Delta_i|^2] = 0$, so $\Sigma_i \, \nabla |\Delta_i|^2 = 0$, which is to say that $\Sigma_i \, \partial|\Delta_i|^2/\partial m = 0$ and $\Sigma_i \, \partial|\Delta_i|^2/\partial b = 0$, which are two problems we can solve separately. $\partial|\Delta_i|^2/\partial m = (\mathrm{d}|\Delta_i|^2/\mathrm{d}\Delta_i)(\partial\Delta_i/\partial m)$, and similarly for b, $\partial|\Delta_i|^2/\partial b = (\mathrm{d}|\Delta_i|^2/\mathrm{d}\Delta_i)(\partial\Delta_i/\partial b)$.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Mathematical optimization (p. 3611) (29 notes)
- Hand computers (p. 3492) (10 notes)
- Linear algebra (p. 3551) (4 notes)
- Gestures (p. 3471) (2 notes)
- Statistics
- Regression

Quasiquote patterns

Kragen Javier Sitaker, 2007 to 2009 (9 minutes)

[Note added 2019: Darius Bacon is working on a pattern-matching dialect of Scheme called Squeam, and I may have gotten these ideas from talking with him about it. It's been long enough that I can't remember for sure.]

It occurred to me that if you wanted to write a tree-rewriting dialect of Lisp, you could use quasiquotation both for the pattern to match and the thing to rewrite it to. For example, in David Andrew Kranz's 1988 (1990?) dissertation, "Orbit: An Optimizing Compiler for Scheme," he describes much of the CPS-conversion process in tree-rewriting rules such as the following:

```
(convert <atom> <cont>) => `(,<cont> ,<atom>)
(convert '(<proc> . <arguments>) <cont>) =>
(convert <proc>
    (lambda (p)
        ,(convert-arguments <arguments>
                        `(,p ,<cont>))))
(convert-arguments '() <final-call>) => <final-call>
(convert-arguments '(argument . rest) <final-call>) =>
    (convert <argument>
    `(lambda (k)
     ,(convert-arguments <rest>
                (append <final-call> '(k)))))
```

It seems to me that these could be written in a pattern-matching Scheme dialect as follows, without breaking backwards-compatibility:

```
(def (convert atom cont) `(,cont ,atom))
(def (convert (proc . arguments) cont)
    (convert proc (lambda (p) ,(convert-arguments arguments
                          `(,p ,cont)))))
(def (convert-arguments () final-call) final-call)
(def (convert-arguments (argument . rest) final-call)
    (convert argument
    `(lambda (k) ,(convert-arguments rest (append final-call '(k))))))
```

(I'm using "def" as a synonym for "define" here. Or maybe a pattern-matching extension thereof.)

I suspect that the original syntax was not formally specified.

Here are some more things specified in this style.

```
(def (length ()) 0)
(def (length (a . b) (+ 1 (length b))))
(def (append () a) a)
(def (append (a . b) c) `(,a . ,(append b c)))
(def (assv a ()) '())
(def (assv a ((a . b) . r)) `(,a . ,b))
(def (assv a ((x . y) . z)) (assv a z))
(def (delv a ()) '())
```

```
(def (delv a (a . b)) (delv a b))
(def (delv a (b . c)) `(,b . ,(delv a c)))
(def (eval ('var var) env) (cadr (assv var env)))
(def (eval ('lambda args body) env) `(,args ,body ,env))
(def (eval ('apply (func-args body func-env) args) env)
    (eval body (augment-env func-args (eval-args args env) func-env)))
(def (eval ('if cond cons alt) env)
    (if (is-true (eval cond env)) (eval cons env) (eval alt env)))
```

Note the implicit pattern-matching equality tests in assv and delv.

This syntax has the disadvantage that the symbol "quote" is treated specially, in order to support tagged things like the last bit. I had considered using full quasiquote syntax, which avoids that problem, but it has two disadvantages: it's backwards-incompatible with Scheme's current syntax, and worse, it's more verbose.

(It also lacks a way of requiring, in the pattern, that a particular item be just a plain symbol, rather than a list or an array or something. You could specify, say, "unquote" or "symbol" to mean this, and write (symbol x) or ,x to mean "some x, which is required to be a symbol".)

If you add special treatment for the symbol "..." to the declaration syntax, as in the Scheme macro system, then you get an arguably superior syntax for list tails with a special secret feature.

```
(def (append () x) x)
(def (append (x xs ...) y) `(x . ,(append xs y)))
(def (assv a ()) '())
(def (assv x ((x xs ...) zs ...)) `(x . xs))
(def (assv x ((y ys ...) zs ...)) (assv x zs))
(def (delv x ()) '())
(def (delv x (x xs ...)) (delv x xs))
(def (delv x (y ys ...)) `(,x . ,(delv x ys)))
```

The special secret feature is that it generalizes to some kinds of restructuring:

```
(def (augment-env () () env) env)
(def (augment-env (var vars ...) (val vals ...) env)
 (augment-env vars vals `((,var . ,val) . ,env)))
```

(The above definitions of augment-env, assv, and eval constitute a complete interpreter for a Turing-complete language.)

Pattern-matching simplifies writing compilers in general. Here's a compiler into the language above from a subset of Scheme:

```
(def (compile ('lambda vars body)) `(lambda ,vars ,(compile body))))
(def (compile ('if cond conseq alt)) `(if ,cond ,conseq ,alt)))
(def (compile (f args ...))
 `(apply ,(compile f) ,(compile-args args)))
(def (compile-args ()) '())
(def (compile-args (arg args ...))
 `(,(compile arg) . ,(compile-args args)))
(def (compile x) `(var ,x))  ; or (def (compile (symbol x)) `(var ,x))
```

Here's a little recursive-descent compiler for the following

grammar. (Our input strings here are Lisp lists.)

```
mul-expr  ::= atom-expr * mul-expr | atom-expr / mul-expr | atom-expr
atom-expr ::= <number> | <symbol> | (expr)
expr      ::= mul-expr + expr | mul-expr - expr | mul-expr
```

Each parse routine takes a suffix of the whole input string, and returns a list of two lists: the compiled version of what it parsed, and the remaining suffix.

```
(def (compose-parser inner outer tokens)
 ((lambda ((inner-result leftovers))
    `(,(outer inner-result) ,leftovers))
  (inner tokens)))
(def (parse-atom ((expr) stuff ...)) `(,(parse expr) ,stuff))
(def (parse-atom (x stuff ...))
 `(,(if (symbol? x) `(var ,x) `(const ,x)) ,stuff))
(def (parse-expr stuff) (parse-expr-tail (parse-mul-expr stuff)))
(def (parse-expr-tail (compiled ('+ rest ...)))
 (op-node '+ compiled rest parse-expr))
(def (parse-expr-tail (compiled ('- rest ...)))
 (op-node '- compiled rest parse-expr))
(def (op-node op compiled rest inner)
 (compose-parser inner
        (lambda (parsed-rest) `(,op ,compiled ,parsed-rest))
        rest))
(def (parse-expr-tail (compiled other)) `(,compiled ,other))
(def (parse-mul-expr stuff) (parse-mul-expr-tail (parse-atom stuff)))
(def (parse-mul-expr-tail (compiled ('* rest ...)))
 (op-node '* compiled rest parse-mul-expr))
(def (parse-mul-expr-tail (compiled ('/ rest ...)))
 (op-node '/ compiled rest parse-mul-expr))
(def (parse-mul-expr-tail (compiled other) `(,compiled ,other)))
```

That's pretty ugly! And it doesn't even quite compile into RPN; you still need this:

```
(def (rpnify ('+ a b)) (rpnify-op '+ a b))
(def (rpnify ('- a b)) (rpnify-op '- a b))
(def (rpnify ('* a b)) (rpnify-op '* a b))
(def (rpnify ('/ a b)) (rpnify-op '/ a b))
(def (rpnify-op op a b) (append (rpnify a) (append (rpnify b) `(,op))))
(def (rpnify ('var x)) `(var ,x))
(def (rpnify ('const x)) `(const ,x))
```

That's kind of ugly in its runtime; a cleverer implementation would walk the op tree from right to left, passing along an accumulator of ops as it went:

```
(def (walk ('+ a b) ac) (walk-op '+ a b ac))
(def (walk ('- a b) ac) (walk-op '- a b ac))
(def (walk ('* a b) ac) (walk-op '* a b ac))
(def (walk ('/ a b) ac) (walk-op '/ a b ac))
(def (walk-op op a b ac) (walk a (walk b `(,op . ,ac))))
(def (walk ('var x) ac) `((var ,x) . ,ac))
```

```
(def (walk ('const x) ac) `((const ,x) . ,ac))
```

I should probably write a version that does that in a single pass. But first! Something to run the pattern-matching.
(match pattern actual bindings).

```
(def (match () () vars) vars)
(def (match () other vars) #f)
(def (match ('quote x) x vars) vars)
(def (match ('quote x) other vars) #f)
(def (match ('symbol x) (symbol y) vars) (match x y vars))
(def (match ('symbol x) other vars) #f)
(def (match (ap . bp) (a . b) vars)
 (let ((new-vars (match ap a vars)))
      (if new-vars (match bp b new-vars))))
(def (match (ap . bp) other vars) #f)
(def (match (symbol x) y vars) `((,x . ,y) . ,vars))
```

That leaves out the equality-testing logic --- in both the ('symbol x) and the (symbol x) cases, we need to verify that there's no existing conflicting binding for x.

Recasting that (mostly) in a form that doesn't rely on pattern-matching:

```
(define (match pattern actual vars)
  (cond ((null? pattern) (match-null actual vars))
    ((symbol? pattern) (match-symbol pattern actual vars))
    ((pair? pattern) (match-pair pattern actual vars))
    (else (error "invalid pattern"))))
(define (match-null actual vars) (if (null? actual) vars #f))
(define (match-symbol pattern actual vars)
  `((,pattern . ,actual) . ,vars)) ; XXX doesn't check for dups
(define (match-pair pattern actual vars)
  (cond ((eq? (car pattern) 'quote)
     (and (equal? (cadr pattern) actual) vars))
    ((eq? (car pattern) 'symbol)
     (and (symbol? (cadr pattern))
      (match (cadr pattern) actual vars)))
    (else
     (let ((new-vars (match (car pattern) (car actual) vars)))
       (if new-vars (match (cdr pattern) (cdr actual) new-vars) #f)))))
```

And here's a fix to match-symbol that does equality testing:

```
(define (match-symbol pattern actual vars)
    (let ((existing (assq pattern vars)))
  (cond ((not existing) `((,pattern . ,actual) . ,vars))
    ((equal? (cdr existing) actual) vars)
    (else #f))))
```

From there, it's a simple extension to supporting multiple patterns:

```
(define (match-many patterns functions actual)
  (if (null? patterns) (error "match failure")
(let ((bindings (match (car patterns) actual '())))
```

```
  (if bindings ((car functions) bindings)
    (match-many (cdr patterns) (cdr functions) actual)))))
```

Ideally we'd really like to compile the pattern-matching down to efficient Scheme code which does a reasonably small number of tests and doesn't re-examine the pattern as well as the actual, each time around. I don't think I'm quite up for that yet.

Unsurprisingly, this is not a new idea; Andrew K. Wright's 1996 proposal for pattern-matching in Scheme uses a slight superset of this pattern syntax, with the same semantics. (I switched to using equal? instead of eqv? in the above code after reading his paper.) However, I think it's interesting to allow definition of new cases on existing functions this way, rather than shoving it off into a "match-define" ghetto as Wright does.

Topics

- Lisp (p. 3552) (9 notes)
- Pattern matching

Cheap frequency detection

Kragen Javier Sitaker, 2017-06-29 (updated 2019-06-19) (50 minutes)

Most audio and radio signals are passed through some unknown linear time-invariant system, such as linear circuits or multipath fading, before we get them. So it's often to our advantage to try to detect features of these signals that survive such mangling. Those features are sinusoids, or, more generally, complex exponentials.

If you want to detect sinusoidal components in a signal, the standard approach is to use the Fourier transform. This has a couple of disadvantages:

• It requires a substantial amount of computation per sample, specifically (for the radix-2 FFT) 2 lg N real multiplications where N is the window size and lg is the base-2 logarithm;
• When a signal is only present during part of the window, its energy is smeared across the whole window, which may remove aspects of interest (you may want to know when it happened) and may push it below the noise floor;
• It more or less requires floating-point computation, which increases the number of bit operations substantially.

There are a number of other approaches which can be used in cases where the Fourier transform's disadvantages are fatal, but I haven't found a good overview of them. Here I'm going to talk about counting zero crossings, SPLLs, CIC decimation, the Minsky circle algorithm, the Goertzel algorithm, Karplus-Strong delay line filtering, linear predictive coding, and uh this thing I just came up with in the shower, and finally an extended zero-crossing approach.

Counting zero crossings

A very simple and commonly used nonlinear filtering or demodulation approach is to count the number of times the signal crosses the X-axis.

If the majority of your signal power is in a single frequency (perhaps because you've already filtered it) the count of zero crossings is a reasonable approximation of the frequency. A step up from this is to measure the time interval of the last N zero crossings, maybe the last 16 crossings or the last 64 crossings, in order to get an estimate whose precision is limited by your sampling rate rather than by both your sampling rate and your time window.

The simplest approach is just this, assuming 8-bit samples being returned from `getchar()` (unsigned, as is `getchar()`'s wont):

```
volatile int crossings = 0;
...
for (;;) {
    while (getchar() < 128);
    crossings++;
    while (getchar() >= 128);
    crossings++;
}
```

However, this allows any arbitrarily small amount of noise near a slow zero crossing to add extra zero crossings. Normally in circuits we deal with this by adding a Schmitt trigger, as described in Otto Schmitt's 1937 paper, "A Thermionic Trigger"; as Schmitt says, "in another application, [the thermionic trigger circuit] acts as a frequency meter more linear than one of the thyratron type, and one immune to locking," whatever that means. The Schmitt trigger adds a little bit of hysteresis through positive feedback, and indeed roughly any kind of positive feedback on digital inputs nowadays is called a "Schmitt trigger".

In code, adding this hysteresis is even simpler:

```
volatile int crossings = 0;
...
for (;;) {
    while (getchar() < 128+16);
    crossings++;
    while (getchar() >= 128-16);
    crossings++;
}
```

This requires the noise to move the needle by at least 32 counts before it qualifies as a "crossing", which you would think is 18 dB below full-scale, but of course it's not an average over the whole signal, but an increment at that point. Impulsive noise can overcome this while totaling well below -18 dB by being at -∞ dB in between impulses.

The other problem is that unless the actual signal is substantially higher than those -18 dB, this will miss crossings, maybe all of them. You can adjust the hysteresis to compensate with, for example, a peak detector (note I haven't tried this):

```
volatile int crossings = 0;
...
int threshold = 0, x;
for (;;) {
    int peak = 0;
    while ((x = getchar()) < 128 + threshold) {
        if (128 - x > peak) peak = 128 - x;
    }
    crossings++;
    threshold = peak >> 3;

    peak = 0;
    while ((x = getchar()) > 128 - threshold) {
        if (x - 128 > peak) peak = x - 128;
    }
    crossings++;
    threshold = peak >> 3;
}
```

This may count a few extra crossings at the beginning of the process until it warms up; from then on, it notes the peak of each half-cycle to use to adjust the hysteresis threshold for the next crossing. If the signal you're detecting drops suddenly (by a factor of

8, as specified by `>> 3`) within a single cycle, it could suddenly stop counting cycles; similarly, if there is a noise spike that exceeds the signal by a factor of 8, it could suddenly stop being able to detect the signal.

Per sample, this requires two subtractions, two comparisons, a conditional assignment, and a conditional jump, plus a couple more operations after each zero crossing.

A more robust version of this algorithm would use perhaps the median peak, the median, or some percentile over the last several cycles, rather than simply the peak, and would decay the threshold toward 0 over time in order to recover from the kinds of situations described above. This requires a few more operations.

For frequency detection of a signal, the zero-crossing approach may be superior to the linear FFT-based approaches discussed below. For example, if you want to control a synthesizer using your voice, you need to be able to discriminate at least between musical notes. If you want to distinguish between 110 Hz (A) and 106 Hz (closer to G#) with an FFT you need your frequency bins to be 4 Hz apart, which means you need to be running the FFT over a 250 ms window. But at 48 ksps, the zero crossings of a 110-Hz square wave are 436 samples apart, while the zero crossings on a 106-Hz square wave are more like 453 samples apart, and it seems like you could probably use the median of the last 4–8 intervals between zero crossings. 8 half-waves at 106 Hz would be 38 ms.

However, I suspect that all of these variants are strictly inferior to using a phase-locked loop, which is almost exactly the same amount of computation but has truly impressive noise immunity, and can additionally tell you the power of the frequency it's detecting (at a little extra computational expense.)

Software phase-locked loops

A phase-locked loop is a nonlinear filter that measures the frequency, phase, and possibly power of a signal with constant or slowly changing frequency. You use a phase detector to set the frequency of a local oscillator to match the frequency of the signal, and the phase detector uses that local oscillator as its reference. The simplest phase detector is just a chopper and a low-pass filter; here's a software implementation in one line of C, suitable for tone frequency tracking:

```
/* A PLL in one line of C. arecord | ./tinypll | aplay */
main(a,b){for(;;)putchar(b+=16+(a+=(b&256?1:-1)*getchar()-a/512)/1024);}
```

This is potentially quite efficient, taking almost exactly the same amount of computation as counting zero crossings; at its base, it involves four additions or subtractions per sample, plus a bit test, a conditional, and a couple of bitshifts.

This takes a signal (by default at 8 ksps) from the ALSA audio driver with one sample on `getchar()`; `b` is the current phase of a local oscillator with period of 512 counts, whose free-running frequency is one cycle per 512/16 = 32 samples, so 250 Hz. The phase detector accumulates its error signal in `a`. The input sample is chopped by the ≈250Hz square wave from `b`, then either added to or subtracted from `a`, which has an exponential low-pass filter applied to it by way of

subtracting a 512th of itself, yielding a time constant (I think?) of 512 samples or 64 ms. The range of `a` extends up to where a == a+255-a/512, which is to say when a == 512*255 == 130560; the lower limit is analogously -130560. So `a` is scaled down by dividing by 1024 to get `b`'s free-running increment, which means that in theory it could cause `b` to run backwards or at up to 130+16 = 146 counts per sample, thus about 3.5 samples per cycle or 2280 Hz. (Actually, not even that much, because half the time you have to be feeding it zeroes, so its maximum stable magnitude oscillates around 130560/2.)

(In practice I've never managed to get the above code even up into the kHz range.)

`putchar()` then outputs the low 8 bits of `b`, forming a sawtooth wave from it with twice the chopper frequency, around 500 Hz.

More conventionally formatted and without the uninitialized-read undefined behavior, the above code reads as follows:

```
int main()
{
    int a = 0, b = 0;
    for (;;) {
        a += (b & 256 ? getchar() : -getchar());
        a -= a/512;
        b += 16;
        b += a/1024;
        putchar(b);
    }
}
```

This kind of phase detector tries hard to keep the chopper in quadrature with the detected signal. If you want to know the power of the detected signal, you can chop the input signal with a second chopper in quadrature with the first and sum its (squared) output.

Note that this kind of phase detector is optimized for detecting square waves, and thus can lock onto odd harmonics of the signal it thinks it's detecting. This may be an advantage or a disadvantage in a particular application.

The 1/512 in the above code, which is a low-pass filter on the phase detector output (and oscillator frequency input), directly limits how fast the PLL can track a changing frequency; less apparent is that it also limits how much noise immunity the PLL has, and how far the PLL's frequency can jump to achieve lock (the "capture range"). The proportionality factor by which the phase output adjusts the local oscillator frequency (the 1/1024) limits how far the PLL can track before losing lock (the "lock range"). A couple of hacks to improve these tradeoffs are to sweep the natural frequency of the LO when it hasn't yet achieved lock, to use several concurrent PLLs with different natural frequencies, and to tighten the low-pass filter once lock is achieved.

With a different kind of phase detector, a PLL is also useful for things like beat detection and beat matching, which is in a sense its primary use in hardware — generating clock signals with a predetermined relationship to existing clock signals, including clock and data recovery for asynchronous data transmission.

The one-line program is about the same length in machine code as

in C. An amd64 (but LP64) assembly listing for the 63 bytes of this loop is as follows, keeping `b`, the local oscillator state, in %ebp, and `a`, the phase detector, in %ebx. GCC -Os did not optimize the multiplication and divisions into a conditional and bitshifts as you might expect:

```
40                  .L3:
43 0011 89E8            movl    %ebp, %eax
44 0013 25000100        andl    $256, %eax
44      00
45 0018 83F801          cmpl    $1, %eax
46 001b 4519ED          sbbl    %r13d, %r13d
47 001e 31C0            xorl    %eax, %eax
48 0020 4183CD01        orl $1, %r13d
49 0024 E8000000        call    getchar
49      00
51 0029 4489E9          movl    %r13d, %ecx
52 002c 0FAFC8          imull   %eax, %ecx
53 002f 89D8            movl    %ebx, %eax
54 0031 99              cltd
55 0032 41F7FC          idivl   %r12d
56 0035 29C1            subl    %eax, %ecx
57 0037 01CB            addl    %ecx, %ebx
59 0039 B9000400        movl    $1024, %ecx
59      00
60 003e 89D8            movl    %ebx, %eax
61 0040 99              cltd
62 0041 F7F9            idivl   %ecx
63 0043 8D6C0510        leal    16(%rbp,%rax), %ebp
65 0047 89EF            movl    %ebp, %edi
66 0049 E8000000        call    putchar
66      00
69 004e EBC1            jmp .L3
```

There are many ways to improve this simplified PLL.

A simple one is to use a simple-moving-average filter instead of an exponential filter to smooth the phase detector; this gives you better tracking of frequency changes for the same noise immunity, or better noise immunity for the same tracking of frequency changes.

Different phase detectors are best for different kinds of signals. For sinusoidal signals with no significant harmonics, like a person whistling or an FM radio signal, you can get a better phase detector by weighting samples toward the middle of the sample interval more highly than samples toward its edges; multiplying by even a triangle wave (a square wave convolved with a simple moving average) gets you most of the way there. For signals where only, say, leading edges are significant, you can use a phase detector that only considers the few samples near that leading edge.

The shower algorithm

Let's suppose you want to detect some fixed frequency f and you've already resampled your signal to a sampling rate of 4f. Two orthogonal sinusoids at frequency f then are [-1, -1, 1, 1] and [1, -1, -1, 1]; two others are [0, 1, 0, -1] and [1, 0, -1, 0]. If you can find the dot

product of your signal with a pair of these sinusoids, then you can use their Pythagorean sum to precisely compute the energy in that frequency component of the signal.

If you're dumping the decimated samples into a four-sample circular buffer x0, x1, x[2], x[3], with some incrementing pointer xp:

```
x[xp++ & 3] = new_sample;
```

Then you could imagine using x[xp], x[xp-1], x[xp-2], and x[xp-3] with the appropriate modulo math. However, this is totally not necessary, because you actually don't care how these sinusoids are aligned with the signal; you only care that they are orthogonal. It's totally valid to compute one phase component as x0 - x[2] and the other as x1 - x[3] and then compute their Pythagorean sum:

```
return pythsum(x[0] - x[2], x[1] - x[3]);
```

So far, though, we've gotten away without requiring the potentially large number of bit operations required to even square a number. For low-precision data types, like 8 bits, it would be totally valid to use a lookup table of squares, then binary-search it to find the square root. However, there are more approximate alternatives that may be good enough in many cases.

Specifically, max(|a|, |b|) and |a| + |b| are both approximations of the Pythagorean sum that are never wrong by more than a factor of √2 and can be computed in a small linear number of bit operations. Both are precisely accurate when a or b is 0 and have their worst case when |a| == |b|; max(|a|, |b|) is low by a factor of √2 then, and |a|+|b| is high by a factor of √2. Both have level sets that are square, but rotated 45°. If we sum them, the resulting octagonal level sets have a worst-case error of a bit under 12%, just under 1 dB.

This Pythagorean-sum-approximation algorithm looks like this:

```
if (a < 0) a = -a;
if (b < 0) b = -b;
return ((a < b ? b : a) + a + b >> 1);
```

YMMV. On something like an AVR ATMega, the necessary three comparisons, three conditional branches, pair of additions, and right shift are probably slower than just computing the Pythagorean sum with the two-cycle multiplier. In the other direction, the 3 dB worst-case error of max(|a|, |b|) or |a| + |b| is insignificant in many frequency-detection contexts, where you're trying to determine whether a signal is more like -15dB or more like your -50dB noise floor.

In many cases you might want to be integrating the wave over a significant period of time. In a case like that, there are a few different cheap approaches you can take. A circular buffer of four simple-moving-average filters provides optimal noise immunity for a given step response; you can do the same thing with the accumulators for single-pole exponential filters; and a chain of two or three moving-average filters inexpensively gives you something approaching a Gaussian window. Finally, you could simply use a buffer of four accumulators to sum the corresponding samples across

some rectangular window, without attempting any kind of nonuniform weighting.

The problem of frequency response in rectangular windows

The disadvantage of rectangular windows in general (whether simple moving average filters or fixed windows of a few hundred samples or whatever) is that their Fourier transform is sinc. Implicitly multiplying your signal by the rectangular function in this way effectively convolves its frequency spectrum with sinc, which dies off relatively slowly (1/n) as you get far away from its center. Even a simple triangular window, which can be achieved by convolving two identical rectangular functions together (and thus squaring their frequency response), dies off at a much more reasonable pace ($1/n^2$). And, in a sense, this is the basis for CIC decimation.

CIC decimation

CIC decimation does not itself detect signals; it just (linearly, with linear phase, essentially with a convolution of simple moving averages) low-pass filters them and reduces the sampling rate. However, an Nth-order CIC decimation filter involves only N integer additions (using N accumulators) per input sample plus N subtractions per output sample (using N previous output samples). As explained above, though, there are very efficient ways to detect signals in such decimated data.

Here's a second-order CIC decimation filter (untested) that reduces the sample rate by a factor of 73. It uses unsigned math because in C unsigned overflow is well-defined and guarantees the property that (a+b) - b == a, regardless of overflow, and the CIC algorithm needs that.

```
enum { decimation_factor = 73 };
unsigned s1 = 0, s2 = 0, d1 = 0, d2 = 0, i = decimation_factor;
for (;;) {
    s1 += getchar();  // integrator 1
    s2 += s1;         // integrator 2
    if (0 == --i) {
        i = decimation_factor;
        unsigned dx = s2 - d1; // differentiator ("comb") 1
        d1 = s2;
        unsigned dx2 = dx - d2; // differentiator 2
        d2 = dx;
        putchar(dx2 / (decimation_factor * decimation_factor));
    }
}
```

For each input sample, this involves two additions, a decrement (or increment), and a jump conditional on zero. Each output sample additionally requires two subtractions; in this case, I've also rescaled the output by dividing by a compile-time constant (which usually costs a multiply), so that it will be in (I think precisely) the same range as the original samples; however, this loses precision and dynamic range, and in many cases, no such rescaling is needed.

A 440 Hz signal (A above middle C in A440 standard pitch)

sampled at 8 ksps is 18.18 samples per cycle. The factor of 73 above resamples an 8 ksps signal such that a signal of roughly 438.4 Hz, about 6 cents flat, will occupy 4 output samples, as recommended above for the shower algorithm. How well an actual precise 440 Hz signal will be detected depends on how long you integrate the results over.

The second-order nature of the above filter effectively windows each sample with a triangular window.

Here's an (n-1)th-order version (untested):

```
enum { decimation_factor = 73, n = 4 };
unsigned s[n] = {0}, d[n-1] = {0}, i = decimation_factor;
for (;;) {
    s[0] = getchar();
    for (int j = 1; j < n; j++) s[j] += s[j-1];
    if (0 == --i) {
        i = decimation_factor;

        unsigned dj = s[n-1];
        for (int j = 0; j < n-1; j++) {
            unsigned djprime = dj - d[j];
            d[j] = dj;
            dj = djprime;
        }

        printf("%d\n", dj);
    }
}
```

There's a quasi-inverse of CIC decimation, which is CIC interpolation; in essence, this amounts to using the Method of Finite Differences that Babbage used to tabulate polynomials on the Difference Engine. I say it's a quasi-inverse because it doesn't undo the low-pass filtering that CIC decimation does, I think not even the part that's below the Nyquist frequency of the decimated signal.

I think it's feasible to control the CIC decimation rate using a phase detector, like a PLL, and it may be possible to dither the decimation rate with something like the Bresenham algorithm; however, since the signal out of the comb filter is amplified by a linear factor of the decimation rate, this may be somewhat tricky, as oscillations of the decimation rate turn into oscillations of the output signal amplitude, which needs to be controlled for.

If you are resampling to several different frequencies, the initial per-sample integration steps, and even the counter increment, can be shared between them.

The Minsky Circle Algorithm

In HAKMEM (MIT AI Lab memo 239), item 149 (p. 73) describes an algorithm attributed to Minsky for drawing circles — or, more precisely, very slightly eccentric ellipses:

```
NEW X = OLD X - ε * OLD Y
NEW Y = OLD Y + ε * NEW(!) X
```

Rendered into C:

```
int x = 255, y = 0, n = 1000;
while (--n) {
    x -= y * epsilon;
    y += x * epsilon;
    pset(x0 + x, y0 + y);
}
```

As Minsky comments, "If ε is a power of 2, then we don't even need multiplication, let alone square roots, sines, and cosines!"

Here's a version that outputs a sine wave in minimized C:

```
main(x,y){for(y=100;1+putchar(x+128);x-=y/4,y+=x/4);}
```

Of course, division by 4 can be implemented by an arithmetic shift right by 2 bits.

HAKMEM items 150–152 go into some more detailed analysis.

We can think of the epsilons as rotating the (x, y) phasor by some angle (specifically, $\cos^{-1}(1-\frac{1}{2}\varepsilon^2)$, according to item 151; note that Baker's transcription contains an erroneous omitted superscript 2). If we add input samples to x (or y) then it will sum whatever frequency component is in sync with the rotation:

```
for (;;) {
    x -= epsilon * y;
    y += epsilon * x + getchar();
    putchar(x * scale);
}
```

Frequency components that are not in sync with the rotation will average out to zero. Frequency components that are in sync will grow without limit.

You can use two different ε factors for the two multiplications, which gives a more elliptical "circle" and a denser range of frequencies. In particular, one of them can be 1, which means you can generate a tone of arbitrary frequency with only a single scaling operation per cycle, plus the addition and subtraction:

```
main(x,y){for(y=100;1+putchar(x+128);y+=x-=y/8);}
```

In somewhat more standard formatting, although with a still somewhat eccentric use of `for`:

```
/* Output an audio sinusoid with Minsky's ellipse algorithm */
#include <stdio.h>

int main()
{
  for (int x=0, y=100; EOF != putchar(x+128); x -= y >> 3, y += x)
    ;
  return 0;
}
```

The loop here compiles to this amd64 machine code:

```
  400450:   89 e8                   mov    %ebp,%eax
  400452:   c1 f8 03                sar    $0x3,%eax
  400455:   29 c3                   sub    %eax,%ebx
  400457:   01 dd                   add    %ebx,%ebp

  400459:   48 8b 35 e0 04 20 00    mov    0x2004e0(%rip),%rsi # 600940 <__bss_st
art>
  400460:   8d bb 80 00 00 00       lea    0x80(%rbx),%edi
  400466:   e8 b5 ff ff ff          callq  400420 <_IO_putc@plt>
  40046b:   83 f8 ff                cmp    $0xffffffff,%eax
  40046e:   75 e0                   jne    400450 <main+0x10>
```

Unfortunately, this version spends almost all of its time calling and returning from `_IO_putc`. A version that writes into a 512-byte buffer instead, achieving 850 megasamples per second on my laptop, is as follows:

```
#include <stdio.h>

int main()
{
  char buf[512];
  int x=0, y=100;
  for (;;) {
    for (int i = 0; i != sizeof(buf)/sizeof(buf[0]); i++) {
      x -= y >> 3;
      y += x;
      buf[i] = 128 + x;
    }
    if (fwrite(buf, sizeof(buf), 1, stdout) != 1) return -1;
  }
}
```

Its inner loop compiles to the following nine instructions (again, on amd64):

```
  400460:   89 ea                   mov    %ebp,%edx
  400462:   c1 fa 03                sar    $0x3,%edx
  400465:   29 d3                   sub    %edx,%ebx
  400467:   48 63 d0                movslq %eax,%rdx
  40046a:   83 c0 01                add    $0x1,%eax
  40046d:   8d 4b 80                lea    -0x80(%rbx),%ecx
  400470:   01 dd                   add    %ebx,%ebp
  400472:   3d 00 02 00 00          cmp    $0x200,%eax
  400477:   88 0c 14                mov    %cl,(%rsp,%rdx,1)
  40047a:   75 e4                   jne    400460 <main+0x20>
```

The book "Minskys and Trinskys," by Corey Ziegler Hunts, Julian Ziegler Hunts, R.W. Gosper and Jack Holloway, explores the variations of this algorithm in more detail; I don't have the book, but according to Nick Bickford's 2011 post on the subject, they prove that, using δ for the ε factor that multiplies Y:

$$X_n = X_0 \cos(n\,\omega) + (X_0/2 - Y_0/\varepsilon) \sin(n\,\omega)/d$$
$$Y_n = Y_0 \cos(n\,\omega) + (X_0/\delta - Y_0/2) \sin(n\,\omega)/d$$

where

$$d = \sqrt{(1/(\delta\ \varepsilon) - 1/4)}$$
$$\omega = 2 \sin^{-1}(1/2\sqrt{(\delta\ \varepsilon)})$$

If this is correct, Gosper's earlier result in HAKMEM that $\omega = \cos^{-1}(1 - 1/2\varepsilon^2)$ should be a special case of it (where $\varepsilon = \delta$); it isn't immediately obvious to me why this is so, but these do seem to be consistent for a couple of trial values:

$$\cos\ \omega = 1 - 1/2\ \delta\ \varepsilon$$
$$\sin 1/2\omega = 1/2\ \sqrt{(\delta\ \varepsilon)}$$

I hypothesize, but haven't proven either experimentally or rigorously, that if you start with 0 and add each new input sample to x, you will accumulate a phasor in x and (possibly, depending on the algorithmic variant, scaled) y of all of the samples encountered so far, rotated by the appropriate angle. This will give you the total you've encountered so far of a given Fourier component.

The Goertzel Algorithm

The Goertzel algorithm (sometimes more specifically "the second-order Goertzel algorithm") is an optimized version of Minsky's algorithm, requiring only one multiply or quasi-multiply per input sample. It's actually older than Minsky's formulation, dating from 1958. At its heart is the oscillator $s[n] = (2 \cos \omega)\ s[n-1] - s[n-2]$, which oscillates with an angular frequency of ω per sample; to look at it another way, it's the state transition function $(s, t) \leftarrow ((2 \cos \omega)\ s - t, s)$. To this you just add each input sample x: $(s, t) \leftarrow (x + (2 \cos \omega)\ s - t, s)$; in this way, the energy in the desired frequency accumulates in s and t, and at the end of the accumulation process, you can measure it.

Since $2 \cos \omega$ is constant, this is just a constant multiplication — and some values are inexpensive to multiply by. For example, you can multiply a value a by $1-2^{-4}$ as follows:

```
a -= a >> 4;
```

This is what I mean by "quasi-multiply".

$\cos^{-1}(1-2^{-4}) \approx 0.3554$, and consequently this works out to an oscillation period of $2\pi/0.3554 \approx 17.68$ samples. Here's a C version that emits a sine wave that repeats precisely, due to roundoff error, every 53 samples, for a period of 17.66 samples, 452.8 Hz (lamentably about halfway between A and A# above middle C) in linear unsigned 8-bit samples at 8 ksps:

```
/* Generate a 452.8 Hz sine wave. ./goertzel | aplay */
#include <stdio.h>

int main()
{
  int s = 0, t = 32;
  for (; EOF != putchar(s + 128);) {
    int tmp = s;
```

```
    s += s;
    s -= s >> 4;
    s -= t;
    t = tmp;
  }
}
```

Not counting the output bias addition, this requires an addition, two subtractions, and a bit shift per sample. Here's a one-line version:

```
main(s,t,u){for(t=32;u=s,1+putchar(128+(s-=t-s+s/8));t=u);}
```

(For the special case where $\omega = \frac{1}{3}\pi$, $\cos\omega = \frac{1}{2}$, so $2\cos\omega = 1$; we can thus get a 6-sample cycle with only a subtraction per sample; in that case $(s, t) \leftarrow (s-t, s)$. At 8 ksps this is $1333.\bar{3}$ Hz, 19 cents sharp of E6.)

In a sense, this oscillator works by obtaining the derivative information — stored as an explicit second variable in Minsky's algorithm — from the difference between s[n-1] and s[n-2]. We can rewrite the recurrence relation as follows:

$$s[n] = (2\cos\omega)\, s[n-1] - s[n-2]$$

Let's suppose:

$$s[n-2] = k\cos\theta_0$$
$$s[n-1] = k\cos(\theta_0 + \omega)$$

Presumably in this case s[n] should be identically $k\cos(\theta_0 + 2\omega)$. Is it?

$$\begin{aligned} s[n] &= 2\cos\omega\, s[n-1] - s[n-2] \\ &= 2\cos\omega\, k\cos(\theta_0 + \omega) - k\cos\theta_0 \\ &= k(2\cos\omega\cos(\theta_0 + \omega) - \cos\theta_0) \end{aligned}$$

As you would remember from high-school trigonometry if you were smarter than I am,

$$\cos(t + h) = \cos t\cos h - \sin t\sin h$$
$$\sin(t + h) = \sin t\cos h + \cos t\sin h$$
$$\cos 2\omega = \cos^2\omega - \sin^2\omega$$
$$\sin 2\omega = 2\sin\omega\cos\omega$$

So

$$\begin{aligned} \cos(\theta_0 + 2\omega) &= \cos\theta_0\cos(2\omega) - \sin\theta_0\sin(2\omega) \\ &= \cos\theta_0\cos^2\omega - \cos\theta_0\sin^2\omega - 2\sin\theta_0\sin\omega\cos\omega \\ \cos(\theta_0 + \omega) &= \cos\theta_0\cos\omega - \sin\theta_0\sin\omega \end{aligned}$$

$$k\cos(\theta_0 + 2\omega) = k(\cos\theta_0\cos^2\omega - \cos\theta_0\sin^2\omega - 2\sin\theta_0\sin\omega\cos\omega)$$

$s[n] = 2\cos\omega\, s[n-1] - s[n-2]$
if
$s[n-2] = k\cos\theta_0$
$s[n-1] = k\cos(\theta_0 + \omega)$

then

$$
\begin{aligned}
s[n] &= 2 \cos \omega \, k \cos (\theta_0 + \omega) - k \cos \theta_0 \\
&= k (2 \cos \omega \cos (\theta_0 + \omega) - \cos \theta_0) \\
&= k (2 \cos \theta_0 \cos^2 \omega - \cos \theta_0 - 2 \sin \theta_0 \sin \omega \cos \omega) \\
&= k (\cos \theta_0 (2 \cos^2 \omega - 1) - 2 \sin \theta_0 \sin \omega \cos \omega) \\
&= k (\cos \theta_0 (\cos^2 \omega + \cos^2 \omega - 1) - 2 \sin \theta_0 \sin \omega \cos \omega) \\
&= k (\cos \theta_0 (\cos^2 \omega - (1 - \cos^2 \omega)) - 2 \sin \theta_0 \sin \omega \cos \omega) \\
&= k (\cos \theta_0 (\cos^2 \omega - \sin^2 \omega) - 2 \sin \theta_0 \sin \omega \cos \omega) \\
&= k (\cos \theta_0 \cos^2 \omega - \cos \theta_0 \sin^2 \omega - 2 \sin \theta_0 \sin \omega \cos \omega) \\
&= k \cos (\theta_0 + 2\omega)
\end{aligned}
$$

$\therefore \cos (\theta_0 + 2\omega) = (2 \cos \omega) \cos (\theta_0 + \omega) - \cos \theta_0$ QED

This shows that given two points on a sinusoid of the right angular frequency and any amplitude or phase, the Goertzel algorithm will continue extrapolating further points on it indefinitely.

Measuring the energy is inexpensive (requiring two real multiplies, a subtraction, a couple of squares, and an addition) but not entirely obvious if we only keep around the last two samples of s. The standard presentation is that we transform them into a complex number encoding the magnitude and phase of the signal:

$$y[n] = s[n] - \exp(-i \omega) \, s[n-1]$$

Which is to say:

$$y[n] = (s[n] - (\cos \omega) \, s[n-1]) + i \, (\sin \omega) \, s[n-1]$$

Note that that, if ω is small, the exponential gives a value close to 1, so this is very nearly the difference between the two last values of s.

The idea is that the basic recurrence relation given above for s will rotate this resultant phasor around by an angle of ω without altering its magnitude. Does it?

In the case of an arbitrary multiplier, the Goertzel algorithm beats Minsky's circle algorithm by almost a factor of 2, since Goertzel requires only a single multiply per sample, while Minsky requires two multiplies per sample. But in a case where the multiplication can be achieved by a bit shift and is basically free, Minsky requires just

```
s += t >> n;
t -= s >> n;
```

while Goertzel requires

```
u = s;
s += s - t - (s >> n);
t = u;
```

So you have three additions or subtractions instead of two, and maybe a bit of shunting as well, but one less bit shift.

Another factor to consider is that, for a given multiplier precision, the Minsky algorithm covers the frequency spectrum much more densely; for a multiplier ε (2^{-n} in the example above) the Minsky frequency is $\cos^{-1} (1-\frac{1}{2}\varepsilon^2)$, while the Goertzel frequency is $\cos^{-1} (1-\frac{1}{2}\varepsilon)$. So, for example, if we are multiplying x, y, and $2 \cdot s[n-1]$ by 1/256 to get the value we subtract, Minsky gives us a 3.9-milliradian

rotation and a 1608-sample period, while Goertzel gives us a 63-milliradian rotation and a 100.5-sample period. This is somewhat intuitive, in that the Minsky algorithm applies the multiplier twice per iteration.

The single-scaling version of Minsky's algorithm is presumably similar to Goertzel in its angular resolution, but with two additions or subtractions per sample instead of three; the results I mentioned above for that algorithm should be sufficient to show whether that is true.

You can think of the Goertzel algorithm (or the Minsky algorithm) as integrating the complex phasor y at a particular frequency over the input signal — that is, y is a sum table (or prefix sum or cumulative sum) of the input signal at that frequency. This suggests that if you want to know the amount (and phase) of signal at that frequency between two points in time, you can subtract the corresponding points in y, just as with a first-order CIC filter, and thus inexpensively apply a moving average filter to it. (You will need to rotate the two phasors into phase with a rotation appropriate to the window size, costing two multiplies.)

Goertzel and Minsky as complex integrators

This of course suffers from the rectangular-window problem I mentioned in the shower-algorithm section. You can apply the CIC approach, making a second-order or third-order sum table and then infrequently taking differences at some window width, giving you the amplitude and phase of the chosen frequency windowed by a triangular or near-Gaussian function. However, I think computing these sum tables will require accumulating them as complex numbers and doing the rotation by the appropriate phase before adding each new sample — which, in the general case, requires a complex multiplication (four real multiplications).

However, in the case where we can do the multiplication inexpensively, as in the examples above with a bit shift and a subtraction or addition, this may be a reasonable approach. (See also the section below about avoiding multiplications.)

Karplus-Strong delay line filtering

The Karplus-Strong string synthesis algorithm consists of nothing more than a recursive unity-gain comb filter with a little bit of low-pass filtering. Originally, the delay line was initialized with random noise, but it's the resonances of the filter that provide the envelope and most of the frequency response. Here's a one-line C implementation initialized with just an impulse:

```
s[72]={512};main(i){for(;;i%=72)s[i]+=s[(i+1)%72],putchar(s[i++]/=2);}
```

This version does two indexed fetches, two indexed stores, three divisions, an addition, and a couple of increments per sample, but the divisions can be replaced by equality tests and conditional stores, except for one which is a bit shift. So you need an indexed fetch, an addition, an indexed store, a index increment, a bit shift of 1, and the compare-to-threshold-and-reset operation on the index to implement the circular buffer.

Here's the same algorithm written in a more reasonable way, with a

time limit:

```
#include <stdio.h>

enum { delay = 72 };

int s[delay] = { 512 };

int main()
{
  int i = 0, i2 = 0;
  for (int n = 8000; n--; i = i2) {
    i2 = i + 1;
    if (i2 == delay) i2 = 0;

    s[i] += s[i2];
    s[i] >>= 1;

    putchar(s[i]);
  }

  return 0;
}
```

As a recurrence relation, this is computing

```
s[n] = ½s[n-71] + ½s[n-72]
```

Of course, you can start with an empty buffer and add input signal to it, which will be convolved with the infinite impulse response of the recurrent filter — which response is precisely the sound you hear when running it with no input starting from the above buffer with just an impulse in it. That impulse response has every frequency that fits an integer number of times into the input buffer — the fundamental of 72 samples (111.$\bar{1}$ Hz), but also 36 samples (222.$\bar{2}$ Hz), 24 samples (333.$\bar{3}$ Hz), and so on. The averaging of adjacent samples attenuates higher frequencies.

If you negate the recurrence, it will instead allow signals with an odd number of half-cycles to resonate. For example:

```
s[n] = -½s[n-71] - ½s[n-72]
```

This will preserve signals with a period of 144 samples, 48 samples, 28.8 samples, 20.57 samples, 16 samples, and so forth. This allows you to shorten the buffer by half and eliminates all the even harmonics, which may be a drawback or an advantage, depending on the signal you're trying to detect.

Autoregressive filtering

CIC's integration and comb filters, the Karplus-Strong comb filter, and the Goertzel s recurrence are all special cases of autoregressive filtering, as used in linear predictive coding for speech — they "predict" each new sample as a linear combination of the previous samples. There exist efficient algorithms for finding the optimal autoregressive filter to minimize the (squared?) residual, such as the

Yule–Walker equations. The coefficients of this filter are the coefficients of a polynomial in the z-domain whose zeroes are the formants of, say, a speech signal.

Avoiding multiplication with single-addition and dual-addition multipliers

Several times in the above, we've referred to cases where it's possible to use just a bit shift instead of a constant multiplication, or merely to subtract or add a shifted number. This is interesting because bit shifts are, in hardware and occasionally in software, free — they require zero gates, zero bit operations, and zero time. An interesting question, then, is what set of multipliers we can achieve with a single addition (or subtraction, which requires the same number of bit operations):

```
>>> numpy.array(sorted(set(x for a in range(8)
                           for b in range(8)
                           for x in [(1 << a) + (1 << b),
                                     (1 << a) - (1 << b)])))
array([-127, -126, -124, -120, -112,  -96,  -64,  -63,  -62,  -60,  -56,
       -48,  -32,  -31,  -30,  -28,  -24,  -16,  -15,  -14,  -12,   -8,
        -7,   -6,   -4,   -3,   -2,   -1,    0,    1,    2,    3,    4,
         5,    6,    7,    8,    9,   10,   12,   14,   15,   16,   17,
        18,   20,   24,   28,   30,   31,   32,   33,   34,   36,   40,
        48,   56,   60,   62,   63,   64,   65,   66,   68,   72,   80,
        96,  112,  120,  124,  126,  127,  128,  129,  130,  132,  136,
       144,  160,  192,  256])
```

So, for example, in the neighborhood of 127, we can compute multiplications by 120, 124, 126, 127, 128, 129, 130, 132, and 136 with a single addition or subtraction plus some bit shifts.

Suppose we can manage two additions or subtractions, not just one. Then we can, for example, multiply by 27 with two additions:

```
x += x << 1;  // multiply by 3
x += x << 3;  // multiply by 9
```

Multiplying by 27 in the usual way would have required three additions: x + (x << 1) + (x << 4) + (x << 5).

This approach gives us 1052 separate multipliers with bit shifts of up to 8:

```
>>> singles = set(x for a in range(8)
                  for b in range(8)
                  for x in [(1 << a) + (1 << b),
                            (1 << a) - (1 << b)])
>>> len(numpy.array(sorted(set(a*b for a in singles for b in singles))))
1052
```

Additionally, though, we can add or subtract the original number, possibly with a shift. So, for example, to multiply by 59, although its Hamming weight is 5, we can calculate (x << 6) - (x << 2) - x.

Combining this approach with the previous one gives us 1366 multipliers with bit shifts of up to 8:

```
>>> len(sorted(set(x for a in singles
                   for b in singles
                   for x in [a*b, a+b, a-b])))
1366
```

Allowing larger bit shifts actually extends our range further; with shifts of up to 16, we can multiply by any constant integer factor in (-256, 256) with two shifted additions or subtractions except for the following handful:

```
>>> singles = set(x for a in range(16)
                  for b in range(16)
                  for x in [(1 << a) + (1 << b),
                            (1 << a) - (1 << b)])
>>> doubles = set(x for a in singles
                  for b in singles
                  for x in [a*b, a+b, a-b])
>>> numpy.array([x for x in range(-255, 256) if x not in doubles])
array([-213, -211, -205, -203, -181, -179, -173, -171, -170, -169, -165,
       -149,  -85,  171,  173,  179,  181,  203,  205,  211,  213])
```

This implies that, given an appropriate constant scale factor, we can always do an approximate multiplication with two shifted additions and subtractions while introducing an error of one part in 2·171 = 342 or less. In fact, more than three-quarters of the multipliers in (-1024, 1024) are reachable:

```
>>> len([x for x in range(-1023, 1024) if x not in doubles])
499
>>> len(doubles)
24418
```

This is especially beneficial for higher-precision operands — for example, 16-bit or 32-bit operands, for which the $O(N^2)$ bit operations of a full-precision multiply could be prohibitive.

In cases like the Minsky algorithm and the Goertzel algorithm where constant multiplication is being used to rotate a phasor progressively over time by a constant angle, it may be reasonable to "dither" the angle by alternating between two different inexpensive rotations; this introduces phase noise or jitter, but when it amounts to a small fraction of a cycle, it shouldn't make much difference.

The extended zero-crossing approach

A problem shared by the Minsky algorithm, the Goertzel algorithm, and the shower algorithm is that they are all ways to calculate or approximate a component of the Fourier transform or the short-time Fourier transform (which I think is the best any linear algorithm can do), and as a result they are fundamentally limited by the same uncertainty principle: to distinguish between, in my example, 110 Hz and 106 Hz, they need at least 250 ms of data, regardless of the sampling rate.

I think, however, that phase-locked loops and zero-crossing

detection can do better. In the absence of noise, or with relatively low noise, they can provide a very accurate measurement of frequency with very little data; in the extreme, by measuring the time of a single half-cycle.

However, even that is poor performance; in theory, in the absence of noise, we need only three sequential samples of the sine wave! Any three samples, as long as they're much less than half a wavelength apart! Their first differences give us two slopes, and their second difference gives us the second derivative; the ratio between this second derivative and the central value gives us the negated squared angular frequency. (Roughly; hmm, I should work out what it is for real, because especially for high frequencies this is only approximate.)

The problem is that the phase-locked loop with the square-wave edge-detector that I showed is only getting information from the samples immediately next to the zero crossing — as is the zero-crossing detector. But usually there is lots of useful information further away from the zero crossing, too. We should be able to take advantage of that information to more precisely estimate the phase of the wave at each point in time.

The phase of a pure sine wave of unity amplitude and angular frequency is just atan2(x, dx/dt); for angular frequency ω and an arbitrary amplitude, it is atan2(x, dx/dt/ω), as the amplitude is a common factor of both arguments and thus cancels.

We don't really need to know the precise derivative to uniquely identify a part of the cycle, though. We only need to know the value at that sample and whether the derivative is positive or negative. (Alternatively, we don't really need to know the precise value; we only need the derivative and whether the sample is positive or negative.)

If we count the time interval since we last passed the current phase angle, this should give us some kind of guess at the period of the wave — potentially a new guess on every sample! If we accumulate these guesses over some period of time, we can take the median of the accumulated guesses (rather than, say, the mean) as the period of our wave.

This is like zero-crossing time measurement, but instead of measuring time between crossings of just the X-axis, we're counting the time between the crossings of every single phase. Some degree of hysteresis is appropriate, but now the hysteresis threshold can be more or less an angle rather than an amplitude.

If we're concerned about computation time, though, we probably don't actually want to calculate the arctangent precisely, even if that were possible for an unknown-frequency signal. Instead we would like to lump each sample into some kind of angle bin based on something that's cheap to compute about that sample.

For example, we could draw a square around the origin, with the sides meeting where x[n] == x[n] - x[n-1] and where x[n] == x[n-1] - x[n]. On the right side of the origin, crossing the real axis, we have a side where x[n] > 0; on the left side of the origin, crossing the real axis, we have a side where x[n] < 0; on the top side, crossing the imaginary axis, we have a side where x[n] - x[n-1] > 0; on the bottom side, crossing the imaginary axis, we have a side where x[n] - x[n-1] < 0. Which side we put a given sample on depends on which of these is greatest, subject to a somewhat arbitrary scaling decision

that will double performance for frequencies around some optimal frequency. So if we're on the right side, that means that x[n] is higher than all of 0, x[n] - x[n-1], and x[n-1] - x[n]; if we're on the left side, it means it's lower than all three; if we're on the top side, x[n] - x[n-1] is higher than 0 and higher than either x[n] or -x[n]; if we're on the bottom side, it's lower than all three.

More on phase detection and frequency suppression

Frequecy detection is sort of intimately interwoven with frequency suppression. Consider the following example.

You want to suppress a 50Hz interfering signal introduced into electrical measurements by a nearby 50Hz fluorescent tube. The waveform is periodic at 50Hz, and indeed symmetric, but very far from sinusoidal.

The simplest approach is simply to apply a feedforward comb filter: by adding the signal to itself as it appeared 10ms ago, you will completely suppress the noise from the fluorescent tube, because that comb filter has nulls at 50Hz, 150Hz, 250Hz, etc. But it does some violence to the remaining signal, since each impulse in the signal now appears twice, 10ms apart, thus smearing things out in the time domain and adding 6dB to components of 0Hz, 100Hz, 200Hz, etc. And, of course, if the lamp turns on or off suddenly, you'll have half a cycle of bleedthrough at the end, which is attenuated but not suppressed if the turnoff is gradual.

Within the linear-filtering paradigm, you can trade some of these undesirable characteristics against one another by using more than one previous sample. For example, instead of merely adding the signal 10ms ago, you can add half the signal 10ms ago and subtract half the signal 20ms ago. This results in further temporal smearing of whatever gets through, but the echo signals are now 6dB quieter. If we carry this further and use the average of ten samples (the corresponding points in the last ten half-cycles, half negated), the added echo is now 20dB down.

However, I think we can do better with some nonlinearity. For example, if we use median filtering rather than mean filtering over the corresponding points in the last ten half-cycles, random impulses will not be echoed at all. Or you can use a hybrid: instead of purely the median, use the mean of the six medial values, discarding the two highest and two lowest values as outliers; or use a weighted mean with weights not restricted to 0 and 1. And we could extrapolate an expected amplitude for the waveform to suppress, allowing us to completely suppress sufficiently gradual turn-ons and turn-offs.

Human-voice sounds are periodic, entirely asymmetric, and also very far from sinusoidal due to their laryngetic origin as impulse trains. It's common for the second harmonic to be even stronger than the fundamental! For such signals we couldn't subtract negative half-cycles; we'd have to use the corresponding points in the last ten cycles, instead. And, since they're not very stable in frequency, we'd need to extrapolate frequency shifts as well.

How do you detect the frequency shifts, though? You can look to see if the waveform is to the left or the right of the expected waveform, but of course it's a question of how far to the left or right

you're looking, which perhaps you can reformulate as a question about how to approximate its derivative. Or you could do a full cross-correlation between signals.

One alternative I've been thinking about is to use the signal and a high-pass-filtered version of its *integral* to do phase detection. That way, you don't have the extreme amplification of noise that derivatives get you.

If we can have a prophecy budget, we can use subsequent cycles of the signal as well as previous cycles to estimate the current cycle.

Topics

- Programming (p. 3658) (286 notes)
- Math (p. 3564) (78 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Small is beautiful (p. 3714) (40 notes)
- Audio (p. 3331) (40 notes)
- Numpy (p. 3600) (6 notes)
- Goertzel (p. 3476) (4 notes)
- Minsky algorithm (p. 3586) (3 notes)
- Numerical methods

Web prefetch

Kragen Javier Sitaker, 2017-06-15 (1 minute)

I saw this article: "Ludicrously Fast Page Loads - A Guide for Full-Stack Devs" https://lobste.rs/s/mj7w3d and I thought, 'I wonder what counts as "ludicrously fast"? It turns out the article never actually explains what he means by "ludicrously fast" except that it should be less than 5 or 10 seconds, but it still seemed like a thought-provoking question.

I'd like to get below 10ms, so that even at 100fps it's never more than a screen refresh away. On this shitty DSL, even pinging other people on the same /24 as me takes 50ms, so basically nothing outside my house is under 10ms. So we have to prefetch.

(The article has good points on how to reduce your page rendering times, but I despair of getting HTML5 and CSS to under 10ms. By contrast, some simpler layout models can get much faster indeed.)

15:15 < xentrac> on most internet connections, that unavoidably involves push, but my internet connection is currently about 2.4 megabits per second, and I can only read about 70 bits per second, so the push selection algorithm could have an 0.003% hit rate (i.e. 0.003% of what it downloads is something I actually choose to look at) and still be useful 15:17 < xentrac> huh, 15:18 -!- akurilin2 [~alex@208.80.70.250] has joined #lobsters 15:20 < xentrac>

Topics

- Performance (p. 3621) (149 notes)
- Networking (p. 3594) (7 notes)
- HTML (p. 3508) (6 notes)
- Browsers (p. 3351) (6 notes)

A mechano-optical vector display for animation archival

Kragen Javier Sitaker, 2014-12-28 (updated 2015-09-03)
(28 minutes)

DRAFT

I was sitting in Christmas Mass, sweating in midsummer, and the reflection of the lights off the oscillating fans pointed out to me the immense unrealized potential for mechano-optical laser displays, like the kind we used for psychedelic light shows back in my childhood. In particular, we can use them for archival of moving pictures.

However, it looks like opacity holograms will continue to be more practical and have comparable information density.

Reflecting a laser to a desired point on the wall

Point a laser pointer obliquely at a mirror-surfaced cylinder rotating around its axis. The laser pointer bounces off the cylinder and makes a spot on the wall, at an angle that depends on its angle to the surface; because the surface is curved, it diverges a bit in the plane perpendicular to the cylinder's axis. It doesn't move as the cylinder rotates.

Now suppose we cut a flat mirror-finish facet into the cylinder, parallel to the axis, such that it will rotate into the beam at some point. When this happens, the spot on the wall will jump to a new position as the surface under the beam abruptly changes angle; then, as the cylinder continues rotating, the rotation of that surface will move the spot perpendicular to the cylinder's axis; and finally, when the beam runs off the other edge of the facet, the spot will jump back to where it was originally.

If we rotate the cylinder fast enough that the motion isn't visible, it will draw a short dash on the wall around the point where the beam normally ends up.

The angle subtended by the dash, as seen from the center of the cylinder, is exactly twice the angle subtended by the facet.

If the facet, instead of being flat, describes a logarithmic spiral centered at the center of the cylinder, then instead of a dash, we will just be drawing another point on the wall.

If, instead of making it a spiral, we angle the facet so that it's not flat and parallel to the axis of the cylinder, but conical instead, we can displace the point parallel to the axis as well.

By combining conicality and spirality, we can place the point anywhere on the wall.

From here on, I'm going to use "up" and "down" to mean the directions on the wall parallel to the cylinder's axis, "right" to mean the direction in which the point reflected from a flat axis-parallel facet would be moving, and "left" to mean its opposite.

If a facet, rather than being perfectly flat, is slightly concave, we can focus the initially-collimated beam to a smaller point on the wall,

if we know the focal distance to the wall, so that the point can be smaller than the illuminated part of the facet. This actually isn't going to be useful for the rest of what I'm discussing here.

Animations by scanning the beam over the medium's surface

By forming a series of many such facets on the circle that the beam describes on the surface of the cylinder, we can draw many points on the wall in quick succession, forming an arbitrary image --- indeed, one that changes over time. If the facets are smaller than the spot on the cylinder illuminated by the beam, several of them will be illuminated simultaneously, and indeed we can run several parallel tracks of them. Smaller facets allow us to program a larger number of points.

Fresnel reflectors allow you to keep surface nonflatness to small scales

There isn't a guarantee that the facets will form an approximately cylindrical shape if we put them edge to edge. Consider, for example, the case where the animation is just a single stationary point to the left of center; your shape is a logarithmic spiral. And, of course when one point is significantly above or below its predecessor, there is a gap between the facets, which must be bridged in some fashion. You can do this Fresnel-lens-style by introducing discontinuities in the surface that won't be illuminated, so that the aforementioned single-point-left-of-center has the shape of a circular-saw blade rather than a continuous spiral.

If you bias the image to be in one particular direction from the centerline, such that an unbroken surface encoding it would be roughly a logarithmic spiral expanding as it rotates, you can ensure that the discontinuities introduced by the Fresnel-reflector operation are never illuminated, so you don't lose any light to them.

Edges of facets will tend to diffract: a planar wavefront reflected from the center of a flat facet will be reflected as a planar wavefront, but the part that reflects from the edge of that flat facet will form cylindrical wavefronts spreading out from the edge. If the facet is very large compared to the wavelength of light, the diffraction from the edges will not be significant, but if it is small, a significant amount of incident light will end up reflected as stray light by this mechanism. Worse, if these edges occur at regular intervals (like the sawblade shape mentioned previously) the diffracted light will form planar wavefronts going in other directions, but divided up by color, like any other diffraction grating. It's possible that you could use this mechanism to get an arbitrary color picture from a white light source, but I'm not sure you have enough degrees of freedom; alternatively, randomizing the discontinuities should break up such patterns and result in the stray light being just a generalized wash.

Diffraction-limited microradian pixel density is around one frame per 4 sq mm

opattern.svg" > Public-domain Airy disk image by :en:Sakurambo.

Also, though, the divergence of the reflected beam is diffraction-limited: a small facet is like a small aperture, and so you end up with the projected point being an Airy disk. This is what imposes the information capacity limit on this medium: if you try to light up more pixels by making smaller facets, you have more stray light and you unavoidably light them up with bigger spots.

So how many pixels can you have?

This depends on the Airy limit. Suppose we use 600nm as our relevant wavelength, which is a slightly orangish yellow and very nearly as sensitive as our daytime spectral sensitivity peak of 555nm.

So how far off the center axis of a facet at right angles to the light source do you have to go before the far edge of the facet is a whole wavelength further from you than the near edge, so that you're in the first null? Consider if we try to make our pixel facets 20μm across. The sine of that angle is 600nm / 20 microns, or about 1/32 to 1/40, so it turns out it's about 1.4 to 1.8 degrees --- and the whole bright beam reflected from that facet, inside that first null, is therefore a cone of 2.8 to 3.6 degrees. (1.22 λ/d is the Airy formula typically given for a round aperture, which is a little smaller than the 2 λ/d I'm using here.)

Public-domain laser diffraction Airy disk photo by :en:Anaqreon.

To put that in more normal terms, a visible-wavelength light beam passing through a 20-micron aperture will be diffracted into a cone with about 3 or 4 degrees of divergence.

(To see if I was smoking crack here, I checked the Wikipedia article on "beam divergence". It says, "Gaussian laser beams are said to be diffraction-limited when their radial beam divergence [half the cone angle] is close to the minimal possible value, given by $\theta = \lambda/\pi w$, where λ is the laser wavelength and w is the radius of the beam at its narrowest point, which is called the 'beam waist'." In this case, λ/w is about 1/16 to 1/20, so our divergence in radians should be a πth of that, which is only slightly narrower than the answer I got above, and the difference may depend on where you set your cutoff.)

This stands in sharp contrast to what we normally expect from, say, a megapixel display, which is pixels of something like 2 arcminutes, 0.03 degrees, two orders of magnitude better. You can improve things by putting the projector closer to the wall than your eye is, but you're still only going to get reasonable resolution over about one radian in each dimension, before things start to kind of sketch out around the edges due to the super oblique angles.

If you want to get megapixel-sharpness images out of this thing, you're going to have to use bigger facets so that you can get milliradian-level resolution. So you can't get 2500 pixels per square millimeter in a useful way.

You can use curved facets if your image is made out of lines; the curvature is equivalent to a bunch of facets that are very narrow in one dimension, and so scatter the light more in that direction, along the line of light, at right angles to the long skinny facet.

A mixed model may provide the best information density: use

small facets toward the center of the beam to direct the overall brightness of a scene to the right places (with necessarily high divergence) and larger or curved facets, perhaps in the dimmer outer parts of the beam, to draw outlines with milliradian-resolution definition. Such a facet might be 10 microns by 1mm, diffracting the incident beam into a line about 100 milliradians by 1 milliradian, in some arbitrary orientation, placed in some arbitrary place on the screen. You have space for, say, 75 such lines in a square millimeter, with another 400 facets in the range of 24 microns across, each directing some brightness onto an area something like 30 milliradians in size. 1–4 square millimeters should be enough to encode a "frame" of video, and getting a spot of light down to 1–4 millimeters is easily feasible. It isn't necessary to get it *up* to 1–4 millimeters, because instead of using a diffuse spot, you can just scan the spot quickly over a long distance of the surface.

This means that a second of animation is something like 100 square millimeters, or a square centimeter, of reflective medium.

There's an additional degradation of resolution in the left-right direction caused by the beam striking the reflected medium being of nonzero size and the medium rotating through it, but I don't think that's important for two reasons. First, there's no fundamental obstacle stopping you from making that beam be however small you decide to make your individual facets; second, you can diminish that rotation arbitrarily by using arbitrarily large cylinders, or even using a flexible cylinder that you depress to be locally flat so that its movement under the beam is purely translational, or rotational around its focal point on the wall. If you don't take such measures, though, a 1mm-diameter beam illuminates about 1/50 radian on a 10cm-diameter cylinder, so you get a left-right smearing of about 1/25 radian from the rotation.

As a point of comparison, fancy professional laser shows often use galvanometer-driven mirrors capable of 25000 points per second, and what I've proposed above (475 "points" per square millimeter and 100 square millimeters per second) is about 47500 points per second; so this is clearly capable of producing visually arresting animations.

Solar illumination instead of using a laser

If we don't have a laser pointer, we can use a sunbeam through a hole the size of the desired beam. The divergence of the beam, and thus the resolution of the projected image, will be limited by the angular size of the sun in the sky, about half a degree. This resolution can be improved substantially, at the cost of brightness, by putting a pinhole between the hole and the sun, blotting out most of the sun's disk. If the pinhole is fairly near, that will introduce substantial nonplanarity in the light wavefront, which will change the effective focal length of concave facets.

A milliradian-sized pinhole to blot out most of the sun will blot out about 74/75 of it, reducing the illuminance available to light the image by about 20dB, down from, say, 100 kilolux down to, say, 1000 lux at the point where the beam hits the facets, so the illuminance on the screen will be something like 1 lux, if it's being spread out over a screen area around 1000 times the size of the illuminated area on the facets. This probably means that such an animation will require a dark room for viewing by sunlight at full resolution without

concentrating the sunlight using some kind of nonimaging optics.

(Of course, if you just illuminate it with direct sunlight, you'll get the image anyway, just blurred by convolution with the sun's disk.)

Non-cylindrical media

So far I've been talking about cylinders, but all of this continues to work just as well for animation if we're talking about a disc, too, or even just a flat sheet. Any illuminated circle on the surface corresponds to some image; as the circle shrinks, it will include images from fewer and fewer physically adjacent frames of video until it starts including only certain drawing elements of a single frame; and any path the circle takes over the surface will produce an animation. Purely translational motion also eliminates the rotational left-right intraframe smearing mentioned in an earlier section, which can easily reach tens of milliradians.

Low-resolution text

Suppose that, instead of shooting for milliradian-resolution line drawings, you just want to put an animation of readable text onto your surface. Maybe you only need to be able to display a word or two at a time, or even a letter. Does this help? Remember, before, we were looking at something like only 100 vectors per square millimeter, because of milliradian-level sharpness requirements at visible wavelengths.

How many vectors do you need per letter?
<img src="http://canonical.org/~kragen/bible-cropped.png" >

I designed a 6-pixel-tall proportional pixel font a while back, with the objective of conserving text with laser-printed microscopic letters on paper, which encoded my 4.45-megabyte test Bible in 4866x19254, or 21 black-and-white pixels per letter, which means that each letter averages 3.5 pixels wide, including the space needed between the letters to make the text readable. It's black on white, and it's about 29% black, so that's about 6.1 black pixels per letter. Based on that, let's figure that we probably need about 4 vectors per letter and a resolution of about 0.05 radians.

Well, this lets us use the originally-hoped-for 20-micron-wide facets, which do in fact give us about 0.05-radian resolution. For lines, again, we can use narrow facets that diffract light into a streak in the desired direction, so that they actually take up *less* space. For example, a lowercase "i" might require a full 20-micron-square facet for the dot, but only a 20-micron-wide by 6-micron-tall facet for the vertical line beneath it.

If we figure that the average facet is then half of that 20-micron-square configuration, then we can get about 5000 vectors per square millimeter, or 1250 letters. That is similar to my laser-printed microprint approach, which at 1200dpi only gets 2232 pixels per square millimeter, which works out to about 1100 letters; but it doesn't need a microscope to read it. (A laser-printed 1200dpi pixel is about 20 microns in size, so this comparability is not totally surprising.)

Now, though, we run into a different problem. An average word might need only about 20 vectors to display it, but it's going to be projected over 0.3 by 0.9 radians, which means that you only have room for about three or four words to be displayed at a time. But

that's 60–80 vectors, which fit into a 200-micron-square area. Maybe you can get a laser pointer down to 200 microns square, but getting a sunbeam that small is hard.

If, instead, we figure that we need something more like 0.015-radian resolution, and thus our facets need to be 60 microns across when they're full points, we can fit 30 or 40 words onto the screen at a time. But now we only have about 500 vectors per square millimeter, which is also about 40 words. 40 words per square millimeter is somewhat inferior to microfilm, but dramatically higher than traditional printing's 0.014 words per square millimeter.

It's probably best to organize the words such that they "scroll" around the screen like the words in an old ytalk session: rather than attempting to move old words, after filling up the last line, you wrap around to the top and put new words on the top line. If your spot of light is too small, you'll have less than a 30–40-word screenful of text visible, while if it's too large, you'll have new words overlapping old words, maybe several screensful. Ideally, as you scan your light beam over the text, new words fade in shortly after old words fade out. You can display some kind of position indicator in some fixed part of the screen; it can be quite a long streak, so it can use up very little space on the medium.

Solar sundials

You can use this technique to make a super awesome solar sundial which not only projects the time, to the minute, onto the wall, but also tells you what day of the year it is, although typically you'll have two choices for that.

Fabrication techniques

How would you go about fabricating such a demandingly-shaped reflector? If it's a one-off, I think you can probably use electrochemical machining, which is sort of the opposite of electroplating --- you make your workpiece the anode, pumping electrons out of it, which it then hungrily obtains from negative ions floating around in water, which then combine with metal atoms from the surface; you limit the current flow to a tiny part of its surface area by moving the cathode around very close to it; and you wash away the electrolyte rapidly enough to prevent the metal ions from the anode from being reduced onto the cathode. Due to anodic leveling, it produces a mirrorlike finish.

You probably want to use a sort of rake-shaped cathode to produce the desired surface complexity, dragging many wire points over the surface, each with its height controlled separately to within a fraction of a micron, say, 50 nanometers. They should probably be spaced something like 10 microns apart, 100 of them to a millimeter. Each one will tend to cut out a spherical shell into the metal around it, cutting a cylindrical trench through the metal as it moves. We want to keep the deviation from the desired shape of the metal down below, say, 100 nanometers, to avoid fucking up the light wavefront too much, and there's 5 microns from the center to the edge of the trench; acos(1 - (100nm/5 microns)) = 0.2 radians, so we want those wire points to be carving out 25-micron-radius spheres. That's going to make it a little tricky to control the angles of 10-micron-wide facets, but I think it's doable.

A 25-micron inter-electrode gap is smaller than is typical for electrochemical machining, but not outrageously so; 80 to 800 microns is typical. Some "pECM" processes use an inter-electrode gap of as little as 10 microns, along with vibrating electrodes and pulsed current.

We can do a little electropolishing afterwards to try to anodically smooth the surface, but we'd like to be removing very little material. Electropolishing is also going to round off the Fresnel-reflector knife edges and result in more stray light.

How much overall material do we need to remove?

Suppose our facets have angles varying over more or less a quarter of a radian in every direction from the overall surface, and that we can generally choose to alternate them to minimize discontinuities. In the above, the biggest facets were up to a millimeter in size. If we're cutting millimeter-sized facets at a quarter-radian from horizontal, we have to cut up to a quarter-millimeter deep! On average, we might be removing 100 microns of the surface of the material, 100 milliliters per square meter, or 0.01 milliliters per square centimeter.

If we're doing this in aluminum (which may not be the highest-quality choice, but millimeter-thick gold is expensive) that's 27 milligrams per square centimeter. Aluminum is conveniently 27 grams per mole, so that's a millimole per square centimeter. We have to pump off three electrons from each aluminum atom in order to turn it into aluminum chloride, so we need three millimoles of electrons per square centimeter. A coulomb is only 6.2e18 electrons, while a millimole is 6.0e20, so that's 97 coulombs per square centimeter, plus whatever gets wasted in electrolyzing the water; at 1 ampere, that's about a minute and a half per square centimeter, or 280 micrograms per second.

Some random paper about the material removal rate of electrochemical machining (Sudiarso, Latifah, Ramdhani, and Mahardika 2013) says that at 0.6 to 1.09 amps on aluminum 1100 with a 2-mm-diameter brass cathode, at 15 volts, they were able to remove 79 micrograms of aluminum per second. That's a factor of almost four less efficient than the calculation I pulled out of my ass above, which is far more accurate than that calculation had any damned right to be.

That also works out to be about (79 micrograms/sec) / (0.8 amps * 15 volts) = 6 or 7 micrograms of aluminum removed per joule. 27 milligrams per square centimeter is then 4 kJ/cm^2. This is a very small amount of energy.

How can you control the position of 100 metal points per millimeter to within 50 nanometers, with a total range of vertical motion of 250 microns (5000 times the resolution)? This seems very challenging indeed. The most promising approach would seem to be using a small number of points, such as 10 to 30, with a separate piezoelectric actuator for each one; a small hydraulic actuator might also work.

I conclude that it's probably feasible to do electrochemical machining of these surfaces, but it requires making some advances in electrochemical machining.

Electrical discharge machining (EDM) is reportedly somewhat more advanced than ECM and can typically produce even better surface finishes, but it doesn't seem like it is applicable to this process,

because it consumes its electrodes. Wire EDM is a common process, using a consumable wire electrode, but it runs the wire through the workpiece and then discards it; so it can't be used to make concave shapes. Perhaps it might be possible to perform a kind of wire EDM that consumes the end of the wire, completely, rather than merely spark-eroding the side of the wire somewhat.

It might make more sense to press or stamp the facets into the surface using a hard tool, one press per facet, particularly if you're using a soft metal like aluminum or gold. You need five degrees of freedom (two to control the reflection angle of the facet) with 10-micron positioning precision, and, in the Z-axis, enough force to stamp plastically 125 microns deep into the surface.

For reproduction, there are existing models of mass-production of reflective metallized surfaces with micron-level detail, specifically for diffraction gratings. Master gratings are cut into glass on a ruling engine at the Grating Lab in upstate New York, which has three ruling engines, one made by the hands of Michelson himself; and then casts are taken in plastic from the glass, and then other casts from the casts, and so on. Each grating can only withstand a few casting operations, perhaps five or ten, before being damaged enough to degrade its quality, but you can make several generations of them, which is sufficient to supply the world demand for lab-quality diffraction gratings. The final gratings are metallized, typically by vacuum deposition of aluminum.

Exactly the same process would work for these animations, although the particular resin and hardening process used for the gratings seems to be a secret.

Topics

Gold leaf trusses

Kragen Javier Sitaker, 2019-08-31 (11 minutes)

Everything we make has some minimum mechanical strength to keep it from breaking at undesirable moments, whether a spoon, a skyscraper, a scarf, a sedan, or a circuit; and this strength has several different components, including resistance to tension, compression, shear, flexion, and impact, which trade off against one another to some extent. Also, we make many things with more material than is needed for its mere mechanical strength, in order to make them more rigid. As described in Sandwich theory (p. 2450), scaling laws are such that many of our large structures fail by buckling, which is limited by rigidity rather then tensile strength, compressive strength, shear strength, or impact toughness; and we take advantage of this by producing structures like I-beams, structural tubing, and sandwich panels which have greater rigidity for the same mass.

But I got to thinking about the implications for very small structures, and I came to some astonishing conclusions about the possibilities of massively parallel automated fabrication, even without molecular nanotechnology.

Think about the aspect ratio of the legs of a daddy-long-legs.

Scaling down a highway sign

But consider scaling down a highway sign. It's cantilevered out over the highway on a steel truss to support its weight and occasional wind loadings. The truss is made of angle irons, L-shaped profiles extruded or bent from sheet steel. To make up some dimensions, let's suppose the angle irons are 6 mm thick, and each leg of the L is 30 mm long, and the truss is in the form of a square beam 500 mm across and 5 m long, any given cross-section of which has 8 angle irons running through it for a total of almost three thousand square millimeters of steel out of the 250,000 square millimeters of the cross-section. (This puts the weight of the truss itself in a bit over 100 kg.) Perhaps the truss can safely support a tonne of road sign and whatnot at this 5-meter lever arm, despite being almost 99% empty space.

Now, suppose we scale it down by a linear factor of 100. Now it is 50 mm long and 5 mm across, weighing a bit over 100 milligrams, and the road sign has shrunk from a tonne to a gram. The moment at the base of the truss has gone from 49 kN m to .00049 N m, a decrease of not six but eight orders of magnitude. But the steel cross-section in tension at the top of the truss has diminished by only four orders of magnitude. So, while previously it was near stressed to near its yield stress, now it is four orders of magnitude away from its yield stress. Suppose its Young's modulus is 200 GPa; its elongation might previously have been 0.07%, or about 3 mm, but now it is 0.000007%, or about 3 nanometers.

This means that we can thin out our angle irons quite a bit. We scaled them down from being 6 mm thick to being 60 microns thick, which is about six times as thick as common aluminum foil (see Single-point incremental forming of aluminum foil (p. 769)). If we managed to scale them down by four orders of magnitude, they

would be 6 nanometers thick, and would still be able to hold up the one-gram model highway sign, with the same relative deformation and safety factor under its scaled-down load as the original full-scale highway sign.

This change would change our model truss from being 99% empty space, or rather air space, to being 99.9999% empty space. Instead of 100 milligrams, it would weigh 10 micrograms, which it turns out is about 0.8% of the weight of the air in its air space. It's 99.2% air by mass now, lighter than the lightest silica aerogel.

However, removing 99.99% of its remaining solid material has also removed 99.99% of its tensile and compressive strength. While previously its tensile and compressive strength was proportionally 100 times greater than those of the full-scale highway sign, now they are 100 times less. So perhaps it will fail in some other way when trying to support the model highway sign; we might need as much as a full milligram of steel to provide the necessary tensile and compressive strength. But at this scale we are no longer obliged to provide massive amounts of unnecessary tensile and compressive strength merely to get adequate rigidity and flexural strength.

(The very light model highway sign would probably be unable to withstand much of a breeze.)

Now, we can't actually do this with steel; it isn't malleable enough to roll that thin, and in contact with air, I doubt 6-nanometer-thick steel foil would last long.

Gold leaf

However, we can do it with gold leaf, which is typically 0.2 microns thick (according to Compressed sensing microscope (p. 306)) and stable in Earth's atmosphere. Normally we think of gold as being an extremely expensive material to build things out of; it currently costs US$1480 per troy ounce. But if we only need a milligram of gold to build our model, that's only 5 cents at that price. We also think of it as being impractically soft and weak, but with the relative strength boost we get from scaling down in this way, we no longer need the brute strength of steel for most things.

Gold leaf is delicate enough that you need to use special hand tools and blowing of air to manipulate it without breaking it. More practical for many uses may be gold foil, which comes in thicknesses of 1 micron to 10 microns.

Carbon nanotubes

Carbon nanotubes are thinner than gold leaf as well as stronger, and they are also stable in air. They may provide a better material than gold leaf or gold foil.

Glass fiber, basalt fiber, and silica crystals

These materials are also stable in air and are stronger than gold as well as lighter; short enough spun fibers (or cut crystals) will not buckle under compression. You can build trusses out of them if you can make joints, but often at these scales the problem is not so much getting things to stick together as avoiding catastrophic accidental stiction and cold-welding.

Metamaterials: rigidity instead of strength?

More generally we can think of the existing nearly-isotropic bulk materials we routinely build things from — steel, cement, brass, glass, and so on — as meeting their rigidity requirements at a low space cost by virtue of spending a lot of mass on the problem. Historically we've been able to sometimes get lower mass (and lower cost!) by using wood instead, when we can afford a larger volume. Wood is a nanostructured metamaterial, but it, too, is somewhat optimized for low volume, perhaps so that trees can resist wind but I think largely so that they can resist predation. Balsa wood, pith-core trees and the remarkable moringa demonstrate the existence of other possibilities.

Suppose that by using trusses made of gold, glass fiber, or carbon nanotubes, we can produce metamaterials with much better stiffness-to-density ratios. Will this enable us to use only enough material to provide the tensile and compressive strength and impact resistance we need, in macroscopic structures? Maybe not, because the rigidity of a structure and the modulus of elasticity of a material (or metamaterial) are different things.

In Plastic cutters (p. 1074) I say ASTM A36 steel has a Young's modulus of 200 GPa, and like iron and steels in general, a density of 7.9 g/cc; that makes its stiffness-to-density ratio 25 kJ/g. Balsa wood has density ranging up to 0.38 g/cc and axial Young's modulus up to 9 GPa, giving a surprisingly lower, and similar, 24 kJ/g.

Why are these so similar? Maybe because balsa wood's axial elasticity, like steel's, comes from straining crystal lattices and atomic bonds, and the particular crystal lattices and atomic bonds involved aren't enormously different, perhaps by a factor of 2. Balsa wood spreads them out over a larger cross-sectional area, the rest of which is air, which contributes insignificant mass and stiffness; it thus decreases its density and its Young's modulus proportionally. The same is true of, for example, steel tubing, which is as light as balsa wood once its width is on the order of 100 times the thickness of its walls.

There are really three contexts where we have to brute-force thicken things up to get the rigidity we need instead of just spreading the mass over a larger area. One is where there's some mechanical constraint that makes the extra space unavailable: your truck has to fit under bridges, your boring bar has to fit into the hole being bored, your axle needs to fit through the bearing. A second is where we need something like *hardness*: the force that must be resisted is being applied at a point or over a small area, and so we need a concentration of material in that area to resist it. A third is where making the truss or honeycomb structure or whatever is difficult or expensive because of the limitations of our fabrication technology, and in general that's a question of things being very small.

(Balsa wood is still better than any artificial material so far, though.)

Flight

Very-low-density nanostructured materials made out of thin members have a long history in flight; dandelion seeds, feathers, and parachuting spiders are three examples. Aside from the possibility of lighter-than-air flight (a gold-leaf balloon made of two sheets sealed together at the edges should be able to fly if you can fill its middle

with at least 4 mm of hydrogen or helium) the possibility of ultralightweight metamaterials enabling insect-scale structures would seem to offer many fascinating options. Also, structures so delicate that they could easily be blown away might be best off if far from the ground.

Topics

Electrolytic anodizing, with a small movable electrode

Kragen Javier Sitaker, 2018-10-28 (2 minutes)

You can produce different brilliant colors on titanium, or on metal coated with a titanium sputtered coating, by producing different thicknesses of titanium dioxide surface layers. Because titanium dioxide has such a high refractive index — 2.5–2.6 — the reflection from the top of the coating is much stronger than from other similar metal oxide coatings, resulting in almost complete interference and thus strongly saturated iridescent colors.

The conventional way of making such oxide layers on titanium is by heating it in air, with the temperature of the surface largely determining the thickness of the oxide, but this carries the drawback that it can generally only produce very gradual variations of thickness across a surface, because if there are large temperature differences over small distances in the surface, they will quickly disappear. Perhaps with localized laser heating, this disadvantage could be removed.

But I have another idea in mind. Rather than heating the material to promote diffusion of oxygen through the oxide layer, let's anodize it. This involves applying a positive voltage to the titanium in an aqueous electrolyte, stealing electrons from the surface metal atoms and giving them to water molecules, liberating oxygen from them to combine with the metal.

Anodizing titanium to produce different colors is a known technique: https://thekidshouldseethis.com/post/anodizing-titanium-the-rainbow-metal, for example.

If this is done with a cathode very close to the surface of the anode, it should produce a very localized coating; and applying varying amounts of current to different spots on the anode should produce different coating thicknesses, and thus different colors.

This should enable the production of brilliantly colored patterns on titanium surfaces. I don't know if the full gamut of visible colors can be thus produced by dithering light from nearby dots, but I suspect so.

Topics

Oval cam lock

Kragen Javier Sitaker, 2019-11-26 (5 minutes)

Reading Victor Papanek and James Hennessey's inspiring 1974 *Nomadic Furniture 2* and looking at the interesting "Jim's Peg-Lock System" on p. 108, I got to thinking about camlocks as a substitute for screws.

Jim's Peg-Lock System inserts a short wooden dowel into a round hole through a square piece *intersecting* a wooden dowel of the same size run through another round hole in the same square piece at right angles to it --- slightly skew to it but without enough distance between their axes to prevent interference. This is reportedly explained in more detail in *Nomadic Furniture 1* pp. 126-7; from this volume, though, it's not clear whether you cut a bite out of the other dowel or just compress the heck out of the wood. If there's a bite out of the other dowel, it will be under no force, but cannot be withdrawn from its hole unless the peg is removed. If there's no bite, or a bite that's slightly too small, the wedging mechanical advantage is potentially enormous.

This got me to thinking about the celt, a type of stone axe, and oval dowels. If you stick an oval dowel in an oval hole made to fit it loosely, it can slide linearly but binds up if you start rotating. If you apply further force, you may be able to get it to rotate 90°, depending on how oval it is. (In the circular limit, the mechanical advantage is infinite, but the displacement is zero, so the force is limited by the sponginess of the wood.)

If the short axis of the oval hole runs across the grain of the wood, then the rotated oval dowel will be pressing against endgrain, like a well-made celt; in this configuration it will not fall out from moisture changes and will not tend to split the wood. At the 90° rotation angle, it is in an equilibrium; it does not have a tendency to twist back one way or the other. There are no stress risers because the wooden surfaces are smoothly curved. Some kind of wax, drying oil, or soap might be a good way to lubricate the initial assembly in a way that stabilizes it further without making disassembly impossible; glue would make the joint permanent.

If instead the oval hole's long axis is parallel to the wood grain and a slit cut is made to the end of the wood, through the oval hole, and some distance beyond, the rotation of the wooden dowel can force the holed wood to expand laterally, pressing on the inside of some other slot or hole, rather like a wedge tenon, but reversible. If the slot or hole is closer to the fulcrum than the hole is, this can provide a further mechanical advantage. An unoccupied round hole at the base of the slit can eliminate stress risers there and increase compliance for this use.

Alternatives to an unobtainium oval dowel might include two round dowels with equal flats cut on one side and face-glued, and an eccentric round dowel, like a crankshaft. The eccentric round dowel is easier if the two cylinders are of different sizes so that one is entirely contained in the hypothetical extension of the other. This crank configuration, if used to power a prismatic joint by sliding in a slot, provides a variable mechanical advantage that reaches a

zero-displacement singularity, like a Vise-Grip or similar toggle linkage. If movement is permitted a bit past this singularity and then stopped, for example by the end of the slot, it can support a *stable* equilibrium not dependent on friction and thus not vulnerable to vibration.

A similar stable equilibrium can be provided in the oval-hole mechanism by pooching out the sides of the oval hole a bit at the ends of the short axis, thus converting the unstable equilibrium into a stable equilibrium (with unstable equilibria near it on each side), and eliminating the need for linseed oil or whatever as a form of wood Loctite.

If, instead of sliding against the wall of a slot near the end of its travel, the eccentric dowel is rolling another disc or dowel along a track, it may be possible to make this mechanism work without any sliding friction, just rolling friction. If the two eccentric arcs are cut into opposite edges of a piece of material, it may be possible to make the entire wedging mechanism planar with sheet cutting.

Mechanisms like these are viable alternatives to screws in many situations.

Topics

Constant space lists

Kragen Javier Sitaker, 2018-12-10 (10 minutes)

We can manipulate arrays of (equal-sized) objects in constant time without the possibility of failure and without creating garbage. It would be desirable to also manipulate flexible, irregular data structures in constant time without the possibility of failure and without creating garbage. These three desiderata are nearly in increasing order or specificity — allocation implies the possibility of allocation failure, and if you are occasionally creating garbage, then eventually you will run out of memory if you don't allocate any more; and allocation almost always implies non-constant-time behavior, except in extreme cases such as stack allocation and allocation from a fixed pool.

The Lisp memory model and failure

The Lisp memory model, an arbitrary directed object graph with garbage collection, is very flexible; it lets us treat arbitrarily large and irregular objects as if they were register values, just by using their memory addresses to name them, as long as we don't mutate them. That's why most modern programming languages have adopted it: Python, Java, Lua, JS, and many others. (Others, like C++ and Golang, use a hybrid scheme in which some objects are embedded in other objects.)

The fundamental object graph construction operation is to create a new object from a tuple of (references to) existing objects; its simplest form is Lisp's `cons`. As long as you stick to this operation, reference counting is adequate, though slow, to manage memory; generational garbage collection without a write or read barrier is also adequate, and fast. However, note that this operation can fail. The price of using this memory model is that memory allocation is ubiquitous, so nearly any computation can fail or take an arbitrarily long time, so it is best used for programs where failure is an option.

Rather than constructing new objects, we could mutate existing objects. In this case, the fundamental operation is to overwrite a field with a reference to some existing object. Since this permits creating references from older objects to newer ones, generational GC becomes more complicated, and cyclic references make refcounting dangerous. The operation cannot fail, and it's constant time if you're not using refcounting, but it can create garbage, and you cannot tell whether it has created garbage or not without a global reference graph computation.

The Z-machine

The Zork Z-machine memory model was designed for flexibly simulating virtual worlds on tiny microcomputers where memory exhaustion was a constant danger, and one which really harshes the buzz of dungeon exploration if it comes to pass. It runs in constant space and constant time and does not create garbage, and although it is not as flexible as the Lisp memory model, it is more flexible than most of the alternatives.

All objects in the Z-machine are arranged into a single hierarchy by

way of three pointers per object: parent, first child, and next sibling. The fundamental hierarchy mutation operation is to change the parent of an object, which cannot fail. In Zork and similar games, this was used to express physical containment.

(I lied somewhat here when I said that it runs in constant time and does not create garbage; if you allow an object to become a descendant of itself, it can create garbage, because then it becoes disconnected from the rest of the hierarchy; and if you check to see if you are doing this, that check will not run in constant time, and it may fail.)

Sketchpad's ring structure

In Ivan Sutherland's Sketchpad, each type of object participated in one or more "rings", which were intrusive circular doubly-linked lists in memory. XXX add more info

Zzstructure

Ted Nelson's "ZigZag" program relates objects along an arbitrary number of "axes", which can be traversed in either direction; hierarchies are expressed with two axes, a "first-child" axis and a "next-sibling" axis, which traversed in reverse are "is-the-first-child-of" and "previous-sibling". To preserve his "ZigZag" trademark, Nelson recommends calling this kind of structure "Zzstructure". A straightforward implementation of it in memory just uses doubly-linked lists, like Sketchpad but without the circularity, or like the Z-machine generalized to many hierarchies, but without the constant-time traversal to the parent.

The fundamental operation of zzstructure is, as I understand it, to change the next or previous object of an object along an axis. If previously next-sibling(A) was B, then also previous-sibling(B) was A; if we set next-sibling(A) ← C, then next-sibling(A) will be C, and previous-sibling(C) will be A. What happens to previous-sibling(B)? I assume it must become nil, and similarly with next-sibling(D) where D was the former value of previous-sibling(C), if any.

So this operation need not allocate, and it cannot fail, but it can create garbage, because B and previous-sibling(C) may have become unreachable.

Pipeline and magtape processing

A popular alternative in some environments is to use "streams" or lazy lists rather than reifying the entire list in memory. So, for example, in Python you can compute a maximal nondecreasing subsequence of a sequence in "constant space" as follows:

```
def mnds(xs):
    xs = iter(xs)
    last = xs.next()
    yield last

    for x in xs:
        if x >= last:
            last = x
            yield last
```

One or another kind of iterator pattern like this is common to many languages — CLU had a special-purpose iterator construct, Smalltalk uses a general-purpose closure mechanism to implement it as a pattern, and Ruby bears the traces of having switched from the CLU approach to the Smalltalk approach, plus a little syntactic sugar. The C++ STL is famously based entirely on "iterators", but they use a totally different design which is not as amenable to streams. Python's approach is somewhat derived from Icon's, whch is derived from SNOBOL's, which is derived from backtracking for string processing. Prolog implementations can also generate a sequence of possibilities in constant space by backtracking.

This approach goes back to the earliest days of computing; not only did business data processing in COBOL typically work by reading one or a few records into the very limited memory of the time, but some early machines like Turing's Pilot ACE exposed the sequential nature of their delay-line memories. And in some sense, this is what's happening at the lowest levels of a CPU: the CPU registers are constant space, and they are used to lazily materialize sequences of values laboriously retrieved from main memory.

Constant-space lists

This may just be Sutherland's idea from Sketchpad, but what if our fundamental memory mutation operation is to move an object from one list to another?

Let's say a given type of object participates in a given set of intrusive lists, which we can treat as fields of the object. These lists are doubly-linked, so removing a linked object from one such list is easy, as is inserting an unlinked object before or after a linked object. Combining these two operations gives us an atomic move-between-lists operation.

This move-between-lists operation is constant-time and cannot fail. Can it create garbage? Yes, because the object you are moving may have been the only surviving external reference to the list you are removing it from.

(Heterogeneous lists presumably need some kind of run-time type identification information to get from the list header to the entire object, since different types of objects might have their list links at different field offsets.)

I give up on the microscopic view

I was hoping to find a set of one or more operations that would give me what I wanted: a way to manipulate flexible, irregular data structures in constant time without the possibility of failure and without creating garbage. Clearly you can write a program that does some kind of computation on flexible, irregular data structures in constant time without the possibility of failure and without creating garbage, and you can even prove these properties, but you I don't know how to do it by pushing those requirements down to the atomic level of the program.

However, although you need some kind of less-local proof to establish the safety of any of these approaches, the different sets of primitives have different sets of proof obligations. The Z-machine approach, for example, only requires that you prove that you aren't reparenting a node to be its own descendant; in many cases this is

easy, like when the node is a leaf node, or a node of a type that is statically known not to occur in the ancestors of the destination, or when the destination is closer to the root than the node being reparented. Similarly, the constant-space lists approach just requires you to prove that there's a reference somewhere else to the list you're removing the node from, or alternatively that it was the only node in the list.

By contrast, the immutable Lisp approach requires you to prove that there's enough space for the node you're allocating, perhaps because you preallocated it — also simple, but very different. It's very similar, though, to the kind of proof you need to do for constant-time code, where instead of proving an upper bound on the amount of allocation done by a function call, you prove an upper bound on the amount of time it can use.

Topics

Hot water bottles

Kragen Javier Sitaker, 2018-07-14 (4 minutes)

Sleeping with a hot-water bottle is an extremely efficient and relatively safe way to deliver heat to your bed — no heat is wasted on warming up the air, your mattress, or the walls. It just slowly leaks out through your blankets.

It's also relatively safe, in that it is very unlikely to cause a fire (unlike a malfunctioning electric blanket, space heater, or kerosene or wood stove). And it can effectively harness abundant low-grade solar heat, such as from flat-plate collectors.

A traditional approach is to fill a rubber hot-water bottle of 1ℓ or so with boiling water, which is somewhat dangerous; sooner or later the rubber will crack and the hot water will escape, and if this happens rapidly while the water is still near to boiling, it can seriously fuck you up. Also, the rubber hot-water bottle is somewhat expensive, so many houses only have zero or one of them, even in otherwise economically developed countries.

A cheaper alternative is to fill discarded PET bottles with hot water, well below PET's usual softening point of 90°. The PCO1810 and PCO1881 caps normally used on soft-drink bottles are easy to secure, resilient to pressure, abrasion, and impact, and very reliable, although for extended warm use, I trust the gasketless all-polypropylene caps more than the caps with a separate gasket inserted into the cap. The water need not be potable, although it's probably a safer situation if it is potable, in case someone drinks it by accident or in desperation.

The energy provided by a hot-water bottle is proportional to its mass and to its ΔT above your body temperature. Sooner or later, under the blankets, your skin will reach nearly 37°, so that's the reference point; a bottle at 38° has, in effect, half the energy of the same bottle at 39°.

Up to about 45° there is no real hazard from hot water under normal circumstances; if it does leak, it can burn you, but slowly enough that your instincts will probably protect you adequately. I think 50° is probably a good balance point between burn severity and energy capacity, even though PET can handle temperatures up to 80° with ease.

People generate about 100 watts each (2000 kcal/day = 97 W), so sleeping with a 100-watt hot-water bottle should warm you up about as much as sleeping with another person, which is pretty comfortable on nights ranging from cool to quite cold. More, perhaps, since it is hot enough for heat to flow from it into your body, rather than just stopping the heat loss from one side of your body.

So how much water would you need to be really luxuriously warm, say, 300W for 8 hours? That's 8640 kJ or 2063 kcal (which should be unsurprising, given the 1 person ≈ 100 W equation in the previous paragraph) which requires 159 kg of water at ΔT = 50° - 37° = 13 K.

This is a dismayingly large number, 53 3ℓ bottles. If you go up to 80°, you get down to 48ℓ, 16 3ℓ bottles, at the expense of potentially serious scalding if one of the bottles springs a leak. A 1ℓ bottle starting

at 100° only averages 13 watts over 8 hours, though it can provide over 100 watts for 1 hour if its insulation is thin enough.

I conclude that phase-change materials like Glauber's salt (is its freezing point high enough?) and active thermostats like the ones in waterbeds are a more practical ways to warm up your bed, and hot-water bottles are more a question of comfort than of actual temperature control.

Topics

Pipe dome

Kragen Javier Sitaker, 2017-07-19 (7 minutes)

The 20mm PVC pipe I tried to use for the Fuego Austral dome wasn't stiff enough to support much weight, let alone wind loading; it supported maybe 10kg above and beyond its own weight at the crossing of three pipes in the center of the six-pointed star dome. It was 1mm thick and cost AR$27 (US$1.80) for three meters. The largest pipes available in the category were 46.8 mm inside diameter, 50.8 mm outside diameter, and are thus 2mm thick: twice the wall thickness, 2.4 times the lever arm, and 2.4 times the perimeter, for a product of 11.5 times stiffer. I'm not absolutely sure, but I think it cost AR$95 for three meters, 3.5 times more cost (and thus 3.3 times more cost-effective at buying stiffness), and would have weighed 4.8 times as much per three meters. 11.5 times stiffer would imply it could carry about 110 kg of load at the center of the dome.

I paid some AR$700 for 26 three-meter lengths of the 20mm pipe. At AR$95 per three-meter length, this would be AR$2470 (US$164). Adding the AR$1450 cost of the 81 square meters of used advertising vinyl we bought to cover it, the total comes to some AR$4000 (US$263). This amount of vinyl is a bit of overkill: only 57 square meters should be necessary, costing only AR$1026. But then, the whole design is kind of overkill.

The longer lever arm means that the roughly 500mm radius of curvature I established in destructive testing in Parque Lezama would be about 1.2 m. This still seems like a pretty tight curve: a 2.4-meter-diameter circle is quite tight compared to the 6-meter diameter of the desired dome. Thicker pipe walls would help more with resilience; spreading the same material over more area would help more with stiffness.

If this amount of pipe encloses a 5-meter-diameter circle of effectively usable floor space, that's 19.6 square meters of floor space, at a cost of about US$13 per square meter.

How does this compare to the cost of a hexayurt?

A standard 18-panel H18 12-foot hexayurt is 8 feet (2.4 m) on each hexagonal side and 2.4 meters tall at the inside of the wall; I suppose that means it's 2.1 meters from the center to a wall, making its interior consist of six 2.1-meter-height, 2.4-meter-base triangles, for a total of 15.4 square meters of usable floor space.

http://www.appropedia.org/Category:Hexayurt_project says it costs "around [US]$300 per unit". It originally used 1" (25mm) Tuff-R foil-faced polyisocyanurate foam panels, according to http://www.appropedia.org/Hexayurt_playa#Which_Hexayurt.3F, but http://www.appropedia.org/Hexayurt_Safety_Information says that's a fire risk. Also apparently the necessary four rolls of tape cost like US$150, and FOAMULAR 150 panels of 1" thickness (without even the aluminum facer!) currently costs US$19 at Home Depot, for US$342 total cost for the foam and thus US$490 for the total hexayurt, US$31/m². This is about 2.4× the cost of the dome.

(Rmax Thermasheath-3 1-inch 4'×8' R-6 polyisocyanurate panels with aluminum facers cost US$19.25 each at the Emeryville Home Depot, so this is probably actually about the right cost.)

Another interesting comparison is the weight. The dome probably weighs about twice the weight of its vinyl coating, which is, say, 400g/m²; that's 22.8kg of vinyl in the dome, or about 46kg total, or 2.3kg/m² of usable floor. The Thermasheath-3 boards weigh 7 pounds each, according to http://www.homedepot.com/p/Thermasheath-Rmax-Thermasheath-3-1-in-x-4-ft-x-8-ft-R-6-Polyisocyanurate-Rigid-Foam-Insulation-Board-787264/100549260, which works out to 57kg, or about 3.7kg/m² of usable floor.

So, this dome is several times cheaper than a hexayurt and a bit lighter, but it might also be significantly less livable because it provides little to no insulation. A closer comparison might be a plywood hexayurt, which is US$132 for those same 15.4m².

A minimal-cost composite dome covering might consist of Tyvek, Mylar, and bubble wrap. I don't know if this can get near the cost of AR$18/m² (US$1.20/m²) that we're paying for used opaque billboard covering, but I suspect so.

http://articulo.mercadolibre.com.ar/MLA-611584214-cinta-de-aluminio-puro-de-30-micrones-ideal-para-aislacion-_JM offers 30-micron aluminum reflecting tape for AR$400 per roll of 45 m × 48 mm, a product of 2.16m².

http://articulo.mercadolibre.com.ar/MLA-612633055-membrana-burbuja-10mm-aluminizada-aislante-termico-_JM is 3.5mm-thick aluminized bubble-wrap membrane for insulating uses for AR$289, 1m × 15m, or AR$19/m². It says "10mm" in the description, but this is misleading; that's the diameter of the bubbles.

http://articulo.mercadolibre.com.ar/MLA-604333399-aislante-aislacion-termica-para-techos-rollo-x-25-m2-_JM is a roll of 25m² of bubble wrap (polyethylene, I suppose) without an aluminum facing for AR$121, or AR$4.84/m². No thickness is specified, but I suspect 3.5mm.

http://articulo.mercadolibre.com.ar/MLA-612656558-membrana-aislante-espuma-tipo-isolant-tba10-10mm-aluminizada-_JM is a 1m × 20mm roll of 10mm-thick aluminized flexible polyethylene foam for AR$530, or AR$26.50/m².

http://articulo.mercadolibre.com.ar/MLA-614120886-aislante-termico-burbujas-aluminizado-20-m2-_JM is 19.4m² of 3.5mm-thick aluminized bubble wrap for AR$320, or AR$16.50/m². This is probably the best bet so far.

Various space-blanket products on Mercado Libre are about 4m² for about AR$150, or about AR$40/m². It seems like aluminized Mylar ought to be available for less money, but I'm not seeing where.

One of the geodesic domes I saw at Fuego Austral was covered with silage plastic (for covering heaps of grains). This is perfectly opaque and airtight, and it's white to keep down solar heating of the grain pile. It seems likely to have a lower cost per square meter than

other reflective, opaque substances; http://articulo.mercadolibre.com.ar/MLA-614005553-bolsa-para-silo-agrinplex-9x60-f-forraje-segura-_JM is a bag of such plastic of 9 feet diameter and 60 meters of length, "Agrinplex" brand, for AR$7000. (Silobolsa and Siloplast are two other brands.) If we figure that 9 feet diameter means 28 meters of circumference, which is 8.6 meters, that's 517 m², or AR$13.50/m². If combined with bubble wrap, it might make an adequate dome covering. However, it's probably not as strong or stiff as Tyvek.

http://articulo.mercadolibre.com.ar/MLA-614441671-tyvek-rollo-130-x-12-m-_JM is a 1.3 m × 12 m roll of Tyvek for AR$479, or AR$30.70/m². This by itself is considerably more expensive than the billboard vinyl it replaces in this use.

http://articulo.mercadolibre.com.ar/MLA-614133525-wichi-roofing-igual-a-tyvek-membrana-hidrofuga-x-m2-_JM is a 1.16 m × 26 m roll of off-brand Tyvek clone ("Wichi" brand) for AR$484, or AR$16/m², about the same cost as the billboard vinyl.

http://articulo.mercadolibre.com.ar/MLA-606787652-tyvek-aislacion-hidrofuga-techo-superior-a-ruberoid-_JM is a 1 m × 30 m roll of Tyvek for AR$1050, or AR$35/m². It gives the weight of the roll as 2.8kg, or 93 g/m². This is something like 20% of the weight of the billboard vinyl (the vendors say 30%, but my tired back says otherwise, although maybe the vinyl was wet or something.) The publication clarifies that it's 100% HDPE and weighs 80.6 g/m² (maybe the other 12g/m² is the cardboard it's wound around) and is 220μm thick.

Topics

UHMWPE clothes could be lightweight and sturdy

Kragen Javier Sitaker, 2018-06-05 (3 minutes)

UHMWPE seems like it might be an interesting material for clothes, and indeed, there is a startup ("SheerlyGenius"? I forget) making sheer UHMWPE pantyhose — the idea is that they will be considerably sturdier than nylon pantyhose, which is somewhat surprising to me.

UHMWPE fiber has 2.4 GPa of tensile strength, apparently. You could imagine a cloth sort of like rip-stop nylon made from it, with somewhat thicker fibers every millimeter and very thick fibers every centimeter. Say, 3× and 10× thicker than the regular fibers.

I don't have a super great reference handy for the tensile strength of cellulose, but Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) says MDF's UTS is 18 MPa, which is probably low but in the ballpark. ASTM A36 steel has a yield stress of 290 MPa, HIPS has 32 MPa UTS, and annealed aluminum's yield stress is around 15–20 MPa.

So let's say cotton's tensile strength is 40 MPa. I have some sturdy cotton serge shorts here whose cloth is about 700 μm thick. Achieving similar strength with UHMWPE would require 40/2400 of the average thickness, or 12 μm; you could have 160 or so 6-micron-thick threads per millimeter, with an 18-micron thread each millimeter, and then a 60-micron thread each centimeter. This gives us an average thickness of roughly (60·60 + 18·10·18 + (10000-(60+18·10))·12)/10000 = 12.396 microns. Actually the extra 0.396 microns from the thicker and very thick threads should probably be doubled, making it 12.8 microns.

However, each 60-micron thread would break under a force of some 7 newtons, the weight of 700 grams. If you really want rip-stop strength, you likely need another approach. Something like knitting, for example, so that the force at the tip of a rip is distributed over more cloth, but ideally with knots frequently enough to prevent runs from spreading. Alternatively, you could scale the 60-micron threads up to, say, 300 microns — still half the thickness of the cotton, but now with a breaking force of 170 newtons, which would make the cloth unlikely to tear by accident.

Let's suppose that we knit some shorts from 20-micron-thick UHMWPE threads, then, and that the knitted cloth is about 60 microns in thickness and about 60 microns per row of knits or purls. This might require 1100 mm × 550 mm of cloth, totaling 36 cubic centimeters and about 25 grams. The 550 mm of length is a bit over 9000 rows of knitting. The tensile strength of a leg of the shorts as a whole would be something like 2.4 GPa · 20 microns · 550 mm · 0.5, which works out to 13 kN, a bit over a tonne. You could still cut the cloth with scissors, but ripping it might not be feasible.

To make this garment opaque, you'd probably need to mix in a substantial amount of some very opaque pigment, such as finely divided titanium dioxide, carbon black, or gold. If this is within the body of the fibers, which is advisable for durability, it will weaken

them, requiring a compensating increase in thickness.

Individual Dyneema fibers are supposedly 12–20 μm in diameter, as currently manufactured.

Topics

Quadratic opacity holograms

Kragen Javier Sitaker, 2015-09-03 (7 minutes)

I think I know how to print 140 to 200 pages of comfortably naked-eye- readable text and full-color images on a transparency foil using a regular 1200dpi laser printer with reasonable resolution and only a factor of 10 or 20 loss of contrast; you will see different pages of text from different angles.

In 2001 I came up with this idea of opacity holograms, which is a multi-image nonsecure version of Naor and Shamir's visual cryptography, and in 2007 I figured out how to extend Naor and Shamir's idea directly to grayscale. I've been thinking about using opacity holograms for low-cost text archival in order to take advantage of the high storage capacity of laser-printed paper without needing a microscope to read it; but a major problem has been that the technique I'd come up with, like lenticular 3-D, reduces image brightness/contrast/resolution by a factor of N to encode N images; while real holography, using interference patterns you can't plausibly print on a regular laser printer, only reduces these by a factor proportional to √N. So real holography has, so far, been vastly superior for encoding large numbers of images in a single image.

I finally think I have a √N technique, but I haven't tried it yet.

You have N low-resolution truecolor input images to encode, plus a bilevel "key" image at the resolution of your actual output device, which could be random but need not be; the crucial features of the "key" image is that its autocorrelation is very small at all small spatial shifts other than 0, and that its correlation to any input image at some small spatial shift is also very small. As the key is bilevel, each of its pixels is either black or white, which we will consider as -1 and 1. Its autocorrelation at small shifts being very small requires that it be almost exactly half black. When laser-printed on a transparency film, the black pixels of the key will obscure corresponding pixels of the output image, while the white pixels will remain transparent.

Each of the N input images is encoded with the key at a different spatial shift, one at which it is effectively uncorrelated with the key. Each encoded pixel is equal to the corresponding input pixel if the corresponding key pixel is white, or color-inverted from the corresponding input pixel otherwise: black for white, white for black, red for cyan, 50% gray for 50% gray, and so on. That is, we multiply the spatially shifted key by the input, considering medium gray as 0, black as -1, and white as 1. Since the "key" image contains very little large-scale variation in black-pixel density, each of the encoded images will appear entirely as a flat medium gray from a distance, or random noise, close up; but if either multiplied by or obscured by the key image, a close approximation of the input image is visible.

What's the distribution of that random noise? It's guaranteed to have a mean of medium gray, but beyond that, it could be anything from constant medium gray to a bimodal distribution where all the pixels are either white or black.

Then, we sum all of these encoded images (without saturating!) to produce a combined encoded image. From the perspective of any

individual encoded image, this amounts to adding Gaussian noise whose variance is proportional to the number of other input images (multiplied, I guess, by how far they tend to get from medium gray). So we can expect that the noise from adding 100 images will be only about 6× worse than the noise from adding 3.

Converting this into something you can actually see on the screen or print is a little tricky. The combined encoded image will probably have a nearly Gaussian distribution of color, which means it's barely using most of its dynamic range. To display on the screen, it's probably okay to lop off the tails of the distribution at a point that most (say, 99%) of the pixels are accounted for; but then dithering is a problem, because regular dithering will shift errors to neighboring pixels on the theory that your eye will kind of blur them together. But that doesn't work if your eye can't see the neighboring pixels because they're obscured by the key image. Fixed dithering would work, and so would random dithering, but you've already effectively done a bunch of random dithering by adding other effectively random images. So nearest-neighbor "dithering", which amounts to mere thresholding at the median in the case of a bilevel device like a B&W laser printer, is probably just fine.

You could print the key on one side of transparency film (preferably archival polyester, not acetate, which is prone to vinegar syndrome) and the combined encoded image on the other. The pixel shifts you can get with the parallax between the different sides of normal transparency film add up to somewhere around a hundredth of an inch total, so you should be able to get up to about, say, 12 to 14 by 12 to 14 (144 to 196) separate spatial shifts for the key; so you can encode that many different documents on a sheet.

The noise will reduce the contrast. If you have 144 documents adding together, the noise will be about 10 or 20 times larger than the "signal" of the input image you're trying to decode. This means that input features that are smaller than about 30 to 200 pixels will be swamped by the noise. 100 pixels (10 pixels square) at 1200dpi is a 120th of an inch square, which compares favorably to normal computer monitors and old dot-matrix printers, although it's lower-quality than we're used to seeing on laser printers. At that resolution, my 6-pixel-high font could still fit 20 lines of text per inch (a point size of 3.6 points), which is too small for normal people to read comfortably. Six lines of text per inch was traditional for computer printers.

You could also use this technique to share a single computer monitor between different people, or between the two eyes of a single stereoscopic user, or even to provide an easy hands-free way to shift between virtual desktops: mount the key image on transparency film some hundreds of microns in front of the monitor, and merge the video images for the different angles by using different shifts of the key.

Topics

- Archival (p. 3322) (34 notes)
- Microprint (p. 3582) (8 notes)

• Printing (p. 3649) (7 notes)
• Opacity holograms (p. 3607) (5 notes)

José, the Galician mover

Kragen Javier Sitaker, 2015-11-09 (2 minutes)

On my way home today (2015-11-09) in T-shirt and jogging shorts, I met an retired Galician furniture mover named José, digging a ditch in the sidewalk with a battered shovel under an overcast sky. Sweat dripped off his face, and he showed me his knee-replacement scar, and I felt his rock-hard Popeye forearm. There on the sidewalk, we talked about what motivates people to honesty and about the difficulties he had had with police demanding bribes when he was in business and the difficulties with employees. I told stories about Diogenes of Sinope.

He told me of a time, 40 years ago, when he was moving furniture for the former propaganda chief of the SS, who had retired to Argentina after the war. The man told him that propaganda could persuade brothers to kill one another.

He also told me of a time when he was 17, working as a mover for his father. After finishing up the job, he presented the bill to the customer, a friend of his father's. The customer said he would pay his father. He went home and told his father, and his father asked, "Why didn't you collect then?"

"I figured he'd pay you. He's your friend!"

"I don't have any friends!" laughed his father.

And, sure enough, the man never paid.

As I said goodbye to José, the lightning began, and then the hailstorm. I shivered in my soaked T-shirt in the freezing rain.

Topics

A unicast phased-array ultrasonic "radio"

Kragen Javier Sitaker, 2013-05-17 (4 minutes)

Suppose you set up a high-Q acoustic resonator strongly coupled to the air at, say, 102kHz, with its output connected to some kind of acoustic rectifier. If your Q was 20, you could tune in to about a 5100-Hz-wide band. This would be enough to receive and demodulate an ultrasonic AM signal with "telephone quality", i.e. low-pass filtered to about 4kHz.

It's feasible to focus a 102kHz ultrasonic signal in air to a spot about 0.3 centimeter across, or to transmit a low-loss collimated ultrasonic beam of such a frequency that's only a few centimeters across. This could allow substantial-distance ultrasonic AM communication through air despite the way that air attenuates high frequencies (about 1.5 dB/ft at 100kHz, and increasing linearly with frequency from 0.5 dB/ft at 50kHz up to 5 dB/ft at 250kHz). For example, if you started with a ten-square-meter dish or phased-array transmitter transmitting at 120dBa (1 W/m², 10 W total) and focused it on a square-centimeter receiver, you'd get an antenna gain of 50dB. If 40dB was an acceptable listening volume, and your "rectifier" was able to recover -10dB of the original signal, you'd need 50dB at the receiver, which means you could afford 120dB of attenuation along the signal path: 80 feet.

At this distance, your Airy disk radius angle ($1.2\lambda/d$) is about 1.2 * 0.003 m / √10 m = 0.001, which at 80 feet gives you an actual radius of 3cm, or 6cm diameter. So you're diffraction-limited by your transmitting antenna rather than scale-limited by the wavelength of the signal. This limits your actual antenna gain to 40dB instead of the 50dB in the previous paragraph, so you'd only be able to actually transmit about 75 feet.

If you could get by with a narrower-band signal, you could use a lower frequency. At 51kHz, where you could transmit three times as far with the same path attenuation (at the cost of less antenna gain, since your Airy disk diameter doubles with the longer waves and triples with the greater distance to 18cm, making it 36 times greater in area, bringing the maximum antenna gain down to about 25dB), your Q=20 receiver could handle a 2.6kHz band. If you were only transmitting speech, you could probably get by with that with a simple hack: frequencies over 2kHz in speech are almost always part of a burst of white noise, such as a sibilant. If you hook up a high-pass filter to the decoded signal and run its output to something nonlinear, you should be able to generate strong harmonics up to a few kHz, which would imperfectly approximate the high-frequency component of the original signal. (I think this is the reason that audio clipping in walkie-talkies improves comprehensibility.)

Being able to transmit comprehensible speech, to a passive receiver with no moving or electronic parts, anywhere in a 225-foot radius (comparable to Wi-Fi) sounds pretty cool, even if not a real improvement over just hollering. The receiver needing to be 36cm across would seem to somewhat blunt that, though, although you

could get a proportionally smaller receiver by making a proportionally larger transmitter.

With this level of spatial demultiplexing, however, you might not need frequency division multiplexing at all. Even at 26kHz, which ought to give you a greater transmission distance (450 feet?), your 10m² transmitter can focus down to a 1.5-meter-diameter spot. A larger transmitter could both transmit more energy and focus it on a smaller spot.

What if instead of transmitting the signal through free air, you transmitted it down a string, like a higher-tech version of paper cups connected with string?

Topics

Text editor design for e-ink displays

Kragen Javier Sitaker, 2018-10-28 (23 minutes)

(Previously published at http://canonical.org/~kragen/eink-design.)

NinjaTrappeur built the Ultimate Writer, a cypress TV typewriter for undistracted writing, and reports that writing in it is tolerable despite the e-ink display's 3-second update delay, but editing is intolerable.

If you want to design a text editor for low-power e-ink displays, you probably want to minimize the number of pixels you update per character (and especially minimize full-screen refreshes) and also avoid unexpected full-screen refreshes. That's because full screen updates take longer than partial screen updates and because they use a lot more energy.

This means that running vi or even ed on a standard terminal emulator is not going to work very well. Scrolling the whole screen up every time you want to display a new line at the bottom involves a waste of time and energy.

Variations of scrolling

You can get a substantial improvement over scrolling, without changing the underlying typewriter model of ed/ex, in at least a couple of ways: the ntalk/Tek4014 approach and the Emacs approach.

In ntalk, when your typing reached the bottom of your text window, you would wrap back around to the top, which would erase a couple of lines there. Erasing then proceeded one line at a time, always leaving a blank line below the text you were currently typing and above the oldest surviving text. No text ever moved; it remained until it was erased. This was a good match for the capabilities of "intelligent" terminals like the ADM3A which could put text anywhere on the screen but had no way of moving it once it was there.

The Tektronix 4014 did something similar, except that it had two columns (the screen was wide enough that typical lines of text fit into a single column) and couldn't do partial display erases — its "direct-view bistable storage tube" could only be erased all at once, although it permitted incremental drawing. So if you wrapped around to the first column again, you usually needed to request a screen erase in order to see more text.

In Emacs, when you reach the bottom of a window and try to go further down, it scrolls — but not by a single line. Instead, it scrolls by a whole half screenful. This is more expensive than the ntalk approach, and it does involve a full screen refresh (slow on either e-ink or the serial lines common when GNU Emacs was being designed), but it's a reasonable balance of efficiency and hysteresis:

• If you're typing lines one after another in this system, each line gets displayed twice: once when you initially type it, and a second time

when it scrolls up by half a screenful. After a second scrolling event, it's no longer visible. This thus implies a multiplier of 2× over the minimal possible expense of displaying and erasing the line once, as in the ntalk approach. This compares to 30× for the standard line-by-line approach, if the display is 30 lines high.
• If you're moving down, and then you decide to move back up, this does not require scrolling, unless you move up by an entire half-window-height. The maximum possible distance over which you could avoid scrolling would be a whole window height, which is what the usual scrolling algorithm provides.

So, the Emacs approach provides half the hysteresis of the best possible hysteresis, and half the efficiency of the best possible efficiency.

However, the problem with editing doesn't end with an efficient simulation of a teletype.

Editing and repainting

Screen editors like vim and Emacs display a window onto the document you're editing on your display and keep it constantly updated; the slogan at the time was "WYSIWYG", "what you see is what you get". (Later that slogan was repurposed to distinguish editors that did on-screen justification and multiple fonts, rather than displaying raw markup.) For changes that amount to adding or removing text at the beginning or end of the document, or overwriting text in the middle, the only required repainting of existing text consists of scrolling, which can be minimized as described above. But most changes in screen editors consist of inserting or deleting text in the middle of a document. In the WYSIWYG paradigm, this requires repainting a lot of existing text either after every new line or even after every keystroke, which is a major cost on these displays.

The pre-WYSIWYG editor paradigm was the "line editor" paradigm, exemplified by ed, ex, or EDLIN. Line editors are designed for teletypes: you type lines of commands at them, and they "print" lines of results back at you, always adding lines at the bottom. You could clearly use this approach with one of the "variations of scrolling" mentioned above.

However, it won't be very efficient, because sooner or later you'll want to see if you modified the text the way you intended. And the only way the line-editor paradigm has to show it to you is for you to "print out" some lines, including both modified and unmodified text. So you may end up redisplaying the same unmodified text several times within the same screenful of typeout, which is an inefficient use of e-ink, and furthermore of screen real estate, unless the edit history was really what you wanted to focus on, as opposed to the end result.

Here's a short example ex session, although it's using the somewhat ersatz implementation of ex included in vim. Note the abundance of redundant text repaints:

```
$ ex eink-design
"eink-design" [noeol] 108L, 5582C
Entering Ex mode.  Type "visual" to go to Normal mode.
:/efficient
However, the problem with editing doesn't end with an efficient
```

```
:n
E163: There is only one file to edit
:/
However, it won’t be very efficient, because sooner or later you’ll
:5p

:/However, it
However, it won’t be very efficient, because sooner or later you’ll
:.,+5p
However, it won’t be very efficient, because sooner or later you’ll
want to see if you modified the text the way you intended.  And the
only way the line-editor paradigm has to show it to you is for you to
"print out" some lines, including both modified and unmodified text.
So you may end up redisplaying the same unmodified text several times
within the same screenful of typeout, which is an inefficient use of
:
e-ink, and furthermore of screen real estate, unless the edit history
:
was really what you wanted to focus on, as opposed to the end result.
:


:
vi, as opposed to vim, was largely written on an ADM-3A, which didn’t
:
have any capability of moving around text that was already on the
:
display; the best you could do was to
:
E501: At end-of-file
:a
.
:p
display; the best you could do was to
:s/$/ redraw it in a new place./
display; the best you could do was to  redraw it in a new place.
:s/  / /
display; the best you could do was to redraw it in a new place.
:a
Perhaps as a result, in vi, when you use the "c" change command,
which deletes a specified span of text and puts you in insert
mode to type new text to replace it with, vi doesn’t delete the
text on the display, shifting the following text to the left.
Instead, it marks the end of the deleted text with a `$`, and
no text moves unless you insert more text than was deleted.
.
:.-10,$p
was really what you wanted to focus on, as opposed to the end result.

vi, as opposed to vim, was largely written on an ADM-3A, which didn’t
have any capability of moving around text that was already on the
display; the best you could do was to redraw it in a new place.
Perhaps as a result, in vi, when you use the "c" change command,
which deletes a specified span of text and puts you in insert
mode to type new text to replace it with, vi doesn’t delete the
text on the display, shifting the following text to the left.
```

```
Instead, it marks the end of the deleted text with a `$`, and
no text moves unless you insert more text than was deleted.
:wq
wq
"eink-design" 114L, 5987C written
```

Blanks and strikethrough

vi, as opposed to vim, was largely written on an ADM-3A, which didn't have any capability of moving around text that was already on the display; the best you could do was to redraw it in a new place. Perhaps as a result, in vi, when you use the "c" change command, which deletes a specified span of text and puts you in insert mode to type new text to replace it with, vi doesn't delete the text on the display, shifting the following text to the left. Instead, it marks the end of the deleted text with a $, and no text moves unless you insert more text than was deleted. It doesn't even remove the deleted text from the screen until you leave insert mode. This reduces repaint traffic considerably on terminals like the ADM-3A.

On an e-ink display, something like vi's approach here might be useful, but we could carry it further. For example, if you start inserting into the middle of a line, we could open up a big blank for you to type in, maybe half the length of the line. It might look like this, using | for the cursor (which wouldn't really take up space) and displaying successive screen states on successive lines:

```
On an e-ink display, something like| might be

On an e-ink display, something like |_____________________________________ mighoot be

On an e-ink display, something like vi's ap|_____________________________ mighoot be

On an e-ink display, something like vi's approach h|_____________________ mighoot be

On an e-ink display, something like vi's approach here|__________________ mighoot be

On an e-ink display, something like vi's approach here__________________ mi|goht be
```

This way, there's a single erase and repaint of "might be" at the beginning of the process, and perhaps another one later if you type enough into the blank, followed by a final one when the system decides you're done editing there, perhaps because you moved to a different line, or started inserting text somewhere else:

```
On an e-ink display, something like vi's approach here might be
```

For deletion rather than insertion, you could use strikethrough analogously to avoid moving other text around, as Microsoft Word does in Track Changes mode. If the text previously read "something like this might be", and the user deleted "this", it might display at that

point as follows:

```
On an e-ink display, something like ~~this~~| might be
```

The horizontal insertion space may not be adequate once text has to move vertically as well as horizontally to accommodate your edit. In that case, you might want to do something analogous vertically: open up half a screenful of blank space in order to insert new text into, expanding it a half-screenful at a time. You'd probably want to mark it somehow to indicate that it "wasn't real".

This scheme is fairly close to the buffer-gap scheme Emacs uses internally to represent its text buffers. In order to avoid having to move text around in RAM after every character inserted, it maintains a gap after the last character inserted; whenever a character is inserted somewhere else, it moves the gap by copying text as necessary from one end of it to the other. It normally maintains only a single gap, and it can be arbitrarily large, since it doesn't need to worry about keeping the two ends of the gap close enough together that you can see them at the same time.

Double-spaced text

A disadvantage of all of the above approaches is that *any* text insertion in the middle will involve *some* repainting of text in order to open up a space for it. The traditional typewriter-era approach to this problem was to type the manuscript originally "double-spaced", with a blank line in between each pair of lines of text, in order to allow sufficient space for markup to include text to be added. With this approach, the above-explained edit would start looking like the following:

```
text from the screen until you leave insert mode.  This reduces

repaint traffic considerably on terminals like the ADM-3A.

On an e-ink display, something like might be

useful, but we could carry it further.  For example, if you start
```

And the insertion might leave it looking like the following, with a caret positioned below the insertion point and the inserted text above:

```
text from the screen until you leave insert mode.  This reduces

repaint traffic considerably on terminals like the ADM-3A.

                                    vi's approach here
On an e-ink display, something like might be
                                   ^

useful, but we could carry it further.  For example, if you start
```

This approach avoids any need to redisplay existing text to accommodate small insertions.

Another approach along the same lines would be to pop up a speech balloon or something over the existing text to contain the edit,

although this obscures some of the existing text. For example:

```
text from the screen until you leave insert mode.  This reduces
repaint traffic considerably on terminals like the ADM-3A._____
                                    |vi's approach here|_______|
On an e-ink display, something like|/ight be
useful, but we could carry it further.  For example, if you start
```

For grayscale displays, it might help to make the popup balloon translucent, display the text within in a larger size, and filter the text to soften its edges, all with the intent of leaving the obscured text readable.

Cursor display and movement

A separate problem is that the long display latency makes it hard to tell what your cursor is pointing at when you're moving it around in text. This is a problem I have a lot of experience with in vi, which I used to use routinely over modem connections with tens of seconds of latency, and I still use routinely over Tor connections with seconds of latency. vi was originally developed under conditions that often included seconds of latency due to not only modem bandwidth but also host machine load, and its command set copes well with this situation.

The basic problem is that, under these conditions, you can't position the cursor effectively using arrow keys (or vi's hjkl), and you can't delete text with the backspace key, especially if you're using key repeat ("Typematic", to use IBM's trademark), because those depend on closed-loop control. The long and unpredictable latency in the feedback control loop between your fingers and your eyes means that you face a painful tradeoff between long settling times and overshoot: if the cursor hasn't reached the desired point and you press the arrow key one more time, you may have caused it to move past the desired point if you didn't wait long enough.

The solution is to use movement commands that are *convergent* and at a sufficiently *high level* to not depend on tight closed-loop control. "Convergent" means that commands cause similar editor states to converge to a single editor state, so initial divergences between the editor state and the user's belief about it, due to the latency, will not compound over time. So, for example, the `{` command moves back by a paragraph. If you believe you're on the third line of the paragraph, but you're actually on the line below the paragraph, it will still take you to the same point before the beginning of the paragraph, eliminating your error. The Vim command `ci(` similarly replaces the entire contents of the `()` parentheses you are within with whatever you type next; it doesn't matter if those parentheses contained 20 or 25 characters, or whether you were on the first or tenth character of the parenthesized expression, the same text gets replaced either way. At a more trivial level, the

These commands, unfortunately, impose a substantial cognitive load and a great deal of practice to use effectively. Jef Raskin's design of "LEAP", a single quasimodal movement command, enjoys the advantages of being strongly convergent and fairly high level, but with a much lower cognitive load.

Briefly, LEAP is similar to the incremental-search found in Emacs and Vim;

Raskin's Canon Cat keyboard has two LEAP keys to be pressed with the thumbs, one on the left for searching backwards and one on the right for searching forwards, which initiate the search when the user begins pressing them and terminate it when the user releases them. In either case, the user is left at the beginning of the matched text, rather than the end, so there is no penalty for typing more characters than needed, and no special provision is needed for special characters such as newlines.

Raskin claimed that in user tests LEAP was substantially faster than using a mouse to navigate text. I haven't done the rigorous tests he claims to, but from my experience, this claim seems plausible to me.

The Cat had no ↑↓ keys, just ←→ "creep" keys and the LEAP keys, the idea being that if the user was moving far enough to need ↑↓, they'd be faster if they didn't have the cognitive load of making a decision between arrow keys and LEAP.

E-ink latency is long, but it's shorter than the kinds of latency vi can cope with. I think LEAP might be a good way to position a cursor quickly on an e-ink display without needing to see the cursor in order to know precisely where it was.

Deletion is more problematic, since LEAPing from one end to the other of the text to delete is usually going to be far too much overhead.

I routinely use ^W in terminals and vim, and Alt-⇐ or Ctrl-⇐ in Emacs, to delete the entire previous word; this is generally much more efficient than deleting just the incorrect letters and retyping them, and often more efficient than going back a word or two to insert a corrected word. It's often easier just to retype the last few words.

Explicit screen refreshes

When I first used AutoCAD (2.14K+ADE), it was on an IBM PC-XT. The XT was capable of executing about 250,000 16-bit instructions per second, with a 320×200 CGA graphics display and an 80×25 MDA text display side by side. Redrawing the entire contents of the CGA monitor from the in-memory CAD drawing typically required a second or two. Therefore, it was undesirable to perform a redraw after every mouse movement, or even after every new line or arc added to or removed from the drawing. Consequently, AutoCAD on this platform would update the display with an approximation of the change; rubberband lines and arcs for the mouse were drawn with XOR so they could be quickly erased, newly drawn lines would overwrite whatever other lines were on the display (even if they were in a layer below them), and deletion of lines and arcs was reflected by drawing them in the background color, leaving visible holes at any intersection. These operations could be done many times per second, permitting a real-time interactive feel.

(A more difficult problem was choosing points on the screen with the mouse, since the display resolution was insufficient to make anything other than very crude drawings by eye. Typically I would specify INT or TAN or whatever on the keyboard, and then point with the mouse to indicate the intersection of which things, or tangent to which arc; or I would directly enter relative coordinates in polar or rectangular form.)

When the damage to the drawing on the screen made it sufficiently difficult to see what was going on, you would issue the REDRAW command from the keyboard, which has the merit of being possible to type without removing your right hand from the mouse. There were

other operations that also required a redraw, notably zooming and panning the drawing.

(This kind of pragmatic and very clever compromise was what made it possible for Autodesk to sell a usable CAD system in the early 1980s on hardware costing a tenth of what existing CAD systems cost.)

Since a screen refresh on the Ultimate Writer takes some 3000 milliseconds, it's in the same ballpark as this PC-XT with AutoCAD, so it might be worth using a similar approach to screen updating: do a full-screen refresh only when the user requests it explicitly, or when you're displaying most of a screenful of new information.

Several of the choices suggested above include a non-WYSIWYG display of edits: line-mode editing, strikethrough, speech balloons, double-spaced interlineal markup, large insertion blanks, and so on. The idea is that these reduce the visual feedback latency from the 3000 milliseconds NinjaTrappeur is suffering now to a mere 600 or so , so that you can reasonably do edits. But it should be easy for the user to request a WYSIWYG redisplay whenever they want to see the final result of the edit and are willing to wait the requisite seconds.

Topics

Golang bugs

Kragen Javier Sitaker, 2018-09-13 (updated 2018-10-28) (6 minutes)

I thought I'd try to keep track of the bugs I have in my Golang programs, including compile errors. These aren't bugs in Go, but bugs I wrote in Go.

• Wrote = instead of := (7×)
• Wrote types before variables instead of vice versa (5×)
• Forgot to `import "io"` (3×)
• Forgot to discard the index when iterating over a slice `range` (2×)
• Incorrectly thought `regexp.Compile` returned a `Regexp` rather than a `*Regexp` (2×)
• Accidentally referred to the variable `name` as `string` because it was declared as `name string`, and the variable `data` as `byte` because it was declared as `data []byte` (2×)
• Tried to quote a string with apostrophes (2×)
• Left off a `return` at the end of a function that had already assigned its return values (2×)
• Tried to pass a string instead of a []byte to net.Conn.Write (2×)
• Used `%#v` instead of `%+v` even though I didn't want hexadecimal printing of my struct fields (2×)
• Forgot to `import "log"`
• Forgot to `import "strconv"`
• Forgot to `import "fmt"`
• Imported "os" unnecessarily
• Imported "bytes" unnecessarily
• Imported "io" unnecessarily
• incorrectly assumed `print()` output goes to stdout, not stderr
• Forgot to include quoted spaces around variable values in `print` argument lists
• Tried to pass a `[]byte` to `print`, which rendered as hex garbage
• Tried to call `ioutil.ReadFile` with a file instead of a filename
• Expected `ioutil.ReadFile` to return a string instead of a []byte
• Tried to use `ioutil.ReadFile` on a 1GB file on a machine with only 4GiB of RAM
• Left out a `range` in a `for`
• Called `Regexp.FindAll` with no count argument
• Confused `Regexp.FindAll` with `Regexp.FindAllIndex`
• Left "\n" off a `print` call
• Forgot to cast `int` to `int64` explicitly
• Referred to `File.Name` instead of `File.Name()` in a `print`, printing out a hex string
• Failed to capitalize `os.Args`
• Forgot to handle errors on `suffixarray.Index.Write`, which the compiler doesn't catch
• Failed to capitalize `File.Read`
• Didn't remember the server needs to start out an RFB conversation by sending the version banner
• Did `copy(result[8:10], ...)` and then `result[11] =..` not noticing the skipped item 10
• Tried to use a `bytes.Buffer` instead of a `*bytes.Buffer` as an `io.Writer`

• Tried to call my `putTo` method, which returns an error, as if it were `Write`, returning two things
• Tried to ignore only one return value from `bytes.Buffer.Write` instead of two
• Tried to call `panic()` with multiple arguments as if it were `print()` or `log.Fatal()`
• Forgot to cast len(name) to uint32 in a field initializer
• Left off a trailing comma in a struct initializer
• Wrote nonexistent variable names `width` and `height` rather than the constants I intended
• Called `log.Fatal` instead of `log.Fatalf`, which produced a particularly confusing error because the last argument happened to contain a "\r", overwriting most of the error message on the screen
• Didn't check the error return from `binary.Write`, since I was writing to a `bytes.Buffer`, but it was actually trying to tell me, "binary.Write: invalid type *main.NServerInit".
• Tried to use `binary.Write` on a struct whose fields weren't exported (no, that isn't a bug, never mind; the bugs were those below)
• Tried to use `binary.Write` on a struct containing a pointer to another struct, instead of containing the other struct by value
• Tried to use `binary.Write` on a struct containing a `[]byte`
• Oh actually you *do* need to export the struct fields for `encoding/binary` after all! But for `binary.Read`, not `binary.Write`.
• Tried to initialize a three-element struct with just one element positionally, thinking that the other elements would default to zero; I guess I have to use named struct fields for that?
• Named a constant `type_SetPixelFormat` and then tried to `case SetPixelFormat:`
• Declared some byte-count-return variables that I didn't use because in one case `Reader.Read` was reading into a buffer of size 1, and in the other case, `io.ReadFull` is guaranteed to read the right number unless it fails (which I was handling already)
• Used `%+v` instead of `%#v` even though I did want hexadecimal printing of my struct fields
• Tried to declare two variables of different types with a comma in between, in a `var` statement
• Tried to use a member variable in a method without a preceding `self.`, as if I were in Java or C++
• Used `log.Printf` instead of `log.Print` and got this `%!(EXTRA)` error in my log message: 2018/09/13 19:51:58 closing %!(EXTRA *net.TCPConn=&{{0xc82006a0e0}}, string= because of error ,* errors.errorString=EOF, string= in , string=message-type)
• Forgot the first argument of `io.ReadFull`
• Got the syntax for a map literal wrong; it isn't `var x map[foo]bar = { ... }` but `var x = map[foo]bar { ... }`
• Forgot to cast a uint16 to int in order to compare it to an (implicitly declared) int
• When aping a log.Printf() line that printed a variable, aped the variable too and consequently didn't print out the value I wanted
• Left out an argument to `log.Printf` and got a `%!q(MISSING)` error in my log message
• Tried to change the type of parameters by shadowing them with `x, y := uint32(x), uint32(y)` at the top level of a function, which totally doesn't work

• Tried to use an int as a uint16
• Wrote := instead of =, unintentionally shadowing an outer-scope variable with one of a different type
• Passed a struct to `binary.Read` by value instead of passing a pointer, which resulted in an error not caught until runtime (which caused the VNC server to drop the connection)
• wrote `format.big_endian_flag` when I meant `format.Big_endian_flag`
• tried to use a byte as a boolean value
• tried to call `string.SplitN` instead of `strings.SplitN`
• tried to pass '\n' instead of "\n" to bufio.Reader.ReadString

Topics

• Programming (p. 3658) (286 notes)
• Golang (p. 3477) (7 notes)

Transmission line diode computation

Kragen Javier Sitaker, 2016-07-30 (3 minutes)

Consider a transmission line shunted at some point with a diode and an ac pulse (a wave packet, say) propagating along it. When the pulse reaches the diode, if the diode is forward-biased, the wave packet will see a short and will reflect back, inverted. On the other hand, if the diode is reverse-biased, the wave packet will see an open circuit (with perhaps some extra capacitance, which can be mostly compensated for by reducing the distributed impedance of the transmission line or by adding inductance) and will pass through unimpeded.

Perhaps by this means you can compute the bitwise OR of two bitstreams stored in a transmission line with a diode, by looking at the signal past the diode; and perhaps you can calculated their bitwise AND by isolating the reflected signal.

Off-the-shelf PIN diodes like the US$2.73 (US$1.27 in quantity) M/A-Com MADP-011027-14150T operate at up to 12GHz. So you could imagine doing this operation on bits that were 13mm long traveling through a transmission line at 0.5c.

With an ordinary diode, unfortunately, this provides no way to amplify the signal.

Specialty parts like Gunn diodes (available as USSR new-old-stock on eBay for US$8–20) can operate at higher frequencies, up to some 40GHz in some cases (supposedly up to 200GHz in some devices I haven't seen for sale), and additionally have a negative differential resistance region which can serve to amplify signals. Supposedly, gallium nitride Gunn diodes can reach 3 THz. (Tunnel diodes, which are also specialty parts, are also a possible option, although they do not reach such high frequencies; they are available on eBay at much lower costs, like US$1.)

These frequencies are far, far higher than any transistor.

Suppose that you found a way to do universal computation with some kind of network of transmission lines and 3 THz Gunn diodes. You could very reasonably use 100 meters of coaxial transmission lines in a desktop-sized device. At 3 THz and a transmission speed of 0.5c, the transmission lines would contain about 200 million oscillations at any given time, each about 50 microns long. You could imagine this waveform containing 400 million bits. With a reasonable number of bitwise computing elements, such as 32, the device would perform 96 trillion bit operations per second. A Skylake CPU with three 256-bit arithmetic units might perform 1536 useful bit operations per cycle (if we count an add with carry as two bit operations), which would be 6.144 trillion bit operations per second at 4GHz. So such a device could be computationally useful even without integrated circuits.

Topics

A nonscriptable design for the Wercam windowing system

Kragen Javier Sitaker, 2018-10-26 (updated 2018-11-13) (6 minutes)

A simpler design as an alternative to the latency-minimizing design in the Scriptable windowing for Wercam (p. 1256).

Modern CPUs are fast enough that we ought to be able to get by, at least for many applications, without any kind of graphics acceleration at all. The role of the window system in such a system is just to multiplex screen space among different windows, composite the windows, and route events to the relevant applications.

The simplest approach — the same one used in xshmu — is to let each application draw on a window buffer, and when it's done, display that window buffer.

Let's assume that window flicker is intolerable, that latency is important, and that more enough RAM is available for several framebuffers. Then we need to ensure that the display is never reading a partially-updated window buffer; at any given time, either the display or the application owns the buffer. Since latency is important, we would like to initiate the redraw as late as possible before the vertical synchronization, so that the window contents can get composited into the framebuffer only 8ms before they're drawn (assuming 60Hz) instead of, say, 24ms.

But Wercam can't ensure that the application finishes drawing into the buffer and relinquishes ownership in time for compositing. Even if it could forcibly steal the buffer away when the deadline passed, it would have a partially-drawn window image to work with. So it needs a backup plan — it needs the previous contents of the window.

It isn't good enough to keep the previous contents of VRAM, because the window may be translucent and something *underneath* it may have changed. In fact, I would like to ensure that this happens as often as possible.

So we need at least two buffers for each window — the current contents and the previous contents in case the current contents are unavailable when the deadline passes. When we get the current contents back, we can relinquish the other buffer back to the application so that it can draw the next frame when the time comes.

The Porter–Duff **over** operation — the one we'll use for compositing translucent windows — involves a multiply-accumulate per pixel component, typically in 8-bit integer arithmetic. I think my Intel Gen8 GPU (see Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033)) can do about 100 billion 16-bit floating-point multiply-accumulates operations per second (400 MHz · 128 FP32 ALUs · 2 16-bit ops per ALU), and my CPU can do 128-bit SIMD (32 bytes) on I think four cores at 1.6 GHz each, which works out to 204.8 billion such operations per second, or 52 billion pixels. My screen draws 1920·1080·60Hz = 124 megapixels per second. This suggests that the CPU should be able to handle on the order of 419 layers, or the GPU half that (although the GPU also has a special-purpose blitter, which is probably competitive with the CPU.)

If the depth of translucent windows starts approaching this limit, it should be possible to request updates from windows deeper in the stack less frequently, perhaps every other frame. Then the foreground windows only have to be composited with the pre-composited background window stack.

We can go even further in this direction since the "over" operation is associative. We can group the windows by Z-order in groups of, say, 3, and composite each group separately, then composite the groups in supergroups of 3, and so on, until we have composed the whole scene. Then we can update any subset of windows, regardless of where they are in the Z-order, with a relatively small logarithmic cost. (But using saturating arithmetic might violate this associativity.)

The Dep kernel (see Speculative plans for BubbleOS (p. 2128)) provides the application and the window system the facility to securely transfer the window buffers back and forth such that each can be sure that the other has relinquished access before it starts to write. (When running Wercam on other platforms, we just have to hope.) Normally, the application doesn't allocate window buffers; it lets Wercam do that. If the application doesn't have possession of a buffer, it has nowhere to draw, and this is one way Wercam can limit the frame rate of background windows when necessary, or indeed eliminate the frame rate of invisible windows entirely. Also, though, normally applications will wait for a paint event from Wercam before drawing and sending a new frame. And they may not send a frame for a long time.

In the degenerate case where no windows are being updated, Wercam could avoid spending CPU time on compositing entirely; similarly with no windows being updated in a certain part of the screen. But its design goal is good worst-case performance, not good average-case performance.

Performance tests

I write a simple dumb alpha-compositor test in C. It appears to work, and it runs smoothly. At 828×512, just drawing a background, it runs at 1.13–1.16 ms (user CPU) per frame. Alpha-compositing an (opaque) copy of the background on top of itself, it runs at 4.23–4.31 ms (user CPU) per frame, implying a cost of 3.07–3.18 ms of compositing per frame, or 7.2–7.5 ns per composited pixel. Some preliminary analysis with Cachegrind finds 0.9 instructions per pixel without compositing plus 29.7 instructions per pixel added with compositing, but almost no difference in D-cache misses (0.125 per pixel with background only, 0.127 with compositing).

A little work with GCC vector extensions gets this down to, if my tests are valid, 1.46 ns per pixel, which is fast enough to put 5.5 layers on the screen, or 22 layers on all four cores.

Topics

- Performance (p. 3621) (149 notes)
- Systems architecture (p. 3691) (48 notes)
- Graphical user interfaces (p. 3489) (23 notes)

Framed-belt DSP

Kragen Javier Sitaker, 2018-04-27 (3 minutes)

Framed-belt-per-sample-rate signal-graph DSP on CPU

Lots of DSP stuff (time-domain FIR filters, IIR filters, Goertzel filters, PLLs, a lot of music synthesis stuff) can be formulated as computing a bunch of quantities for each input sample. The quantities may depend on the input sample, the current values of other quantities (let's call them variables), or even past values of other quantities at some fixed lag into the past.

For a given sample rate, I think you can store these quantities very efficiently in a ring buffer of "frames". Each frame has a specific fixed offset for each variable, at least those that are used in the future; they are like structs or function stack frames. A piece of straight-line code computes the new contents of all of the variables of interest, storing them into their appropriate places in the frame; it can index off a frame base pointer to refer to other values in the current or previous frame (although hopefully it already has those in registers) or even earlier frames.

When the ring buffer is close to being full, you must write each new frame in two places: a place near the end, and a place near the beginning. In this way, when you need to reset the frame pointer to near the beginning of the buffer, you don't need to copy a bunch of frames at that point, and your indexing operations don't all need bitmasks on them in order to make the ring buffer circular. The buffer must be at least twice the size of your processing window (the furthest into the past you ever need to reach, e.g. the number of FIR taps) for this to work.

Perhaps there might be some values computed that you don't save in this belt of frames, thus reducing the space needed.

FIR filters implemented in this way will access memory with a larger stride than FIR filters implemented in the normal way, but I'm not sure that will cause memory bandwidth problems with modern CPUs.

Any fixed topology of single-rate signal transformers can straightforwardly be transformed into a straight-line piece of code that works in this way.

For image processing, where you generate a frame per pixel, you might want to arrange access to belts for previous rows as well. The idea is that any reasonable (Δx, Δy, varname) tuple maps to some constant and reasonably small offset from the current frame base pointer. Tiling might make this more feasible, by preventing the Δy aspect from generating unreasonably huge offsets.

Topics

A comparison of prices for different forms of energy

Kragen Javier Sitaker, 2007 to 2009 (2 minutes)

Different ways of buying energy, and their costs.

A pound of rice (453g) costs about US$0.35, or 0.08 cents per gram. Each gram is about 5 kilocalories, or 21kJ, so in the form of rice, you can get energy for about 27 megajoules per dollar. If a 2000-calorie-per-day person got all their energy from rice (which would result in dying of kwashiorkor) they would use 60 000 calories in a month, or 12 kg of rice, or 26.5 lbs., which would cost US$9.27 at the above rate.

Vegetable oil is slightly cheaper as a source of calories, but it's in more or less the same range.

Protein is considerably more expensive; pure protein costs around US$7 per pound, but there are natural foods that are 25% protein by weight, such as pinto beans, which cost about the same as rice, with the remainder being carbohydrates. This means you can meet your protein needs without increasing the cost of your food from the all-rice diet above.

Of course, you need other nutrients to survive as well, but none of them in multiple-gram-per-day quantities, and none of them are things you derive energy from.

However! Both rice and beans require extra energy to cook them --- minimally, enough to heat them and a similar mass of water from ambient temperature to near boiling. If that's from 25C to 100C, that's 75 calories per gram of water --- figure maybe 100 calories per gram of food. But that's only 0.1 kilocalories, or an additional 2% "tax" needed on top of the energy in the food.

Around here, gasoline currently costs about US$3.00 per gallon. Each gallon is about 130 megajoules, so in the form of gasoline, you can get energy for about 43 megajoules per dollar.

Electricity in the US costs between US$0.06 and US$0.15 per kilowatt-hour, with an average just under US$0.10. A kilowatt-hour is 3.6 megajoules, so in the form of electricity, you can get energy for about 36 megajoules per dollar.

These costs are surprisingly much more similar than I expected.

Topics

Microsoft Windows uses \ for filenames because OS/8 programs used / for switches

Kragen Javier Sitaker, 2019-05-25 (2 minutes)

A response to Michal Necasek's "Why Does Windows Really Use Backslash as Path Separator?", also posted on the orange website.

This is almost correct, but MS-DOS derives from the DEC small systems line (the PDP-8 and PDP-11), not the large systems line that ran TENEX, which took the name "TOPS-10" in the DEC Witness Protection Program (the PDP-6 and PDP-10). TENEX's only real descendant in modern computing systems is the command-line editing (and filename completion?) in bash and zsh.

Much to my surprise, it seems to be true that CP/M did not actually use / for flags, even in PIP, although other incarnations of PIP (like that in Heath's HDOS) did use /. The CP/M 2.2 manual is at http://www.cpm.z80.de/manuals/cpm22-m.pdf and documents the command lines of all the standard utilities, including the assembler, PIP, and the text editor. (It also, in passing, documents the full 8080 instruction set and OS API.)

TENEX was born in 1969 but grew up in the 1970s, but the use of / for switches in DEC-land predates it; https://en.wikipedia.org/wiki/Concise_Command_Language is somewhat confused, but it currently describes how the PDP-6 monitor program used / for switches in, presumably, 1964.

DEC operating systems like the ones CP/M aped used / liberally for switches, and third-party programs we used on CP/M certainly did use / for switches. This was not limited to the PDP-10 large systems operating systems; it was also true on OS/8 for the PDP-8, as described in https://www.pdp8.net/os/os8/index.shtml (though, as you can see, some commands used - instead, like the later Unix). The PDP-8 shipped in 1965, but OS/8 might be more recent than that.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- History (p. 3500) (71 notes)
- Tenex
- Pdp 8
- Pdp 11

Notes on circuitry for the Nutra seed activator

Kragen Javier Sitaker, 2018-08-16 (20 minutes)

The Nutra has a fan (I'm guessing something like a 12V PC cooler), a ¼HP agitator motor, a small pump motor, a water inlet valve, and a heating element.

Lucía tells me the target is for a year's lifetime.

I'm thinking that probably the control solution for these is to hook up an STM32 to a ULN2003 or something similar, then use the ULN2003 to run some relays. None of them needs subsecond response times; I think they are each turned on and off less than ten times during a cycle, which lasts roughly a day, so a lifetime of 20000 openings/closings and an MTBF of 100 000 openings/closings is probably adequate. Most of them will run on line power rather than a lower voltage, so the ULN2003 can't handle them directly.

Heating element

I back-of-the-envelope estimated the heating element at 300 W, but maybe 600 W is a safer level (in the sense of "we are sure it will work", not in the sense of "less likely to cause injury in case of malfunction".) 600 W is about 2.6 amps at 240 VAC (RMS), so a 3-amp relay is probably fine.

But wait, this needs to be PWM-controlled. PWM through a relay is not a good idea; if the PWM cycle is 20 seconds in length (50 mHz), 100 000 openings and closings only gets us to 555 hours of dehydration time. An SSR would achieve higher levels of reliability, but 500 hours is probably already a year, since the overall cycle might have an hour or two of dehydration in it.

Power

The most challenging objective is dehydrating 5 kg dry weight of soaked, activated beans to more or less their original dry weight. They will have absorbed roughly 5 kg of water. Water's enthalpy of vaporization is about 2.4 MJ/kg, so this is 12 MJ which get turned into latent heat. Doing this in one hour would require 3000 watts, which would be 12 amps. At my previous guess of 600 watts it would take five hours, which might be long enough for the beans to rot.

If we could use an air-countercurrent recuperative condenser like the ones in modern condensing clothes dryers, we could perhaps reduce that somewhat, but I'm not sure how much. (There are also heat-pump dryers, basically air-conditioner-based dehydrators in a box, which cost an extra US$1200 or so.) Traditional vented clothes dryers use 4–6 kW and vent 100–225 cfm, or more broadly 3–9 kW, while non-heat-pump condenser dryers typically use 2 kW, and heat-pump condensers use ≈ 1 kW.

Let's say we don't have a recuperator, but 2 hours is adequate, so we can get by with 1500 watts. And let's say the air stream is limited to 40° in order to make sure we don't heat the food past 42°. My psychrometric chart doesn't go to 100% humidity at 40°, but it says that 30 mg H_2O / g air is 64% humidity at 40°, so I guess 100% is

about 47 mg H_2O / g air, and 95% would be about 44 mg H_2O / g air. This means that removing 5 kg of water vapor would require passing about 114 kg of dry air through the beans, or a bit more, since the air in Buenos Aires already contains something like 10 mg H_2O / g air.

Air is about 1.23 kg/m³, so 114 kg of air is 92 m³. This is enough air to fill a 35 m² room of 2.6 meters height, and we're proposing to turn it all into the kind of fog that fills your bathroom after a shower. So you really, really need to vent it outside or run it through a condenser.

Either way, those 92 m³ of air need to be blown through the dehydrator during those 2 hours, which works out to about 12.8 ℓ/s, 770 ℓ/minute, or 27 cfm. This is the kind of airflow associated with bathroom fans with 8 cm ducts, or with small or quiet fans for computer cases — definitely feasible.

You don't want to blow *too much* new air in because you have to heat it all up — air's heat capacity is 1.01 kJ/kg/K, so each additional ℓ/s is 1.23 g/s and 1.24 W/K, and with $\Delta T \approx 20$ K that works out to about 24 W of extra heater power, plus whatever the fan uses. This isn't very much extra energy use but it's energy use that doesn't evaporate much extra water.

(Or is it?)

Here in Buenos Aires, we have abundant municipal water which we could use to chill a condenser, and the Nutra already needs access to both a water intake and a drain. How much water would it waste? If we drop the dewpoint of the air to 25° (thus removing more than half the water) by running it through a countercurrent heat exchanger cooled by tap water starting at 15°, well, we would need 290 ℓ of water to dump those 12 MJ of heat. This might be okay in some kind of objective sense but people who run dietéticas are not going to tolerate it.

Lucía previously calculated that she would need a 2000-watt heater.

Water inlet valve

I have no idea what the water inlet valve is like. I'm guessing it's a solenoid driven by line power (through a diode) at under an amp.

Small pump motor

The pump needs to drain 6 liters (I think) of water within, I don't know, 20 minutes. 5 mℓ/s, 0.08 gallons per minute, 5 gallons per hour. This is a very small pump; a US$11 aquarium pump on Amazon is 227 GPH, 45 times as big, and it's 18 watts. So I think this is more like a 1-watt pump. The intrinsic work being done is very small, possibly even negative, but pumping 5 milliliters per second against a pressure of 1 meter column of water would be 50 milliwatts, plus whatever mechanical inefficiencies are encountered.

It might be feasible to run this directly off the ULN2003.

¼HP agitator motor

This is 190 watts output, perhaps 220 watts input — 1 amp at 240 volts.

Fan

A typical PC fan is like 100 to 300 milliamps at 12 volts. This could possibly be run directly off the ULN2003, which can handle 500 mA on a given pin at up to 50 V, though only one pin at a time.

Relays

So we probably need a couple of 1-amp relays, a 3-amp relay, and maybe another sub-1-amp relay for the pump. If we simplify this to four 3-amp relays, well, the usual suspect seems to be something like the Omron G5LE-1A4 DC12, which costs US$1.30 in quantity 1 and switches 10 amps at up to 250 VAC when you apply 12 VDC and 33.3mA to the coil. It's SPST, normally open. Or the TE Connectivity/Brumfield OJE-SH-112HM,000, apparently identical in every way except for costing US$1.32 and drawing 37.5 mA.

However, the Omron datasheet has a chart that shows durability of 200 000 operations at 3 amps and a 250 VAC resistive load or 120 VAC inductive load. With a 120 VAC resistive load, it's rated for twice the lifetime. The TE relay is only rated for 10 000 ops, though there's an "LM" variant rated to 100k.

All four relays at once would only be 133 milliamps, well within the ULN2003's limits.

There do exist mercury-wetted reed relays with lifetimes of a million operations, but these probably aren't necessary for a lifetime of one year.

Some kind of triac-optoisolator thing might be a reasonable alternative, but a typical triac drops 1.5 volts, so 3 amps would dissipate 5 watts, requiring a hefty heatsink and posing potential reliability problems of its own.

ULN2003 hookup

The ULN2003 only has a guaranteed β of 1000, so getting 33 milliamps on its output requires 33 microamps on its input. Except that actually the curves in the datasheet show that you need over 100 microamps to get anything to happen. This means that, at 3.3 V, the base resistor shouldn't be over 33 kilohms, which means you can't quite use the pullup resistors integrated in the STM32 I/O pins, since they could be as high as 50 kΩ according to the datasheet. You could maybe parallel two I/O pins, but it would probably be better to use external resistors, thus avoiding the chance of a software error burning up the STM32 or the ULN2003.

The ULN2003's maximum base current is 25 mA; 1 mA per pin should be plenty, so anything between 3.3kΩ and 33kΩ is adequate. 10kΩ, everybody's favorite resistor, is probably optimal, giving 330 µA, and thus 330 mA or better on the output pin.

Speaker

We can hook up a speaker to an extra output of the ULN2003 and control it through the built-in PWM of the STM32. If it's a ¼W 8Ω speaker, we need to keep current through it under about 100mA; a dc-blocking capacitor is probably worthwhile here, so that if the ULN2003 is on we don't have a constant current through the speaker with no sound, but also it would be a good idea to limit the current of ac signals and also perhaps filter out inaudible high frequencies. More

precisely, with 8Ω, to keep $I^2R < ¼W$ we need $|I| < 177$ mA, so a series resistance of 68Ω or greater is called for with 12V. Except with dc-blocking that's 12V P-t-P, so 6V RMS with a perfect square wave, so 33Ω would be adequate. The resistor is going to have most of that 6V RMS across it, like about 5V, so it needs to be pretty hefty, nearly a watt in the worst case. Probably prudent to limit the output power further, using 100Ω or so.

At 100Ω series resistance, an instantaneous 12V spike would push 120 mA, and 6V RMS would dissipate 360 mW in the resistor, which requires a largish resistor but nothing ridiculous; it would experience spikes to 1.3 watts. The 60 mA RMS would output 29 mW from the 8Ω speaker. You can see why people prefer to drive these damn things through transformers. Still, 29 mW is probably still audible; it's less than 10 dB below what I guessed was the maximum output for the speaker.

Now, what about filtering out ultrasound? If we shunt the speaker with a capacitor, we'd like its RC time constant to be in the tens of microseconds somewhere, so that 64 kHz (2.5 µs per radian) is strongly attenuated but 10 kHz (15.9 µs per radian) is minimally attenuated, if at all. But the relevant R here is not the 100Ω that's being used to limit the current, but rather the parallel combination of that 100Ω (plus the dc-blocking capacitor, which is hopefully negligible at the relevant frequencies) and the 8Ω of the speaker, which works out to be 7.4Ω. So we need something like 1.3 µF, not the 100 nF you would expect. At 1 µF, we get 7.4 µs, which puts the knee around 21 kHz, which means that we only have about 3× attenuation at 64 kHz (like 10 dB). If we use 2.2 µF, the knee is at 9.8 kHz, which gives us an extra 6 dB.

The dc-blocking capacitor needs to have a sufficiently long time constant with the series resistance to not mess up the bass too bad. Say we want its time constant to be below 40 Hz (4 ms per radian), then we need at least 40 µF. 100 µF is probably fine; at 12 volts this holds 7.2 mJ, which is what gets dissipated in that 1.3-W spike in the series resistance.

Perhaps a second RC section is worthwhile in this case: 10 ohms before the 100 ohms, then a 1µF to ground. So the whole output circuit from the ULN2003 pin would be `{ 100 µF 10 Ω { 1 µF gnd } 100 Ω (1 µF || speaker) 12 V }`, where concatenation is series combination, `{}` is a branch, `()` are grouping, and `||` is parallel combination. In a Falstad simulation, this configuration gives us about -4 dB at 40 Hz, -0 dB from 80 Hz to 5 kHz, -1 dB at 7.5 kHz, -2 dB at 10 kHz, -4 dB at 15 kHz, -5 dB at 20 kHz, -15 dB at 40 kHz, and -22 dB at 65 kHz. However, at high frequencies, this puts the whole 6 V RMS across this new 10Ω resistor through the capacitor shunt to ground, which means it will be dissipating 3.6 watts! And of course the PWM output from the ULN2003 is *always* close to 6 V RMS, except when it's close to dc.

So the second RC section is a bad idea. Probably much less hassle to use an inductor in series with the resistor, one whose impedance becomes large relative to 100 ohms at somewhere around 10 kHz, like a 4.7mH inductor. In a Falstad simulation, this gives about 3 dB attenuation at 3.5 kHz, 6 dB at 6.3 kHz, 10 dB at 10 kHz, 17 dB at 20 kHz, 27 dB at 40 kHz, 30 dB at 48 kHz, 38 dB at 80 kHz, and 42 dB at 99 kHz. Simulation also suggests about a 13.7 mV peak amplitude

part of a 48 kHz PWM modulating square wave would remain across the terminals of the speaker, compared to some 440 mV peak for the actual signal. (Maybe it works even better with L speaker || C instead of L (speaker || C), with a +6dB peak around 6kHz.)

Wait, actually the whole thing is bad in that form. I need a 220Ω pullup resistor on the ULN2003 to get current. And then maybe I don't really need an inductor. Here's the Falstad design:

```
$ 1 3.0000000000000004E-7 26.59566520631553 50 12.0 50
c 960 432 1088 432 0 9.999999999999999E-5 3.5513458288348803
r 1168 432 1264 432 0 8.0
R 816 304 816 224 0 0 40.0 12.0 0.0 0.0 0.5
w 1264 480 1264 432 0
r 816 304 816 416 0 220.0
a 592 432 672 432 0 12.0 0.0 1000000.0
174 272 384 368 480 0 1000.0 0.005 Resistance
g 272 480 272 496 0
R 272 384 272 352 0 0 40.0 5.0 0.0 0.0 0.5
R 592 416 560 416 0 4 65000.0 5.0 0.0 0.0 0.5
O 672 432 672 368 0
S 592 448 368 448 0 0 false 0
170 368 464 320 464 3 20.0 100000.0 5.0 0.3
l 1088 432 1168 432 0 0.001 0.02828594427035853
t 736 432 768 432 0 1 -5.526250662994705 -0.11573671215990365 100.0
t 768 448 816 448 0 1 -5.410513950834801 0.11562175983470305 100.0
w 736 432 736 496 0
r 736 496 816 496 0 10000.0
w 816 464 816 496 0
g 816 496 816 528 0
w 768 416 816 416 0
w 816 416 816 432 0
w 816 432 848 432 0
r 672 432 736 432 0 10000.0
g 1264 480 1264 528 0
r 848 432 960 432 0 100.0
c 960 432 960 496 0 1.0E-6 2.5834701256300843
g 960 496 960 528 0
o 1 64 0 35 0.5846006549323611 0.09353610478917779 0 -1
o 10 64 0 34 20.0 9.765625E-5 0 -1
o 15 4 6 35 10.0 9.765625E-5 1 -1
o 13 4 0 35 2.5 0.1 2 -1
```

Oh, I think I have a reasonable approach that eliminates most of the inefficiencies. {100Ω (L || 100µF speaker) 12V}, where the L provides a dc path around the speaker and capacitor, and the capacitor basically just protects the speaker from dc. The resistor in series with the whole shebang limits the current, but maybe it could be bypassed with a capacitor so that it can be bigger than would be desirable for ac. And you can still shunt the speaker alone with a capacitor to reduce ultrasound.

A potential problem with this is that it could introduce a kind of distortion when the conduction switches from being through the Darlington to being through the freewheel diode, just because of the forward voltage drop of the freewheel diode.

How much inductance does the shunt inductor need? Ideally it

shouldn't steal much current from the speaker at audio frequencies, which means its impedance at audio frequencies should be large compared to the speaker's 8Ω, although maybe the capacitor can help compensate for this. But let's say we want its impedance at 40 Hz to be 16Ω. This requires a fairly hefty 68-mH inductor (like, 8 mm × 12 mm.)

Or you could just drive the speaker through a little 4:1 100Ω-ESR audio-frequency transformer, which will probably filter out the ultrasound as a side effect.

What *is* the PWM frequency? The STM32F030x4 etc. reference manual RM0360 says, "If the APB prescaler is 1, the timer clock frequencies are set to the same frequency as that of the APB domain," which is normally 48MHz except for power saving. I think that means that if you want 8-bit PWM, you can get 187.5 kHz PWM. At such a speed, 1mH of inductance should be plenty — and how much does the speaker itself have? 1mH series and 1µF in parallel reduces the 187.5 kHz square wave to 5.6 mV peak, while the AF signal is like 440 mV peak.

Display

A Nokia 5110 84×48 display ought to be adequate and might fit in well with the overall feeling of the device, although OLEDs are nicer on several axes nowadays.

Sensors

Monarca sells a DHT-22 temperature and humidity sensor for $220. Nubbeo has the cheaper DHT-11 for $60, but its precision is ±2°, which would be terrible for the Nutra, restricting it to a temperature of 40° or below. By contrast, the DHT-22's precision is ±0.1°.

Apart from cost and precision, the sensors are otherwise quite similar.

Topics

Planar lookup tables

Kragen Javier Sitaker, 2014-04-24 (2 minutes)

I've written previously about heightfields for mechanical computation: http://lists.canonical.org/pipermail/kragen-tol/2010-June/000919.html. One difficulty mentioned therein is that fabrication technologies capable of producing individual one-off parts like the three-dimensional heightfields called for are rather expensive; drilling a single hole in a hard material can cost tens of cents, and thousands, if not tens of thousands, of such precise holes will be needed for a complete computing device.

Planar fabrication techniques such as acid etching, laser cutting, sawing with a jigsaw or fretsaw or coping saw or piercing saw, waterjet cutting, oxy-acetylene or plasma cutting, or cutting with a hot or abrasive wire are dramatically cheaper, but they can't cut only partway through the material, either with precision or at all. It would be very convenient to be able to achieve the two-dimensional LUT effect using only such planar fabrication techniques.

This is possible by using a tapered probe that measures the width of the hole, rather than its depth; this allows a flat plate containing holes of various widths to be used instead of the cube containing round holes of various depths I proposed before. The holes can be slot-shaped rather than round, since the taper will be cut from a flat plate. The friction and tensile forces resulting from the taper will be the limiting factor on the number of bits in the machine; this can be improved somewhat by making holes shaped like plus signs or rectangles and using two separate tapers in a plus-sign-cross-section-shaped probe, and perhaps by using stepped tapers.

Alternatively, of course, you could construct the three-dimensional heightfield by squeezing together a bunch of parallel plates, like the body of the common stamped-and-riveted laminated Master padlock, perhaps with glue between the plates to improve precision. But that requires some 33 plates for a single 16×16 heightfield rather than one.

Topics

Spring energy density

Kragen Javier Sitaker, 2016-05-28 (updated 2016-06-06) (13 minutes)

Suppose you want to make springs out of tool steel, which may not be the best possible choice but is close to feasible at least. (Normal people use music wire, apparently.)

http://www.matweb.com/search/datasheet.aspx?matguid=5abd35ce06bd64254b6db980b83683f27&ckck=1 says that AISI type A2 tool steel (which can be hardened by air-cooling after heating) has a modulus of elasticity of 203 GPa, a shear modulus of 78 GPa estimated from the elastic modulus (presumably with $E = 2G(1 + \nu)$, where E is Young's modulus, G is the shear modulus, and ν is Poisson's ratio), and a density of 7.86 g/cc. Unfortunately this doesn't supply the strength data that we need to figure out when it will break.

http://www.matweb.com/search/datasheet.aspx?matguid=8188603e05f954e53b479aaa4b07cdffb is a page about particular brand of A2 tool steel, giving its yield stress as 1580 MPa, its ultimate tensile strength as 2050 MPa, and its modulus of elasticity as 203.4 GPa, with a 1% elongation at break. (This makes sense because 2050 MPa is about 1% of 203 GPa.)

https://en.wikipedia.org/wiki/Shear_strength says that in steels, the shear yield stress is about 0.58 of the tensile yield strength (this is called the von Mises yield criterion, and apparently applies to ductile materials in general?)

This suggests that the shear yield stress of A2 should be about (* .58 1580) = 916 MPa, at which point it's distorted at a slope of (/ .916 78) = 0.0117, or 1.17%.

Let's consider the case of twisting a cylindrical torsion bar made of this material, say 1 m long, 200 mm diameter, and 1 mm thick, thin enough that we can mostly ignore the difference in radii between inner and outer walls, but thick enough that it won't collapse as we twist it. Twisting it by 1.17% means a twist of 11.7 mm over that meter length, which is 0.117 radians of twist. The cross-sectional area there is about $1\text{ mm} \cdot 200\text{ mm} \cdot \pi \approx 628\text{ mm}^2$. At the yield stress of 916 MPa, this cross-sectional area generates (* 916 628) = 575248 N of force, which is almost sixty tons. (We could convert this to a torque by multiplying by the 100 mm radius, but we don't need to.) Building linearly to that force over 11.7 mm of travel distance gives us (* 575248 .5 .0117) = 3365 J of energy stored in the bar.

What's the energy density of that? The metal occupies 0.628 ℓ of volume, so that's (/ 3365 .628) = 5358 J/ℓ. At 7.86 g/cc or kg/ℓ, that's (/ 5258 7.86) = 669 J/kg. This is close to, but larger than, the 300 J/kg cited in https://en.wikipedia.org/wiki/Energy_density_Extended_Reference_Table.

If we were somehow able to stress the metal in tension instead of torsion, we'd get to 1580 MPa at an elongation strain of (/ 1.58 203) = 0.0078 or 0.78%, or 7.8 mm, at a force of (* 1580 628) = 992240 N, and an energy of (* 992240 .5 .0078) = 3870 J, which is better, but

only by 15%. (/ 3870 .628) = 6162 J/ℓ; (/ 6162 7.86) = 780 J/kg.

Nested torsion tubes in series offer the opportunity to exploit this entire 669 J/kg energy capacity; normal coil springs only manage about two thirds of it, and springs such as garage door torsion springs that are stressed in bending rather than tension or torsion only get to half of the tension number. You need some space in between the tubes, and some mass in the coupling between the tubes at the ends, but those can be very small numbers. So 5kJ/ℓ should be a totally reachable energy density in practice.

Unfortunately, these specific energies are substantially lower than I want for the compact application I have in mind, in which I would like to hold several hundred to several thousand joules in a spring weighing under 200g, then release it in submillisecond timescales. 200g · 669 J/kg = 134 J, barely acceptable.

I should investigate whether spring steels, beryllium copper, or nitinol can provide larger energy capacities. Apparently ASTM A228 music wire is a common spring material, as are SAE 1074 and 1075 steels, while AISI 1095 steel (ASTM A684) is used for more demanding applications.

The speed of sound in a material is $\sqrt{(K/\rho)}$, where K is the relevant modulus of elasticity and ρ is the density of the material. In this case, for transmission of shear, sqrt(78 GPa/7.86 (g/cc)) comes out to 3150 m/s. This means that it takes 317 μs for a movement to travel a meter through the spring; the suggested meter-long torsion bar won't be able to respond faster than that. You can, in effect, fold up the spring and nest it inside itself into a series of nested torsion tubes, which in theory won't affect either the response time of the spring or its rate. You can decrease the spring rate (i.e. increase the compliance) by using thinner-walled tubes, up to a point where the spring buckles and collapses; but, if you do that while keeping the energy capacity and mass constant, you have to increase the response time proportionally.

If your torsion spring is a single tube of constant diameter, it is useful to give it a constant thickness as well; otherwise, the thinner part will fail before the thicker part is fully charged. This is because the torque is constant along the entire length of the tube, but that torque translates into different stresses at different thicknesses. Once you start nesting the tube inside of itself in series, you still have the constant torque, but now you have different radii in different parts of the spring, which translates to different tangential forces inversely proportional to the radii; this means that the wall thicknesses also need to be inversely proportional to the radii in order to keep the stress constant through the entire spring.

This has the problem that once the radius is small enough, the inner and outer radii start to differ significantly, which means that the stress on the inside radius of the tube is significantly lower than the yield stress. For example, the 1mm-thick 20mm-wide tube in the example above, in which the stress is below optimal by only 10% on the inner wall, could be put in series with a 10mm-wide 5mm-thick tube — which is no tube at all, but merely a rod! It has no torsional stress at all at its center.

The upshot of all of this is that you can get arbitrarily fast reaction times only at the cost of arbitrarily high forces or arbitrarily low energy capacities per spring.

What about Dyneema? If it has an ultimate tensile strength (and also yield stress!) of 2.5 GPa and a Young's modulus of 100 GPa, and we somehow stress it in tension instead of torsion, then that same 628 mm² cross-section rope of it would hold up to 1.57 meganewtons, about 160 tons; if it were 1 m long and stretched by 2.5%, or 25 mm, then we'd have 19.6 kJ stored, 31.25 kJ/ℓ. And it would weigh only 609 g, so it's 32.2 kJ/kg.

In energy density per kg, that's *48 times better than* the steel spring. Per liter, it's only about six times better. And that's without getting into weird effects like rubber's hyperelasticity or nitinol's pseudoelasticity which I worry might convert their theoretically higher energy capacities into delusions at these time scales. According to ARL's tests http://www.dtic.mil/get-tr-doc/pdf?AD=ADA606636 Dyneema is capable at least of absorbing energy at these moduli at a kilostrain per second.

http://www.sciencedirect.com/science/article/pii/S1877705811005640
9

This suggests that maybe nylon would be an even better place to look for spring energy density! It's not as strong as Dyneema, but it has a dramatically lower modulus, and I think it also avoids the weird pseudoelasticity thing where the "elastic" energy gets lost as heat. http://www.engineeringtoolbox.com/engineering-materials-properti
es-d_1225.html claims that nylon 6/6's density is 1.15 g/cc, its tensile modulus is 2 to 3.6 GPa, its tensile strength is 0.082 GPa. Dividing, that gives it 24% (or more) elongation at break; 24% .082 GPa/2 = 9.8 kJ/ℓ. That's not as good as Dyneema, but it's still volumetrically better than most steels, and a *lot* better per mass, at 8.6 kJ/kg. Nylon springs would easily meet the 200g limit I'm trying for: 200 g 8.6 kJ/kg = 1.7 kJ. In fact, you could probably get by with 100g or 50g. And nylon is a *lot* cheaper than Dyneema still.

The √(K/ρ) speed of tensile sound in nylon, by these numbers, should be 1318 m/s, only four times its speed in air, and low enough that you can only get 1300 mm of nylon spring to respond in a millisecond. However, this should be plenty of time. I really only need a millisecond or so; 100μs response time is more than adequate, and I have in mind for the entire spring to be under 100mm long.

Disappointingly, it seems that nylon is indeed hyperelastic and behaves much more stiffly at high strain rates; http://scholarbank.nus.edu.sg/bitstream/handle/10635/37891/PhD%
20Thesis,%20Habib%20Pouriayevali%20(%20Mechanical%20Dep)%20
0HT081385J.pdf?sequence=1 and http://www.sciencedirect.com/science/article/pii/S1877705811005640
9 show the results of H. Pouriayevali with respect to the issue, showing that his nylon sample compressed 30% under a 50 MPa strain under quasi-static conditions, but at strain rate of -3203 per second, it had only compressed 2% at that same strain (which I suppose means he reached it in 6 μs). At a more moderate strain rate of -980 per second, it had compressed some 8% at that strain, in, I suppose, about 80 μs. The dissertation also, alarmingly, says that nylon "is notably rate-dependent and exhibits a temperature increase under high rate deformation", but it turns out that he's talking about like five kelvins for fairly large strains like 21%.

Pouriayevali's 6-6 nylon samples failed in quasi-static tension at strains which appear to have been around 1.0, 1.2, and 1.4, with stresses around 80 or 100 MPa, which seems improbably rubbery, but fairly plausible strength. He reports a quasi-static elastic modulus of 958 MPa at small strains, but as the graphs show clearly, this drops precipitously at higher stresses.

This stiffness varying with strain rates presumably means you have high hysteresis losses in nylon, more usually known as "vibration dampening ability".

However, I'd probably be fine with >50% losses, as long as I can get the rest of the energy out in well under a millisecond. 12% strain in a millisecond is a strain rate of 120 strains per second, which may be low enough that nylon will have low losses.

Pouriayevali did do lower-rate experiments under tension. His lowest-rate dynamic tension experiment involved an impact of 150 strains per second, reaching some 30 MPa and 5% strain; the quasi-static condition had only 20 MPa at that same 5% strain. That means that if you stretched your nylon by 5% at 150 strains per second, you'd've put in 30 MPa * 5% / 2 = 750 J/ℓ, and then if you unloaded it slowly, you'd only get out 20 MPa * 5% / 2 = 500 J/ℓ. In this case I propose to do essentially the reverse: load it slowly, then unload it quickly. It looks like this will probably be in the neighborhood of that 50% efficiency.

(Actually the stress-strain curves aren't very linear, and they're actually maybe a bit closer together than that makes it sound.)

In chapter 5 of his dissertation, Pouriayevali fits numerical models to the properties of the nylon which seem to suggest something like 10% or 20% energy losses at 150 strains per second when extended to larger strains, with actually much lower losses at higher strains, up to 0.6. Also, the stresses at higher strains are more nearly constant, which should ease design substantially.

Topics

Simple persistent in-memory dictionaries with log² lookups and logarithmic insertion

Kragen Javier Sitaker, 2014-02-24 (6 minutes)

An in-memory dictionary with O(lg² N) lookups, O(lg N) amortized insertions, O(lg lg N) key-ordered traversal, dramatically better memory usage and cache behavior than binary search trees, persistence in Okasaki's sense, and very simple code.

Search trees store sets or bags in logarithmic time. The simplest search-tree algorithms are very simple indeed, although they use a lot of memory and a lot of random memory seeks; but they degrade to linear search in the worst case. Advanced search trees like red-black trees and B*-trees fix these problems, but at the cost of great algorithmic complexity.

A perfectly-balanced binary search tree can be represented without pointers, as an array; the usual binary heap algorithms work this way, taking the nodes at 2N and 2N+1 as the children of the node at N (or 2N+1 and 2N+2, if you're zero-based.) Alternatively and equivalently, you can use binary search: a subtree of 2N+1 nodes is comprised of the concatenation of a left subtree of N nodes, a single root node, and a right subtree of N nodes. These two representations are equivalent and interchangeable, and they share the problem that inserting into them is O(N) rather than O(lg N). (The binary-heap approach seems like it should have better cache behavior for lookups, and worse for sequential traversal.)

Lucene handles a similar problem by maintaining its index as a set of O(lg N) "index segments", each of which is sorted. Insertion takes the form of generating a new index segment, while a search requires doing an O(lg N) search in each of the O(lg N) segments. Creating a new index segment may result in merging some existing index segments into larger index segments.

(Lucene's data structure, because it requires only sequential writes to new segments and never random writes, is also well-suited for concurrency and variable-sized data items.)

The simplest version of this approach would make the index segments powers of two, with a maximum of one index segment of each possible size. So if you had index segments of sizes 1, 2, 4, 16, and 64, then after inserting a new datum, you would have index segments of sizes 8, 16, and 64, having merged the three smallest segments together. (It might simplify the search of each individual index segment to use powers of two minus one: 1, 3, 7, 15, 31, and so on. But then you'd need to allow up to two segments of any given size.)

It's straightforward to see that you'll have between 1 and lg N segments, and that the average case will be ceil(lg N)/2. An insertion will cause zero merges half the time, one merge a quarter of the time, two merges an eighth of the time, three merges a sixteenth of the time, and so on, for an amortized constant of two merges per insertion. Those merges are mostly small and fast: four-item merges

are half as common as two-item merges, eight-item merges half as common as four-item merges, sixteen-item merges half as common as eight-item merges, and so on. However, this series still fails to converge, which makes sense since in its infinite form it represents the work of inserting into an already-infinite set. If you terminate it after N terms, it represents the work of inserting into a set of 2**N items, and it adds up to N.

As with trees, this approach supports "persistent" data structures, where you can hang on to a previous version of the structure simply by holding onto a pointer to it, sharing state with current versions — as long as you store it as a linked list, going from the smallest to the largest index segments. You lose the amortized time guarantee, though; if you save off a version with 1023 items in it and then make 100 modified versions of it with one item added, you'll do a 1024-item merge for each of those 100 modified versions, for 102400 total work.

This data structure is asymptotically slower than a binary tree for lookup, since doing a lookup takes $O(\lg^2 N)$ probes — for example, in a 42-item dictionary, you must search in the 2-item array (1½ probes), the 8-item array (3⅛ probes), and the 32-item array (about 5 probes), for a total of 9⅝ probes, while a perfectly balanced binary tree would take slightly over 5. In a 2730-item dictionary, you must do that and also carry out 7 probes in the 128-item array, 9 probes in the 512-item array, and 11 probes in the 2048-item dictionary, for a total of a bit over 36 probes, while a perfectly-balanced tree would take under 12 probes. I've chosen the numbers 42 and 2730 because they're uncharacteristically average.

(Here I'm assuming that you must examine every candidate location, either because your lookup is unsuccessful or because you could theoretically have more than one value for a given key, but don't. In the case where keys are guaranteed unique, you may be able to stop earlier, but I think that doesn't affect the asymptotic time; and if you actually do have multiple values per key, the worst case is that you have to return every value.)

If we allow index segments with less than the maximum possible number of keys, it's possible to run an $O(N \lg \lg N)$ "optimization" pass on the data structure, sorting everything into a single array. Thereafter lookups will be $O(\lg N)$.

Topics

• Programming (p. 3658) (286 notes)
• Performance (p. 3621) (149 notes)
• Algorithms (p. 3310) (123 notes)

Object embedding

Kragen Javier Sitaker, 2014-02-24 (13 minutes)

In Java (as in Smalltalk, Lisp, Python, ML, and many other languages) when an object A "contains" an object B, the in-RAM representation of object A contains actually just a *pointer* to B, i.e. the address at which B's in-RAM representation can be found. This is in some ways convenient, but it has some drawbacks — not only for efficiency, but even for program correctness.

In C, instead, "A containing B" means that the in-RAM representation of B is actually contained inside the in-RAM representation of A. I don't know of a standard term for this, so I'm calling this "object embedding." (Object embedding is one aspect of one possible implementation of what is sometimes called "compound value types", for example in C# and in *Elements of Programming*.) C, C++, Pascal, and Golang do the same thing; they all inherit it from COBOL's "group items", from which Algol got "record types", which C renamed "struct".

The benefits of by-reference object fields are significant:

• A's **reference to B is polymorphic**, i.e. you can use an object of any subclass of the type that A is expecting. This means that objects-by-reference are absolutely essential to the Liskov Substitution Principle. C++ has a terrible bug resulting from its misguided attempt to make embedded objects polymorphic, known as the "slicing copy". C, Pascal, and Golang do not suffer from this because embeddable objects do not have subtypes, unless you think of the variants of a union type ("sum type") as being subtypes of it.
• B can be **aliased**: several different objects A can "contain" the same object B. This can be desirable if B is large or if it has mutable state that many objects need access to.
• A's reference to B is **nullable**, which is particularly important in languages like Smalltalk, C++, and Java, where you can access an incompletely initialized object (from inside its constructor, if nothing else).
• Pointers allow circular references.

However, compared to Java's approach, it occurs to me that the embedding approach has some real benefits. More than once, I've written code like this:

```
public class LogFileParser {
    private Map<String, String> messageIds;
    public void sawMessageId(String messageId, String queueId) {
        messageIds.put(queueId, messageId);
    }
}
```

This has the tendency to throw a `NullPointerException`, because of course declaring that your LogFileParser objects *could* have a `Map` from `String`s to `String`s doesn't give them one. You need to say:

```
    private Map<String, String> messageIds =
        new HashMap<String, String>();
```

`final` on instance variables is not nearly as useful as you might think when you're obligated to use mutable objects for a lot of those final instance variables. It does, however, avoid the above problem.

By contrast, embedded objects aren't nullable. If you say:

```
struct foo {
    struct foo *next;
    struct bar my_bar;
    int weight;
};
```

then you can never have a `struct foo` whose `my_bar` is `NULL`, because it's not a pointer. So the above NullPointerException cannot arise, not even as a compile error. In C++ you can even have a default constructor for `bar` that ensures that its memory doesn't contain random garbage; in Golang it's initially zero, and you get the same guarantee in C if you use `calloc` or initialize a local `struct foo f = {0};`.

The consequence is that in C++, if you say

```
std::map<string, string> message_ids;
```

— either as a local variable or an instance variable — you have a full-fledged empty `map` sitting there, without any need to repeat again the fact that you wanted one. This kind of thing, in combination with the STL container library, allows you to write entire useful C++ programs that don't mention heap memory allocation at all, because it's all hidden away inside the STL.

(ML and its descendants take another approach to this problem, one which inspired the `final` mechanism in Java: although relationships by objects are by means of pointers, there is no `null` value, and so you can't create an object without supplying values for all of its fields.)

Aside from this **implicit nullability imposed on every object field**, there are a few other problems with by-reference objects.

• **They allow for aliasing**, which is to say, many-to-one relationships: a subobject included merely by reference can also be included in many other objects. Sometimes this is what you want — it's a major reason for using pointers in languages that do support object embedding — but aliasing, like nullability, is bug-prone, and therefore **probably a bad default** for mutable objects.

(Of the languages I've mentioned, only Pascal actually forbids you from making an alias to an object embedded inside another object, e.g. in C, writing `&my_foo.my_bar`. But it's relatively uncommon in practice to store such a reference into another object.)

• They **greatly increase the number of separate memory allocations**. This means that the graph that your garbage collector has to traverse is very much larger, and it correspondingly must do a great deal more work. This is one reason why ML garbage collectors tend to be very efficient — they have to be to make ML practical at all — and also the reason that Golang is reasonably efficient even though its garbage collector is still kind of crappy.

• In themselves, the **pointers take up memory**. If the only pointers in your memory are for object fields that you decided *ought* to be

nullable or aliasable, then most of your memory will probably be spent on the data that your program unavoidably has to manage. By contrast, in a language like Java or Python, it's easy to spend most of your memory on pointers, especially on 64-bit CPUs.

(Having more pointers also makes the garbage collector's job a lot harder, since it has to traverse the pointers in your live objects.)

Quantitatively, it seems that half to two-thirds of a typical Java program's memory is taken up by pointers, because moving to 64-bit increases the needed heap size by 50%.

• CPUs **can't generally prefetch pointer targets**, and they won't be in the same cache line; so accessing subobjects is a lot more likely to generate cache misses. By contrast, accessing embedded subobjects takes not only *less* time, but often *zero* instructions and zero time! See the section below, "zero instructions".
• Allocating every object on the heap means that **any part of your program that creates objects can fail unpredictably** if you run out of memory; in very memory-restricted environments like an Arduino, the heap can start to overwrite the stack and vice versa. Error handling mechanisms can control how the failure is handled, but they can't turn failure into success. (Worse, if your error handling creates objects, it can fail too.) In practice, under many circumstances, it's probably better to recover from memory allocation failures by immediately aborting the program.

This sounds like a bigger problem than it is in practice, because most software can be called upon to handle arbitrarily-sized problems, which means that you can always find some way to make it run out of memory and fail. Dynamic memory allocation is built into the problem, not just your solution to it. So increasing the number of places you can get an out-of-memory error isn't a big deal.

But there is some software that needs to work *all the time* instead of sometimes failing unpredictably, and that software should be written without dynamic allocation, which means it should not be written in a language where object fields are by reference. It's not acceptable if your antilock braking system, your jet engine controller, or your air-traffic control system gets an out-of-memory exception and reboots while you're using it.

On the other hand, if it's okay for your software to fail every once in a while, and it's just a matter of reducing the frequency to once a day, once a year, or once a millennium, dynamic allocation is not a big deal. In fact, dynamic allocation can help a lot with that. One of the major reasons the Fuzz paper found that the GNU tools were so much more reliable than the Unix tools they replaced is that they use dynamic allocation all over the place, so they have few arbitrary limits. Every dynamic allocation is a potential failure in your program; but every arbitrary limit is a potential bug.

However, these disadvantages are irrelevant if you want to practice OOP, because if you don't have ubiquitous LSP polymorphism, you aren't practicing OOP. So, in a way, these can be seen as five disadvantages of OOP.

(In theory I don't see why you couldn't have a functional programming language where object embedding was the norm, but I haven't seen one. Perhaps this is because functional programming languages have mostly adopted other approaches to solving the

problems here that are not questions of mere efficiency: for nullability, sum types like ML's `Option` or Haskell's `Maybe` and constructor expressions; for aliasing, a default of immutability; and, to some extent, for garbage collection, sophisticated generational garbage collectors.)

Parametric polymorphism

One big disadvantage of object embedding is that it requires you to implement parametric polymorphism (aka generics, generic types, or parametrized types) by means of generative programming, which can create a combinatorial explosion in code size. ML-family languages get parametric polymorphism basically for free, because every value is the size of a machine word (ironic that the 1980s explosion of functional programming traces back to Backus's misplaced attack on the "conceptual von Neumann bottleneck"!) so you only need one copy of the machine code for `List.map`, not one copy for each size of list item.

Zero instructions!

My assertion above that accessing embedded objects is not only cheap but actually *free* may seem dubious. What I mean is that, often, what you're doing with an embedded object is accessing fields within it, and if those fields themselves are embedded objects, you can easily end up with an entire chain of three or four or more field accesses compiling down to a single indexed fetch instruction, rather than a series of three or four of them.

As an example, in My Very First Raytracer, I have a function which begins:

```
static bool
find_intersections(ray rr, sphere ss, sc *intersections)
{
  vec center_rel = sub(rr.start, ss.cp);
…
}
```

which invokes these functions:

```
static vec
add(vec aa, vec bb) { vec rv = { aa.x+bb.x, aa.y+bb.y, aa.z+bb.z }; return rv; }
static vec
sub(vec aa, vec bb) { return add(aa, scale(bb, -1)); }
static vec
scale(vec vv, sc c) { vec rv = { vv.x*c, vv.y*c, vv.z*c }; return rv; }
```

So, given all this, the `center_rel` code works out to

```
vec center_rel = { rr.start.x + ss.cp.x * -1,
                   rr.start.y + ss.cp.y * -1,
                   rr.start.z + ss.cp.z * -1 };
```

Even mild inlining optimization can turn that into six indexed memory fetches and three subtractions, followed by three stores, but as far as I can tell, GCC actually somehow manages to allocate an SSE

register to each of the six values, and so those 12 field accesses are incorporated into three instructions. (I can't follow the generated assembly at all, even with the aid of `-g -Wa,-adhlns=raytracer.lst`. GCC has inlined basically the entire raytracer program into a single giant recursive function called `trace` of some 500-plus instructions.)

Again, that's 12 field accesses, plus some math, in 3 instructions, thanks to object embedding.

Further alternatives

What was the name of that guy who advocated using parallel arrays in C, Fortran-style? He was also pretty unhappy about the decline in hi-fi audio systems since the 1970s.

Topics

General purpose layout syntax

Kragen Javier Sitaker, 2017-11-10 (updated 2019-09-01) (34 minutes)

(Quicklayout (p. 2189) concerns the speed of text reflow.)

Graphviz has this really clever way to handle layouts inside nodes of shape "record". Fields are separated by "|", and by default are laid out left to right; within "{}" the default layout is transposed, top to bottom instead. So "A|{B|C}|D", for example, is a three-column layout with two rows, B and C, in the middle column. Hboxes nest inside vboxes; vboxes nest inside hboxes.

There are some other details, and now this is sort of deprecated in favor of a subset of HTML they're implementing. But I think the "|"-separation layout is brilliant, and can be extended in interesting ways.

First modification: use friendlier characters "[,]'\n", and vertical by default

Instead of "{|}", which are optimized for not occurring by accident, and "\" to escape them, let's use "[,]" for nesting and separators, and "'" for quoting, as in Lisp. So "65,535" is written as "65',535". All of these characters are easily reachable on a standard QWERTY keyboard without shift or extreme reaches.

Let's make the top level of nesting be vertical by default, rather than Graphviz's horizontal.

Also, let's add newline as a shorthand for "],[". This allows us to write this table

```
| a | b | c |
| 1 | 2 | 3 |
```

as

```
[a,b,c
1,2,3]
```

rather than "[a,b,c],[1,2,3]".

XXX maybe this should be reversed in some way? Gotta think about that.

I was thinking that maybe the quote or escape character should be "/" and should follow the character being quoted, but I think that's probably bad in that it involves arbitrary lookahead to see if a sequence of escape characters is going to be even or odd and thus determine whether it quotes the character before it or not.

Second modification: table layout

Let's establish the rule that sequences of sibling nested boxes, like the hboxes "a,b,c" and "1,2,3" in the previous example, have aligned fields, so they lay out as a table. If there is an intervening sibling box with no nesting, that terminates the table.

To soften this, let's add boxes whose content is merely "continuation of box to the left" and "continuation of box above". For these we can use ">" and "^" respectively rather than ",", so that

this table

```
 procs -----------------------memory---------------------- ---swap-- -----io---- -o
osystem-- --------cpu--------

  r  b         swpd         free         buff        cache   si   so    bi    bo  o
o in   cs  us  sy  id  wa  st

  1  0            0      1738140       120936      1405284    0    0    22    14  o
o 55  961   2   2  97   0   0

  0  0            0      1737876       120936      1405292    0    0     0     0 1o
o117 1180   1   2  97   0   0
```

can instead be written as follows:

```
[procs>,memory>>>,swap>,io>,system>,cpu>>>>
r,b,swpd,free,buff,cache,si,so,bi,bo,in,cs,us,sy,id,wa,st
1,0,0,1738140,120936,1405284,0,0,22,14,55,961,2,2,97,0,0
0,0,0,1737876,120936,1405292,0,0,0,0,1117,1180,1,2,97,0,0]
```

This provides substantial layout power and is actually easier to use than fixed-width ASCII table layout, as far as it goes.

If you actually want hboxes vertically adjacent to one another not to share columns, you can add extra levels of nesting: "[[[a long name,b,c]]],[[[1,2,3 million tons]]]".

XXX the following part is half-baked.

As a possible modification of this to support outline-type structures, you could maybe allow a nested box with fewer cells to partially terminate a table structure. Take this example:

```
[+,usr>
,+,local>
,,+,bin>
,,,,ncftp
,+,bin>
,,,bash]
```

Here the first line has three cells (the third of which is merged with the second), while the fourth line has five (the first four of which are empty). The next line has only four cells again, so the boundary between the fourth and fifth cell on the previous line is forgotten.

Third modification: hashtags

To identify particular cells or classes of cells, we can insert hashtags into them which are not displayed by default. Perhaps "[a #th,b #th,c #th #ch],[1,2,3]" tags all three of a, b, and c with "th", but only c is tagged with "ch", identifying it uniquely. You can put the hashtags anywhere in the cell; at the beginning or the end, separating whitespace will be removed.

There are at least five kinds of things you might want to do with hashtags: style cells, set or append to cell contents, link to cell contents, read cell contents, and listen for user interaction in cells. All of these might query hashtags (or contents!) in the cell itself, in its

ancestors, in its siblings, or in other cells aligned with it. In a syntax for these, we need three or five directional indicators, plus some way to indicate whether we mean an immediate neighbor or an eventual one.

For example, you might use “>” to indicate an immediately preceding sibling node, “>*” to indicate any sibling node, “/” (or maybe “.”) to indicate an immediate parent node, “/*” to indicate any ancestor node, “^” to indicate a corresponding node in the second dimension of the table (such as a column), and “^*” to indicate a previous node. So perhaps “th” indicates a cell tagged “th”, “ch” indicates the cell tagged ch, “ch^” indicates the cell below the cell tagged ch, “ch^^” indicates the cell two below it, “ch^*” indicates the entire column, and “ch^* sydney>*” indicates any cell in any column containing the “ch” hashtag and any row containing the “sydney” hashtag. (Or vice versa if the table is laid out in columns? Bleh. Also maybe one of these operators should be postfix and you should be able to stick them before or after the hashtag.)

As an alternative to identifying cells with hashtags, you can identify them with search strings with "?string?" or maybe regexps.

Of course there are other kinds of things than hashtags you might want to put into cells and not show up as pure text: sparklines, progress bars, character-level markup, URLs to link to, raster and vector images, and so on.

You could use cursor movement commands as an alternative to hashtags as a way to identify text to modify.

Cheap in-memory representation

It would be nice if we could represent a node unambiguously as a pointer into the string representation (or *some* string representation), which would require that each tree node has a unique first byte in that string representation. We might have to make some minimal changes. Consider again “[a,b,c;1,2,3]”, which is a vbox containing two hboxes which each contain three leaf cells. If we expand this to have a unique root node and expand out the “;”, it becomes “[[a,b,c],[1,2,3]]”, and now each node indeed has a unique first byte. Also, the commas and “]”s are not the first byte of any node, so expanding out the “;” was unnecessary; “[[a,b,c;1,2,3]]” is adequate. This gives us a 15-byte representation of the whole tree, although proceeding from one sibling node to the next may be somewhat slow. (A cache for large jumps could limit this problem.)

Compare this to a Lisp-style or C-style record-and-pointer representation:

```
digraph abc123 {
    rankdir=LR;
    node [shape=record];
    subgraph rows {node [label="{count|3}|<0>|<1>|<2>"]; row0 row1;}
    root [label="{count|2}|<0>|<1>"];
    root:0 -> row0; root:1 -> row1;
    row0:0 -> a; row0:1 -> b; row0:2 -> c;
    row1:0 -> 1; row1:1 -> 2; row1:2 -> 3;
}
```

Just the internal nodes of this tree take up 11 pointers of storage,

typically 4 bytes each nowadays, and that's not even counting the leaf nodes that contain the labels. If you make linked lists of siblings you can ditch the count fields, but then you pay even more in next-sibling pointers:

```
digraph abc123 {
    node [shape=record];
    subgraph i {node [label="<k>kids|<n>next"]; d e f g h i j k;}
    d:k -> e:k -> a;        e:n -> f:k -> b; f:n -> g:k -> c;
    d:n -> h:k -> i:k -> 1; i:n -> j:k -> 2; j:n -> k:k -> 3;
}
```

This has 16 pointers instead of only 11, because it needs 8 next-sibling pointers (two of which are null) instead of 3 count fields.

A friendlier syntax variant with more special cases

Suppose the top-level separator is "\n" and if there's a "\t" it gets inferred as a second-level separator, and we use "{}" for further nesting. Then we can ingest tab-separated values (with some strange quoting conventions) natively, without losing the power of nesting. The main disadvantage is that it's much more irregular, and there are lots of cases where a level of nesting can fail to be inferred because there's only one item.

Some failed ideas

ASCII has separate horizontal and vertical separators: HT and VT (^K). Suppose you adopt the rule that you infer a surrounding hbox when you find an HT, and a surrounding vbox when you find a VT. Then maybe you could infer all your hboxes and vboxes instead of having to mark their beginnings explicitly, and then you would be winning because you would only have to mark the endings, with some third control character, like LF, which would end the current hbox or vbox. For example, given this text:

```
a   b   c
d   e   f
```

which is to say, a<HT>b<HT>c<LF>d<HT>e<HT>f<LF>, you would infer an hbox upon finding the HT after "a", and then pack "a", "b", and "c" into it, and then end the inferred hbox upon finding the LF. Then upon finding the HT after "d", you would infer a new hbox (and perhaps a vbox to contain it and the previous hbox). The idea was that you would never infer an *end* to any current box; they would all be explicit.

This doesn't work for a couple of reasons:

- You can put a vbox as the second field of an hbox with a sequence like a<HT>b<VT>c, but never as the first, because you haven't inferred the hbox yet. So the <VT> will be treated as a separator for the surrounding vbox.
- It's too easy to accidentally end the outer vbox. For example, a<HT>b<LF>c<LF> would end it, because no <HT> has yet been seen to infer

an hbox to pack the `c` into. To be consistent, if more text followed, you'd want to infer an outer wrapping hbox surrounding that vbox, and so on.

• You can't have a box with only one child.

I also thought about using a stack machine model: first you output some boxes separated by separator bytes, and then you combine them by emitting some combining bytes that will join them into hboxes and vboxes. This has most of the same problems and additionally makes text hard to edit.

Some better ideas

It's probably best to stick to LF as the separator. Then, what do we use for group start and group end codes? Nonprintable characters would be nice. ASCII-1967 has STX and ETX, ^B and ^C, which sound promising, but STX is really EOH (end of header). And ^C has acquired the meanings of killing processes or copying text, either of which would be unfortunate to invoke accidentally. ^R and ^T (DC2 and DC4) would work; they used to turn your paper tape punch on and off. SO and SI (^N and ^O) would also work; on VT100s they select the graphical character set, which is a nicely parallel meaning, although they don't nest there.

ECMA-48 defines a variety of opening delimiters in whose closing delimiter is ST, String Terminator, 0x9c; specifically APC (application program command, 0x9f), DCS (device control string, 0x90), OSC (operating system command, 0x9d), PM (privacy message, 0x9e), and SOS (start of string, 0x98). Nesting is explicitly forbidden; according to the spec, most of the strings need to be printable ASCII, not even printable ISO-8859-1. There are also PU1 and PU2 codes, 0x91 and 0x92. These are all in the little-used C1 control codes area of ISO-8859-1 (originally ANSI X3.64), which Windows-1252 repurposed for characters that are more commonly used.

So our table might look like `<DC2>a<LF>b<LF>c<DC4><DC2>1<LF>2<LF>3<DC4>`. This is kind of suboptimal in a number of ways:

• as I said before, it's more bytes than the usual CSV;
• with the unprintable delimiters, if rendered by a traditional terminal, it's actually even more misleading than before, because "c1" appears as a single line. Maybe it would work better to use CR (^M) and VT (^K), which xterm, konsole, libvte display as if it were ^J.
• if you're editing it byte by byte, if you delete either the opening or closing delimiter in the middle, you will get a change of orientation of the text, which could be confusing. Or maybe useful if you actually wanted to transpose a table, I guess, but it also transposes all the subtables. Worse, though, all the text afterwards will change orientation.

A possible further alternative would be to determine box type by its internal delimiters (^I vs. ^J), but still have separate nesting delimiters (^K and ^M, say). Then you could go one step closer to regular terminal ASCII interpretation by inferring (only) hboxes within vboxes when necessary to handle ^I and ^J occurring within the same container. This requires four characters instead of three, but eliminates all the other problems previously mentioned.

You still have the problem of unmatched delimiters swallowing the

rest of the document, at least temporarily, when they're inserted in the middle. There must be some kind of possible solution to that but I don't know what it is. Maybe eliminating arbitrary nesting entirely and switching to transclusion of named chunks (maybe with Knuth-style 1f, 1b labels) for nesting? Sigh.

Use cases

The most obvious use case is a human being interactively preparing a two-dimensional nested layout of text for another human being to read, and I think this approach (supplemented with a few restructuring commands) is probably superior to anything I've seen for that purpose.

The second most obvious case is to produce human-readable output from a computer program. This syntax provides an easy way for program output to be tabular, nested, and semantically tagged for styling.

You might also want to use that semantic tagging for data extraction and to specify additional transformations to make.

The layout algorithm for this format is not quite as simple and fast as the layout algorithm for fixed-width ASCII text, but it's dramatically simpler and faster than the CSS box model, especially if you don't do word wrap.

You also might want to use this data model for parsing data from other sources — HTML tables, spreadsheets, discussion threads, XML files, JSON, relational databases, that kind of thing.

Implementing a prototype

What would it take to get a prototype of this thing running so I can see what it feels like to type into it? Probably the easiest substrate available is DHTML, and although part of my motivation is to have something that reliably lays out and redisplays faster than DHTML does, maybe a DHTML prototype would be adequate.

Given that desire, though, it seems like I'll still have to use JS to do all the computations. In a situation like {{A|B}|{C|D}}|{E|F}, we have to satisfy the following constraints:

- {E|F} is below the ABCD row.
- {E|F} is the same width as the ABCD row.
- E is the same width as {A|B}
- F is the same width as {C|D}.

So far so good — that's just a 2×2 table. But there's more:

- {A|B} is to the left of {C|D}.
- A is the same height as C.
- B is the same height as D.

Again, that's okay; it's just turned our 2×2 table into a 2×3 table. But there are cases where it's essential to not have certain constraints that would flow from treating the whole thing as just one big table. For example, if you replace A|B with {{A|B}}, the constraint with {C|D} goes away. Maybe you can still use rowspan and colspan with nested tables to get this effect, I'm not sure.

The interesting thing here is that each level of nesting has its own parallel cell constraints to deal with. If a table like {a|b|c}|{d|e|f}|{g|h|i} is in a cell that has sibling cells, the table's

own rows (or columns) may be subject to alignment constraints with the contents of those sibling cells — but only to that one level, as the individual cells within those rows or columns will not be.

I still feel that the whole thing can still be pretty much done with two passes: one pass to figure out which heights and widths have to be equal, and a second bottom-up pass to calculate them.

Yes, it's fairly simple. If we just do nesting, without tables, each minimum width is either a constant, for text boxes; the maximum of some other widths, for a vbox; or a sum of other widths, for an hbox. These minimum widths are strictly arranged in a tree. (A corresponding, parallel, and independent tree is present for heights.) So the minwidth of each hbox is a sum of maxima of minwidths of hboxes (or text boxes). If we switch to strict tables, instead, the minwidth of a column is a maximum of sums of minwidths of vboxes. I think this will be simple but I still don't have it entirely clear in my mind.

Ragged tables, where different rows or columns have different numbers of items, and so, for example, a column may only affect certain rows of a table, seem like they will be more complexity.

We can compile the layout into a tree of data dependencies: the minwidth of a column is the maximum minwidth of its cells; the minwidth of a table is the sum of the minwidths of its columns; and so on. Ragged tables make the topology more complicated because the minwidth of the table may come from a row which may actually have a contribution from the actual width of one of its cells that is imposed by the minwidth of a cell in a different row, while that row may have fewer cells and thus not actually impose its own row-minwidth on the table as a whole; I don't know if there's a way to ensure that the dependencies in that case are noncircular, but I suspect so.

Other approaches to the layout problem include, of course the CSS box model, and TeX's boxes-and-glue model, in which each character is a box, hboxes line up vboxes or vrules or characters on a baseline, and vboxes string together hboxes; TeX flows the text into a bunch of hboxes of length `\hsize`. Glue has nine-dimensional flexibility: a default size, a maximum shrink, a maximum stretch, and then three more optional transfinite levels of maximum shrinks and stretches. This allows space between words to be flexible, but not take advantage of that flexibility when you want to, say, center a line with equal infinite stretches on both left and right.

Quasi-unrelated: escape sequence tags

How do you specify things like font size, borders, padding, and the like? The standard ASCII approach is using escape sequences, but those have some big problems; they involve a mode switch, and so as you're typing them (or receiving them) you can't see what's going on. And if there's an interrupted transmission, they can eat some following text.

The org-mode approach to handling links is kind of nice. If you type `[[http://x.org/][foo]`, it's just text; but once you add a final `]`, it just displays as `foo` with underlines, and it's a link. Movement commands skip the hidden characters, but if you delete one of them, the pattern breaks and the remainder becomes visible.

WordPerfect's Alt-F3 "Reveal Codes" screen was one of its most

distinctive features. It displayed boldfaced escape sequences like [HRt] for a paragraph break ("hard return"), [UND] and [und] for beginning and ending underlines, [Just:Left] for a beginning of left-justification, and so on. But this didn't provide any clue as to how you would go about typing them. The current "WordPerfect" product still has such a feature, but now it draws little tags around the "codes".

HTML tags are nice but require a WordPerfect-like mode switch to make them effective.

So here's a thought. If we're going to have inline markup, maybe instead of *preceding* an escape sequence with an escape character, we should *follow* it. Maybe you write `just:left` or `border:black` or `href:http://x.org/` or `18pt` or `sparkline:3,5,1,18,12,19` in your document and then add a ^] (GS, group separator) to interpret the text back to and including the previous whitespace as an "escape sequence", thus removing it from view. This solves most of the problems of escape sequences and provides a sort of command line; it also allows a sort of immediate feedback as you're typing the escape sequence or receiving it, and it's harmless if the escape sequence is truncated. The choice of whitespace as the other delimiter ensures that a corrupted escape sequence can't eat too much text. If you're receiving data really slowly then you will have transient mysterious text appearing on the screen but that's hardly the worst display problem resulting from slow data reception.

Cursor positioning escape sequences

The best way for software to position the cursor in such a terminal, aside from cursor forward and back, is probably with escape sequences to push, pop, search, and move-to-end. Then you can insert text by sending it, or delete text with the DEL character, say.

Simplifying by dropping row/column transposition

All of the above ideas are complicated by the conflicting desires to support HT as an unambiguously horizontal separator and LF as an unambiguously vertical separator, on one hand, and on the other to conceptually have only three kinds of delimiter {|}; and also to simply be made of nested hboxes and boxes, on one hand, and on the other to support tabular layout.

Suppose that we resign ourselves to having four kinds of delimiter instead of three, and to making things out of nested tables instead of nested boxes. Let's say our tables do use HT and LF as horizontal and vertical separators, respectively, and we use two other characters for table nesting; let's say DC2 and DC4, ^R and ^T. The requested height of a table is the sum of the requested heights of its rows, and the requested height of a row is the maximum of the requested heights of its cells, and *mutatis mutandis* for width and columns. In a very simple model where cells can only contain either a table or a string, the width and height of the cell is just the width and height of the table or of the string; if a cell can contain a sequence of characters and tables in some way, then we need some kind of layout rule; several adequate possibilities occur to me.

(VT, ^K, and FF, ^L, are obvious choices for table *separators* but

not so obvious for table beginner-and-enders — should they begin nested tables, terminate them, or begin one and start a sibling? Perhaps better to use characters without so much baggage.)

This seems like it will be simpler to implement and to use, at least until we add rowspan and colspan and partial termination back in.

With these rules, there is the possibility that table cells receive a size that is different from what they requested, at least by being larger in either or both dimensions, and there are a variety of options as to how to handle that: left/top alignment, right/bottom alignment, centering, and stretching. You could extend this principle to the top-level document as well, enabling it to fill a larger window than it has explicitly requested.

A trickier question is how to handle canvases that are *smaller* than requested, for example to fit the entire canvas on the screen or to fit a column or row of the table into that entire canvas. The CSS2 `overflow` options here are `visible` (like TEX with overfull hboxes), `hidden` (discard, mostly like a terminal with line-wrap turned off), `scroll` (display scrollbars), and `auto` (which is `scroll` but hides the scrollbars when unnecessary); to this CSS3 (I think) adds `unset`. But this omits the choice of whether and how to do line-wrapping, which in CSS is socked away in the `white-space` and `hyphens` properties, and doesn't offer the traditional terminal option of wrapping per character rather than per word. An additional option used in, for example, Android's keyboard autocomplete suggestions, is to squish or stretch the contents on the relevant axis, to fill the box (such approaches to filling lines, impossible in the hot-lead era, are sometimes called "microtypography"), and the obvious other alternative is to repeat it, as in dot leaders for tables of contents.

(HTML layout doesn't normally deal with a *vertical* box becoming full, but this is a thing we might want in order to take advantage of modern big displays with multicolumn layouts, and it's necessary for rendering to paper.)

The usual way to handle overflow in ASCII terminal applications is to rely on applications that painted the screen to insert hard line breaks and extra spaces to wrap and justify paragraphs as desired, which clearly has drawbacks when you're using ASCII as a document format rather than an interactive screen-painting protocol, and accounts for the near-unreadability of most ASCII text documents on modern hand computers. So, having overflow-handling options would be pretty useful.

In particular, being able to flow table rows from the bottom of one page onto the top of another is pretty important.

TEX glue has a three-level stretchiness hierarchy — any amount of elasticity at a higher level of the hierarchy is infinite as seen from any lower level — and also separate stretchiness and shrinkiness quantities at each of these levels. Despite this level of complexity (I hesitate to say "expressiveness") the layout engine, except for paragraph filling, is a simple solver of a system of linear equations. (Not to undersell that — closed-form linear solvers use things like QR factorization which take $O(N^3)$ time, while iterative linear-solver algorithms like successive over-relaxation, though only $O(N^2)$, have complicated convergence characteristics.)

A remaining lacuna is how to handle ragged tables; above I suggested that each column should end when it reaches a row that

doesn't contain a cell for it, but perhaps there should be a special case that ends all the columns when there's a blank line (without even a space). Without such a special case, there's no way to end the first column other than ending the entire table with, perhaps, an additional ^T. The result would be that single-column lines of text would either tend to be outrageously squished by other columns, or would push those other columns over outrageously, even if they occurred far earlier in the text.

Layout properties

TEX has a zillion layout properties it stores in "registers", which are automatically restored to their previous value on the exit from the nesting level where they were defined. CSS does something similar: most properties are inherited from the parent node if they are not locally overridden, but properties set in a child node never bubble up to a parent node, much less just the part of the parent node following the child node. This is considerably saner than the approach that results implicitly from the state-machine model described by ANSI escape sequences and their more diverse predecessors.

Traditional terminals quantize layout and colors to character-cell boundaries, as well as abjuring texture and quantizing colors to a small number of primary or near-primary colors, with aesthetically unpleasant results. (You could argue that *traditional* terminals are black-and-white, or print on paper, but those aren't the ones we emulate nowadays.) On a bitmapped screen, just as on a phototypesetter, there's no need to quantize in this way, and on modern computers, even quantizing coordinates to pixels is no longer a problem. (Many computers nowadays have screens near 300 dpi, or 900 dpi horizontally with subpixel rendering, and antialiasing subpixel-positioned text is no longer the performance nightmare it once was.)

From HTML we know that a little bit of extra space around text before running into a color change can make an enormous difference to readability and aesthetics. Compare *this, with no padding,* to *this, with 0.2em padding.* (As of this writing, the ET Book font I'm using has substantial extra leading (vertical blank space) built into it, which shows up as built-in "padding" above and below, and building extra leading and letter spacing into your font in this way may be a reasonable hack to improve appearance in these situations.) Similarly, when text is highlighted in pastel colors, it's more readable and less visually noisy than when it's highlighted in primary colors. And a pastel background box is much more clearly visible with a thin border of the same color, but slightly darker.

In less Tufte-approved and more brochure-like directions, text drop shadows and background gradients are also often nice-looking effects, especially when used as accents rather than for running text, or on top of a visually noisy or poorly contrasting background that could interfere with text readability. Consider the difficulty of reading this problematic background and the same problematic background but with text-shadow, and then consider that problematic backgrounds can come from data plots, 3-D renderings, or photographs as well as poorly chosen static gradients.

Supporting all of these in a general-purpose layout syntax would be entirely reasonable, even without the complexities of the HTML box

model. You could quite reasonably have escape sequences that set such properties for the current table or (for character-level markup) for the span of text following the escape sequence, without importing all the complexities of the CSS box model and text-flow algorithms.

Page/column breaks and tables

On 2019-09-01, dpk commented on #swhack about the limitations of LaTeX tables; reformatted from IRC style, she said:

This combination of things in LaTeX is annoyingly apparently impossible: multiple-column mode; tables spanning multiple pages/columns; groups of rows within said table which should not be broken apart --- oh, and header rows on the said table which repeat after every column/page break.

`tabular` (built-in) can't break across multiple pages or columns at all, so you try the `longtable` package. First thing `longtable` does is complain that it doesn't work in multi-column mode, whether from the `multicol` package or with the built-in `\twocolumn`.

Some page recommends `supertabular`, which is apparently the predecessor to `longtable` but does work in multi-column mode, because why not remove features from things! `supertabular` seems to work, but then you notice you have groups of rows in the table (pairs, in my case) that really need to be on the same page/in the same column as one another, no matter what. `supertabular` doesn't appear to support this.

But, oh look, there's this package that lets you use the `\\*` command instead of `\\` to insert a row break that isn't allowed to cross a page or column break. the package is … `longtable`. Which you can't use.

Then you discover there's a hack to make `longtable` work in multi-column mode, … but it doesn't support printing the headers of the table anew at the top of each new column.

So, TL;DR, I am very annoyed.

This combination of requirements is probably a good one to keep in mind for designing table-layout systems, since evidently it's a combination of features that is difficult to provide if you didn't plan for it from the beginning.

Topics

- Graphics (p. 3483) (91 notes)
- Syntax (p. 3738) (28 notes)
- Graphical user interfaces (p. 3489) (23 notes)
- Terminals (p. 3743) (6 notes)
- Layout (p. 3544) (4 notes)
- Table layout

Resurrecting duckling hashing

Kragen Javier Sitaker, 2019-10-26 (updated 2019-11-10) (8 minutes)

Tonight I was reading *Software Abstractions*, by Daniel Jackson, Michael Jackson's son. and came across the specification of a hotel-room rekeying protocol that uses only keycards to communicate between the hotel front desk and the door locks. When it is desired to rekey the lock, thus locking out the previous "guest", the front desk issues a new key; the battery-powered lock can recognize both the current key and the new key, and upon recognizing the new key, it updates its "current-key state" to refer to that new key.

This is vaguely reminiscent of the problem [Stajano and Anderson] set out to solve: "secure transient association of a device with multiple serialised owners", the "owners" in this case being the hotel "guests", and the "device" being the door lock.

[Stajano and Anderson] https://www.cl.cam.ac.uk/~fms27/papers/1999-StajanoAnd-duckling.pdf "The Resurrecting Duckling, 1999"

The keycard protocol is an ingenious approach, but it can be refined further and applied more broadly.

The weak protocol suggested in the book

The suggestion in the book was that the lock could use, for example, an LFSR to compute the "new key" from the "current key". This approach is unnecessarily vulnerable to some attacks. Most simply, the "guest" could compute the successor of their own room key, and after checking out, come back to rob the new occupant. Slightly more annoyingly, they could compute the successor of the successor, and not only rob the new occupant but also lock them out. Furthermore, they could extend this attack arbitrarily far into the future, computing keys far into the future.

A better protocol

Given a one-way function, instead of computing a *successor* key, the door lock can recognize a *predecessor* key. It compares the presented key PK contents to its "current key" CK, opening if there is a match PK = CK; if that fails, it computes a one-way function of the presented key H(PK) and compares *that* to the current key. If there is a match H(PK) = CK, it updates its current key to be the new key, thus locking out the old key. (This "comparison to the current key" could be achieved by way of a salted KDF such as bcrypt() so that even with knowledge of the lock's memory contents, however acquired, it is infeasible to fabricate even a current key. That is, instead of storing CK, the lock would store some salt S and CKK = KDF(CK, S), and instead of comparing PK = CK, it would compare KDF(PK, S) = CKK.)

To initialize the lock, the front desk generates and stores a hash iteration count *N* of, say, a billion, and a random bitstring R; it hashes the bitstring *N* times — K[0] = R; K[1] = H(K[0]); K[2] = H(K[1]); ... K[N] = H(K[N-1]); and sets the lock's "current key" CK to the billion-hashed version K[N], and then it erases all of K except *N* and

R. To generate the new key, the front desk decrements N and repeats the computation, thus producing a key that will rekey the lock.

The crucial factor here is that the initial N should be large enough that we will not run out of keys before the door lock (or other device) needs to be repaired for some other reason, at which point it can be reinitialized. Even 9999 is likely sufficient for a hotel-room door lock.

As I understand it, this is more or less the OPIE or S/KEY approach, and it's probably in Merkle's dissertation, which I haven't read yet.

Performance optimizations

If the direct approach of performing nearly a billion hashing operations per rekey is too slow, it is straightforward to cache hashes K[N - 1], K[N - 2], K[N - 4], K[N - 8], and so on, such that the amortized average number of hashing operations required per rekey is just under 2, at a logarithmic space cost — 30 keys for this example.

Most of the cache space is used for the last few levels — if you could afford a million hashing operations per rekey, you could cache only 10 keys instead of 30, still getting a thousandfold speedup over the cacheless case. That is, from a certain point of view, 99.9% of the speedup comes from 33.3% of the cache.

As usual, in exchange for a higher space cost, the amortized bound can be made into a worst-case bound. The first rekey operation consumes the cached key N - 1 and computes keys N - 3 and N - 7; the second rekey operation consumes N - 2 and computes keys N - 6 and N - 5; the third consumes N - 3 and computes N - 15 and N - 14; the fourth consumes N - 4 and computes N - 12 and N - 11; the fifth consumes N - 5 and computes N - 10 and N - 9; the sixth consumes N - 6 and computes N - 31 and N - 30; and so on. In this case the space cost becomes linear if we pay only two hashing operations per rekey, which is likely unacceptable. I think that if we pay four hashing operations per rekey, we can keep the space cost logarithmic, though still up to twice the space cost of the amortized-only algorithm.

Reverse metempsychosis and imprinting an ignition key

[Stajano and Anderson] suggest that an IoT device should "imprint on" or "be ensouled by" the first public key it receives when it hatches; this allows the owner to establish a relationship with it that its manufacturer cannot participate in. The rekeying protocol plays the role their paper calls "escrowed seppuku", allowing the manufacturer to command the duckling to die and prepare to be ensouled anew.

Extension to further applications

Because this protocol only depends on the strength of a hash function, it's a good candidate for being resisting quantum cryptanalysis, which is good news, since Google has just announced that they have demonstrated quantum supremacy. But in the form above it can't do very much; it can't even authenticate any other information on the keycard, such as the "guest"'s checkout date or

public key, since that other information would break the hash chain.

Suppose that instead of computing the keys by merely hashing their predecessor, we additionally generate a private-key seed D[0], and compute a series of private-key seeds D[1], D[2], ... D[N] from it by the recurrence D[I] = H(D[I - 1]), as we were doing before with the room keys. And suppose we have some computation A to generate a public key from one of these private key seeds: Q = A(D[I]), which perhaps also generates a private key into the bargain. Now, instead of computing the new room key as K[I] = H(K[I - 1]), we compute that key as K[I] = H(K[I - 1] || A(D[I])), and we put both K[I] and A(D[I]) on the keycard. The verification step changes accordingly.

Now, however, the room lock can verify both the newly presented key and this other datum A(D[I]), which it does not understand and therefore could not have used to verify that it was used to derive the A(D[I + 1]) that it previously held. The front desk, however, can perhaps use the corresponding private key to sign other attestations that the room lock can verify, at least until its next reverse metempsychosis.

Topics

- Programming (p. 3658) (286 notes)
- Systems architecture (p. 3691) (48 notes)
- Protocols (p. 3668) (21 notes)
- Security (p. 3701) (9 notes)
- Cryptography (p. 3397) (9 notes)
- Postquantum

Home dehumidifier

Kragen Javier Sitaker, 2018-05-20 (updated 2019-04-02) (12 minutes)

Aside from uses in things like rescuing rare books from water damage, reducing the water content of hygroscopic plastics prior to molding, and reducing the water content of natural gas to discourage clathrate formation, dehumidifiers and desiccators are potentially useful human household items — indeed, in raw vegan circles food dehydrators are already ubiquitous, as low-temperature dehydration is a crucial preparation step in many raw vegan dishes. In developed countries, clothes dryers are also ubiquitous, to the point that hanging laundry to dry is prohibited in some neighborhoods, and of course a hair dryer is precisely a personal keratin desiccator.

Perhaps a more general desiccation appliance would be useful.

Household applications for desiccation include the following:

• Human food, as mentioned above, both for preservation and to enhance flavor and alter texture. This has limitations on the temperature it can tolerate — raw vegans normally use a temperature limit of 40.5°, but different foods have different optimal ranges.
• Garbage — human food waste is commonly 80% to 95% water by weight. So the food waste stream can be reduced by a factor of 5 to 20 by desiccation. Desiccation also stops the food waste from decaying before further handling. Furthermore, aerobic composting requires a humidity level within a certain optimal range, a range food waste normally exceeds. Incineration suffers even more from excess water than composting does. The usual solution to this problem is to add additional dry matter to the compost, but probably a better solution is to remove the excess water. At Burning Man, we use a very simple solution to this: we hang the garbage outside in net bags, where the sun and dry wind desiccate it within hours.
• Poop — much the same set of concerns applies as those described above for pre-eating food waste, but the desiccation process is more demanding, as the air will pick up shitty odors as well as water, and thus needs postprocessing before release into the environment.
• Dishes — drying dishes with a towel is prohibited by many human health codes, because the towel can both spread pathogens from one dish to another and provide a suitable medium for the proliferation of certain airborne pathogens. Active drying, as performed by dishwashers, can solve this problem.
• Laundry — of course a clothes dryer is precisely a desiccator. The modern unvented clothes dryer uses a heat exchanger to condense the water and pump it into a holding tank or drain.
• Moisture harvesting — humans typically die some 32 to 64 hours if they cannot ingest water, but can survive 1024–4096 hours without food. But they do not need very much water, typically only 2–4 ℓ/day (50 $\mu\ell$/s). When drinking water supplies are disrupted, they could survive much longer with a system that can recycle drinking water from breathed air, from evaporated sweat, from garbage, and from passing air.
• Cleaning tools such as mops — these are wet when in use, pick up a wide variety of micronutrients, are often composed of carbohydrates

or protein, and can sometimes remain wet for a long time after use. This makes them good environments for the growth of microorganisms and fungi, which can be pathogenic to humans, smell bad to them, or both.

• People after bathing; hair dryers are one example, but an "air shower" of high-speed warm wind could eliminate the need for towels and the labor of washing them. This could even be installed in the same physical place as the regular shower.

• People after dying: the decomposition of human corpses involves many microorganisms that produce smells that humans find very disagreeable, and they can even be pathogenic to them. Also, they can host macroscopic organisms, such as flies, which also create health hazards for humans. Desiccated corpses do not have these problems. (In fiction such as *Dune*, they are also an important source of drinking water, but this seems unlikely to ever happen on Earth.)

• Coolth: evaporating water cools the system by the water's enthalpy of vaporization, and this is an important way to cool human habitations in dry places like the Southwestern USA and Iran — and of course humans' voracious appetite for water results from their biologically programmed use of water as sweat for the same purposes, but in other places the available water and the pre-existing humidity of the air limit the possibilities. Most of the uses mentioned above for a desiccator could additionally be taken advantage of to produce a lower-than-ambient temperature.

• Making aerogel, which requires supercritical drying to prevent the hydrogel to a xerogel.

These uses seem like they might require multiple different desiccation appliances. Poop will probably contaminate the poop desiccator in such a way as to make it unsuitable for desiccating laundry, dishes, food, and possibly even food waste. Poop, human corpses, and food waste should probably be shredded before desiccation in order to speed the process; this is never desirable for living people, mops, laundry, or dishes, and only sometimes desirable for food. Vacuum, as discussed below, may be a useful way to speed up desiccation for some things, but it is fatal to living humans within seconds, and thus its routine use after bathing could be counterproductive.

However, it seems like a general-purpose desiccation appliance could perhaps serve a substantial range of uses.

Physically, evaporation is a complex thermodynamic process, but the main determinants are temperature, airflow rate and turbulence, air humidity, and air pressure. Higher temperature, faster and more turbulent airflow, drier air, and lower pressure all speed up evaporation. The interaction among these factors is complex. The usual simplified model is that a thin boundary layer of air next to the moist thing is 100% saturated with humidity, and the air above that has a linear humidity gradient down from 100% down to the ambient humidity, driven by diffusion and advection. But the thickness of the saturated boundary layer also varies with airflow.

The exponential rise of water's vapor pressure with temperature means that even a small temperature difference can make a big difference in the air's moisture capacity, and the amount of water that can evaporate is driven by the difference between the air's moisture capacity and its pre-existing moisture content.

I don't have a good handle on the energy costs of using these different factors to accelerate desiccation.

Oven desiccation experience

I've been using my apartment's electric oven over the last few months to desiccate food waste before discarding. It has a thermostat, a timer, and a circulating fan; I set the thermostat for 70° (160 archaic degrees) and the timer to an hour or two. 70° is hot enough to nearly sterilize foods, while not being hot enough to set common foods on fire in any reasonable period of time. The oven has the major disadvantage that the timer is disabled by power outages — following a power outage, the oven turns back on with no timer — so it isn't safe to leave it unattended. Commonly I put the food on a glass plate or a sheet of aluminum foil before starting the oven, since I can clean or discard those easily if food gets stuck to them.

Vegetable wastes, even onions, generally smell pleasant during this procedure, which also reduces them greatly in volume and perishability. I try to ensure that they are sliced to a thickness of 10mm or less to accelerate the process. Animal cadaver bones lose less mass and smell less pleasant — they smell like cooking meat — but the smell is far less objectionable than rotting meat, which is what they would become in the garbage after a couple of days without pre-desiccation. Eggshells have no noticeable smell and become brittle quickly.

I'm currently trying to compost some of the results from this process, without any desiccated animal remains, but adding my own hair. It seems to be working somewhat — at least the compost has a reasonably earthy odor rather than smelling like rotten fruit, and it had only a few flies at first, nothing since. Most of its mass comes from yerba mate and eggshells. I suspect that the eggshells, which are made of calcium carbonate rather than calcium phosphate like bones, provide a pretty robust carbonate buffer against acidification, but I haven't actually measured the pH of the compost.

In any case, desiccating food waste before composting it allows you to delay composting until you're ready, which can be valuable when you're getting a compost heap started and makes it easy to avoid anaerobic conditions in the compost.

Presumably, oily food could trap water inside a layer of oil, once water was removed from the outer layers of the food. I'm not sure I've observed this; even meat bones seem pretty dry if you crack them open after this process. Maybe it doesn't happen; maybe water molecules can diffuse through the oily layer anyway, or maybe water has lower surface tension at higher temperatures and so it doesn't push and pull the oil into a coherent layer, or maybe the water doesn't have sufficiently coherent surface tension when it's just occupying the pores of the foodstuff.

For a while, I was using a generic tupperware lid as a plate, since it was even easier to clean than glass plates. After lasting through dozens of cycles, one day half of it melted down onto the plate on the wire rack below it. The melted and resolidified plastic was stiff and brittle; the unmelted part remained flexible, resilient, and ductile, despite otherwise being similar in appearance. I'm not sure what happened; three possibilities occurred to me:

• Maybe this time the oven got hotter than on previous occasions, but only part of it — the circulating fan was not as effective as usual. So the hotter part of the lid melted and degraded, perhaps through hydrolysis from moisture absorbed into the plastic. In this possibility, different levels of heat on this one occasion caused one part of the plastic to both melt and chemically degrade.
• Part of the lid had come under chemical attack from some food I'd placed on it, either at that moment or in the past; perhaps it absorbed some oil, plasticizing it further and lowering its melting point. Oil by itself wouldn't explain the brittleness, but maybe other factors could. Acid, for example. In this possibility, different levels of chemical degradation in different parts of the plastic caused the more degraded part of the plastic to melt.
• Perhaps the lid had lost plasticizers to evaporation over time in the oven. If this were the case, though, you'd expect the part that had lost more plasticizers (and was thus more brittle) to be the *last* to melt, not the first.

Regardless, I don't plan to use plastic trays in a desiccator in the future without thoroughly qualifying the plastic for handling the relevant temperatures.

Topics

• Household management and home economics (p. 3504) (44 notes)
• Chemistry (p. 3373) (20 notes)
• Garbage (p. 3468) (10 notes)
• Cooking (p. 3392) (10 notes)
• Drying (p. 3417) (7 notes)
• Sewage (p. 3708) (4 notes)

Dercuano calculation

Kragen Javier Sitaker, 2019-05-01 (3 minutes)

A lot of the notes in Dercuano contain calculations. To take a random example, Lab power supply (p. 2421) says:

> It could maybe dissipate like 20W, which at 12V would be just 1.7 amps…
>
> Supposing arbitrarily that I were to use a similar ATX power supply capable of 18A on its 12V output, which works out to 216W (a bit over a quarter horsepower), it would be nice to be able to carry that 216W most of the way down the range, say down to 2V — which would mean 108 fucking amperes.…
>
> How much energy do we need to store in the inductor at 62.5kHz? That's 16 microseconds per cycle. I'm a little unclear on exactly how the math of buck converters works out but I am pretty sure that it will not involve storing more than 16 microseconds' worth of power in the inductor, which would be three or four millijoules, and I'm pretty sure it's okay for the inductor current to fluctuate by 10% or so, maybe a lot more. So if ½LI² = 4 mJ and I = 18 A, then L = 2·4 mJ / (18 A)² = 25μH,
>
> Refreshing at 1kHz (again, as suggested in the display datasheet) would require iterating at 11kHz. At the AVR's internal RC oscillator speed of 8MHz, this gives us 727 clock cycles per display update…

These calculations are unfortunately "dead", not "live" like the calculations in a spreadsheet, and by the same token, they're time-consuming to verify — if you want to check to see if my calculations are wrong, you need to enter each number into a spreadsheet or calculator and redo the calculation from scratch. I typically use units(1) to do the calculations (as described in Executable scholarship, or algorithmic scholarly communication (p. 2137)) which reduces my chances of accidentally dropping a "milli" or a "kilo" from the calculation, or using the wrong conversion constants, but if you're using Gnumeric or whatever, you have to check those too.

And the calculations are just individual points, while the underlying formula is a much richer interrelationship. If you're looking at the ½LI² formula above (see also Dercuano formula display (p. 495)) and you wonder how high the inductance L would need to go to support 20 A of current, you need to think about it for a while. If you're not used to this kind of thing, the answer may require some square roots and stuff.

Such calculations can be made "live" by generating them from some kind of JS library that does calculations. I'm thinking that something like my prototype RPN editor could be developed into a comfortable way of computing the quantities and displaying how they were derived, although it does require a context switch from editing plain text.

Topics

- Programming (p. 3658) (286 notes)
- Human–computer interaction (p. 3493) (76 notes)
- Dercuano (p. 3406) (16 notes)

Things in Dercuano that would be big if true

Kragen Javier Sitaker, 2019-05-24 (updated 2019-08-21) (24 minutes)

if (true) {

As I pointed out in the Dercuano introductory text (p. 1), Dercuano contains much that is correct and original, but mostly what is original is not correct, and what is correct is not original. I think that phrase originated as a clever insult to somebody's poor work, but in a sense it's just the default state of human cognition: most of the new ideas we come up with are wrong, while most of our ideas are not new, and since correct ideas have better memetic fitness (all else being equal) our unoriginal ideas tend toward correctness. With enough focused effort it's possible to figure out which original ideas are true, and if I were capable, I would have made that effort before making Dercuano public, but I haven't managed it in many years.

On the other hand, there's a third axis along which ideas can be evaluated, aside from (probable) correctness and originality: consequences or interestingness.

What's a big idea? What are consequences?

In Approaches to 3-D printing in sandstone (p. 1095), for example, it says that in Argentina in 2017, ordinary gray portland cement cost US$0.26 per kg, while the white grade cost about three times as much. Conceivably nobody has made this observation before, and quite probably it is a correct observation, so it is likely correct and, in a minimal sense, original. But it really matters very little whether the price ratio was 1:2 or 1:3 or 1:4 in Argentina in 2017, though conceivably that may someday be of interest to some historian of concrete; this knowledge enhances your capabilities very little.

At the other end of the spectrum, consider Becquerel's observation in 1896 that, even in the dark, potassium uranyl sulfate blackened photographic plates left nearby, as if they were spontaneously emitting X-rays, which of course they were. The observation was hardly more *creative* than my note above about the ratio of prices of different kinds of cement, merely an observation of an unexpected and unexplained labwork problem in a footnote of a paper. However, upon further investigation, this observation answered the mystery of how the sun could keep burning for billions of years; provided a source of energy that did not emit CO_2 and required a tiny amount of fuel to a substantial part of humanity; made it possible to send space probes to the outer planets; changed the nature of warfare and ended World War II; revealed the existence of entirely unsuspected types of matter in the universe; and was a key part of the evidence for special relativity, which revealed that mass and energy were not two separate quantities, but the same quantity.

But the consequences of an idea are very situational, whether we're talking about its logical consequences (the other propositions that its truth would entail) or its practical consequences (the results in the

contingent world of its putative truth becoming known).

From the proposition, "Socrates is a man," we cannot deduce that Socrates is mortal; nor can we deduce it from the proposition, "All men are mortal." But if either proposition is known, the other has as a logical consequence the proposition that Socrates is mortal. So it is that the logical consequences of an idea depend on what else is known.

The practical consequences of Hero's aeolipile were, famously, almost nil; but under somewhat different historical circumstances, steam-engines revolutionized industry in the 18th century. Condorcet voting made no impact on the USA's political processes for at least two centuries, and the USA continues electing incompetent demagogues; Condorcet voting ensures that Debian's project leaders are widely respected. Oil drilling in the Song dynasty lowered the price of salt; oil drilling in Pennsylvania made horse-drawn carriages obsolete. Cellphones in the US were relegated to executive status toys until the 2000s; cellphones in India allowed farmers and fishermen to capture what were previously middlemen's profits. Movable type in the Song enabled the preservation of much of the Chinese literary canon, while movable type in Europe gave birth to the Reformation, liberalism, and the Westphalian state. So it is that the practical consequences of an idea depend on what else is practiced.

Here I'm not concerned with the practical consequences of "big ideas" that turned out to be false, like the inevitable withering away of the socialist state or the inevitable triumph of Daesh over "Rome", but only ideas whose consequences would be big *if true*.

So, what ideas in Dercuano could have big consequences, if they turn out to be correct? And why?

Self-replication

One of the main themes of the last several years of Dercuano has been "clanking replicators" — more precisely autotrophic programmable self-replicating 3-D printers (p. 3703), and especially how to achieve autotrophic replication of the control computer necessary to control the printer's actuators.

A workable self-replication design is big, if true, because it totally upends the principles of economics, in a way which I think will substantially improve the material well-being of the average human by reducing opportunities for oppression. I think the change will be more important than the Industrial Revolution, more important than the development of agriculture, possibly more important than fire. I go into somewhat more detail on the expected economic effects in Exponential technology and capital (p. 406), Gardening machines (p. 2365), and Self replication changes (p. 2842), and on how to prevent disasters in Approaches to limiting self-replication (p. 1004), and there's a fictionalized near-future scenario of less-radical digital fabrication technology in 2025 manufacturing and economics scenario (p. 699).

However, on looking at Predictions for future technological development (2008) (p. 341), it's obvious that my ability to forecast what the future holds is pretty poor, and strongly affected by wishful thinking.

The benefit of self-replicating 3-D printers in practice will be

limited by the price of energy, whether that price is measured in a conventional way with currency or in more fundamental terms of natural resources, labor power, and capital investment; but energy should become much more abundant soon due to the uptake of solar photovoltaic energy — see the section below on the solar energy transition.

The problem of self-replication can be crudely divided into the problem of designing a *cyclic fabrication system*, a term I'm possibly abusing to mean a set of material-processing, part-forming, and assembly processes which individually consume one another's outputs but collectively consume only natural materials; and the special problem of how to put together a *computing system* that's fast and reliable enough to direct the cyclic fabrication system to produce the desired product, without requiring exotic materials and geometries those processes can't themselves produce. In particular, alternatives to the very challenging processes used to fabricate modern mass-produced semiconductors would be very welcome, keeping in mind that the economics are very different.

An overview of the whole problem is in Simplified computing, down to the level of mining raw materials (p. 691).

So I explored alternative digital logic technologies in mechanical computation: with Merkle gates, height fields, and thread (p. 2494), Nobody has yet constructed a mechanical universal digital computer (p. 1053), Ideas to ship in 2014 (p. 1409), Simple state machines (p. 760), An extremely simple electromechanical state machine (p. 50), Steampunk spintronics: magnetoresistive relay logic? (p. 1315), Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times? (p. 1027), Making a mechanical state machine via sheet cutting (p. 1013), Transmission line diode computation (p. 3090), Diode logic (p. 272), Snap logic (p. 2580), Hall-effect Wheatstone bridges for impractical steampunk electronic logic gates (p. 2351), Nonlinear differential amplification (p. 2949), Paper/foil relays (p. 3273), and Non-inverting logic (p. 861), largely with an eye to things that could be built without million-dollar semiconductor fabs. Clanking replicators (p. 171) touches on this a bit too.

In another direction, though, the topic control (p. 3390) largely talks about negative-feedback control, including speculative sensor approaches like Charge transfer servo (p. 3017), Starfield servo (p. 1709), and Servoing a V-plotter with a webcam? (p. 62), as well as codesigning physical and control systems for feedback control in High-precision control of low-stiffness sytems with bounded-Q resonances (p. 1002); and things like Differential spiral cam (p. 512) cover control systems that aren't purely digital, which could reduce the demands on the digital part of the system.

When it comes to the materials-processing side of things, I've done some overviews like 2016 outlook for automated fabrication and 3-D printing (p. 2316) and much of the notes in The book written in itself (p. 2400). I've come to the conclusion that Minecraft is misleading; you start with fire, then clay. Any practical *terrestrial* cyclic fabrication system will probably begin with clay ceramic. So in addition to the materials (p. 3560) category, there's a ceramic (p. 3371) category, and Clay fabrication objectives (p. 2111) talks specifically about what to do for clay, and Flux deposition for 3-D

printing in glass and metals (p. 1366) and 3-D printing by flux deposition (p. 466) talk a bit about some processes that I think might work well. More broadly, the manufacturing (p. 3558) category has notes on many different manufacturing processes, and digital fabrication (p. 3411) has notes on digital fabrication processes, some existing and some speculative. Elastic metamaterials (p. 719) talks about workarounds for the limitations of inorganic materials at room temperature, while Plastic cutters (p. 1074) describes a way to minimize the amount of very hard material needed if cutting is one of the processes in the CFS.

Other notes on existing or possible material-shaping processes include Hot wire saw (p. 3159), String cutting cardboard (p. 515), Hot oil cutter (p. 3287), Regenerative fuel air cutting (p. 2622), Laser ablation of zinc or pewter for printed circuit boards (p. 2799), Filling hollow FDM things with other materials (p. 2119), Hot air ice shaping (p. 864), Friction-cutting plastic (p. 2412), Single-point incremental forming of aluminum foil (p. 769), and Sun cutter (p. 56). Freeze distillation at 1 Hz (p. 2796) is a possible material-refinement process, and Spark particulate sieve (p. 2047) covers a possible way to make an air or water filter or mesh for grading solid powders. Cold plasma oxidation (p. 2406) describes a process that is commonly used today for surface treatments, but which I think can also be used for some kinds of cutting and 3-D printing. And at the end of Caustics (p. 1619) and in You can't construct optical systems with arbitrary light transfers, but you can do some awesome shit (p. 981) there are some brief speculations on optical-surface fabrication.

Assembly processes like those explored in Maximal-flexibility designs for printable building blocks (p. 1839) are useful not just for humans, but also potentially for machines, as they can produce macroscopic tight tolerances using low-precision assembly processes. "Voxel printers" is a recent marketing buzzword related to this.

So, in these areas, what in Dercuano might be an idea with big consequences, such as enabling autotrophic self-replication? Because probably Laser ablation of zinc or pewter for printed circuit boards (p. 2799) isn't it — I mean, even if it does work, it's probably only an incremental improvement.

The family of 3-D printing processes described in 3-D printing by flux deposition (p. 466) is applicable to many areas and should extend the range of additive 3-D printing significantly. By my count, at present, it discusses some 27 candidate combinations of materials; I have confidence that at least some of them should work. If they are tried soon, and work, that could be a significant advance; presumably if nobody tries them until 2152 it will be a different story.

If one of the numerous alternatives to semiconductor logic mentioned above works reliably, can work at at least a MHz or so, and can be fabricated under less demanding conditions, that would also be big, if true — again, if tried soon enough. Again, there are enough of them that it's almost guaranteed that some of them will work.

Dirt-cheap precision optics

Self-replicating machines will probably need optics, at least for cameras (see the section about sensors below) and quite likely also for

solar furnaces. But existing approaches to optics fabrication are very expensive, especially for surfaces far from sphericity.

So there are several notes in Dercuano that propose new optics-fabrication processes: Jello printing (p. 2426), Caustics (p. 1619), You can't construct optical systems with arbitrary light transfers, but you can do some awesome shit (p. 981), and Flux deposition for 3-D printing in glass and metals (p. 1366). Any of these would be a substantial advance over existing methods if they work, and this would have significant consequences for achievable optics, entirely apart from self-replication.

Much better sensors

The sensors (p. 3706) category right now consists of five big ideas.

Starfield servo (p. 1709) outlines a way to make some simple physical objects and less-simple algorithms that would enable a cheap webcam to become a remote multiple-degree-of-freedom sensor with, in some dimensions, submicron resolution over a range of a few meters. If it works, and I think it will.

Compressed sensing microscope (p. 306) describes the same technique applied to light microscopy, where it should enable subwavelength near-field imaging without lenses.

Measuring submicron displacements by pitch bending a slide guitar (p. 905) outlines a totally different way to measure submicron displacements over a range of a few meters with inexpensive equipment — electric-guitar pickups, this time, rather than webcams.

The Tinkerer's Tricorder (p. 72) outlines a variety of hacks to build an inexpensive LCR meter similar to the popular M328.

Ghettobotics: making robots out of trash (p. 2747) (and category ghettobotics (p. 3472)) explores how to build

> a self-sustaining industrial economy that consumes nothing but discarded electronics and other trash and produces, with a minimal amount of human effort, useful robots.

It's sort of Self Replication Lite™.

Archival

The archival (p. 3322) category covers lots of possible ways to archive the humans' knowledge to keep it from being lost, at many levels of the stack: physical substrates for information, ways of mass-producing the physical substrates to reduce chances they will all be destroyed, file-format compatibility, and archival virtual machines to guarantee file-format compatibility.

So, for example, Atmospheric pressure harvesting phoenix egg (p. 2081) describes a power source that enables you to build a computer that could continue to run for centuries even while buried, barring too many hardware failures; Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) discusses how to structure interactive hypertext to make archival possible, as do Instant hypertext (p. 630) and Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257).

Some extensions of William Beaty's scratch holograms (p. 2536) describes a way to archive large amounts of information on inexpensive, durable materials in a way that the humans can read without needing a working computer, as do Caustics (p. 1619), Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p.

1455), Rosetta opacity hologram (p. 98), Holographic archival (p. 766), Piezoelectric engraving (p. 1070), Quadratic opacity holograms (p. 3073), Archival transparencies (p. 1345), and A mechano-optical vector display for animation archival (p. 3047).

In between those approaches, there's the possibility of archiving large amounts of information in an executable digital form, but providing a specification for an "archival virtual machine" that can execute the archived information, as proposed by Raymond Lorie and by Nguyen and Kay's "Cuneiform Tablets" paper. Attacks on this problem include Bootstrapping instruction set (p. 459), A simple virtual machine for vector math? (p. 986), Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952), Designing an archival virtual machine (p. 3203), XCHG: An Archival Swap Machine (p. 2997), Archival with a universal virtual computer (UVC) (p. 399), and The Dontmove archival virtual machine (p. 2113).

As an alternative to making time capsules to bridge periods of time when the humans are uncooperative, we might be able to preserve history by enlisting their cooperation; Viral wiki (p. 1024) discusses one approach to that.

If one or more of these approaches is successful at rescuing the humans' knowledge from the Digital Dark Age, that would indeed be Big. (But it's not clear how much of that depends on correctness; it probably depends more on implementation effort.)

Hessian-free quasi-Newton methods

In Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) it is claimed that quasi-Newton methods require maintaining in memory an approximation of the Hessian, while perhaps a similarly quadratic order of convergence can be obtained with just the gradient by using Newton–Raphson iteration along the direction of the gradient, while gradient-descent methods only have linear convergence. If all of that is true, which is unlikely, then it contains a numerical optimization method that is many orders of magnitude faster than the state of the art in high-dimensional spaces.

More generally, I think mathematical optimization (p. 3611) is a significant candidate for including in More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) as a basic element of computer systems design; $1 recognizer diagrams (p. 1264) gives an example of how you can use it to replace ad-hoc procedural algorithm design with something much simpler, which I'm pretty confident will work.

High-density fractal heat exchangers

In Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) it is claimed that a particular three-dimensional fractal design for a recuperator-type heat exchanger could provide recuperators with orders of magnitude higher performance, rivaling that of regenerators. This could be a crucial enabling technology for many kinds of thermodynamic machines, including heat engines (possibly including micro-turbine generators) and climate-control systems (for example, A design sketch of an air conditioner powered by solar thermal power (p. 2233)).

The solar energy transition

One of the largest changes in the material culture of the humans during the 21st century will be the transition away from fossil fuels as their main source of harnessed energy, since the alternative is global warming that may be sufficient to cause a mass extinction; right now it looks like they'll change to solar photovoltaic energy during the late 2020s.

Since most of the resource cost of producing photovoltaic panels is an energy cost, but their EROEI is already quite high, this probably means a rapid exponential growth in the amount of energy available to be harnessed for human activities. This will make energy much cheaper than it's ever been.

The economics of solar energy (p. 1175) is a somewhat dated overview of the basic issues, which are also discussed in The future of the human energy market (2014) (p. 1846), Japan can achieve energy autarky via solar energy, but not much before 2027 (p. 2819), and parts of Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566).

One of the predictable effects of abundant marketed energy is cheaper desalination and an end to water stress. See A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519), Fast sea salt evaporator (p. 1087) , and Calculations about desalination in Israel (p. 2827).

Among the more dramatic results of this transition is that, until there are intercontinental HVDC lines or breakthroughs in utility-scale energy storage, energy is going to be a lot cheaper during the day than at night, which means that "demand response" is going to be really important for taking advantage of the available energy. In Salt slush refrigeration (p. 1230) and Household thermal stores (p. 1533) it is discussed how to do household refrigeration and climate control in a demand-response-friendly way.

It is, however, *possible* to build enough storage with existing lithium battery technology to sustain current energy usage levels during the night; Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) discusses the available resources, and Energy storage efficiency (p. 1300) discusses the economics in more detail.

Replacing fractional-reserve banking

The evils of fractional-reserve banking are a favorite hobbyhorse of economic cranks and conspiracy theorists. I don't think it's evil, and moreover I think the standard economic-crank position dramatically overstates its importance, but it does have some real problems, such as bank runs, and I think that now we can do better; Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) explains how.

Fast parsing

A dismaying quantity of current computer software amounts to ways of caching parsing results because parsing is so slow. One attack on this problem is to use a data structure serialization format like FlatBuffers that permits random access; another is to use faster parsing algorithms. In Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) I suggest ways to

increase parsing speeds to a sufficiently high level that much of that caching code can be thrown away. They might work.

Unified caching

The old joke is that there are three hard problems in computer science: naming, cache invalidation, and off-by-one errors.

"Cache invalidation" is the process of determining when some cached result should be updated, which is a very general concept, and different kinds of caches are ubiquitous in computer systems architecture, at every layer from RTL design up to container orchestration, for reasons that include improving throughput, protecting privacy, tolerating faults, reducing average latency, and reducing worst-case latency. The vast majority of complexity in computer systems does in fact amount to logic that manages different kinds of caches.

I have found several promising ways to unify many, though not all, of those caches in a single caching subsystem, which will dramatically simplify computer systems design at the same time as dramatically improving performance, if one of them works.

In A minimal dependency processing system (p. 911), Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870), Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257), Automatic dependency management (p. 881), and Immutability-based filesystems: interfaces, problems, and benefits (p. 1672), I discuss ways to architect computer systems that simplify this problem; Transactional screen updates (p. 2907), Caching screen contents (p. 2362), and Cached SOA desktop (p. 2229) focus specifically on the problem of GUI caching, because it's a particularly demanding aspect of the problem that illuminates it from a particularly useful angle. In Memoize the stack (p. 2021) and Amnesic hash tables for stochastically LRU memoization (p. 502) I discuss particular generic algorithms that might be useful.

More generally, the topic "caching" (p. 3361) covers many different aspects of the problem.

Paper/foil relays

In Paper/foil relays (p. 3273) I describe an electrostatic relay design that might be feasible at millimeter scale and below to get reasonably-fast digital logic without any advanced materials processing. Unlike electromagnetic relays, these work better at smaller scales.

Very efficient 2-D convolution with flat kernels

In Real-time bokeh algorithms, and other convolution tricks (p. 2661) I explored a number of algorithms for simulating camera bokeh and discovered a general convolution algorithm for kernels with a small number of discrete multiplier values which occur in large contiguous blocks — such as, for example, camera bokeh kernels for ideal lenses. If it works, it's an order of magnitude or more faster than any previously published algorithm for this problem, beating even McGraw's approximate algorithm (although it can handle cases with

spherical aberration, which my algorithm can't.)

"Interaction models"

In Dehydrating processes and other interaction models (p. 3208) I propose a sort of taxonomy of computer systems architectures based on something called "interaction models", which has to do with the relationship between individual programs and the rest of the system. It could stand to be sharpened up a bit.

Expressive multitouch

Historically, human–computer interaction (p. 3493) has mostly been through a keyboard, screen, and mouse. The screen provides perhaps 10 megabits per second of output bandwidth, while the mouse provides perhaps 50 bits per second of input bandwidth (but limited to about 6 bits per second in practice, according to the experiments in Some musings on applying Fitts's Law to user interface design and data compression (p. 1164)), and the keyboard another 15 or so, for a total of about 25 bits per second. While this vast disparity hasn't been much of a limitation for consumption-type activities like watching the Gangnam Style video or playing Flappy Bird, it's a major limitation for using the computer as a means for creative expression — "magic ink" or a "bicycle for the mind", in the phrases of Steve Jobs and Bret Victor. And it is ultimately through creating great things, not consuming great things, that the humans can become great themselves.

Multitouch input devices have dramatically higher input bandwidths than mice or keyboards, so in theory they might offer dramatically greater expressive freedom to computer users. Sadly, despite some promising prototypes demonstrated by Bret Victor and other researchers, the humans mostly interact with them by scrolling vertically with one finger, selecting pre-existing options by tapping on them, or using an on-screen keyboard.

We could escape this multitouch Skinner-box Sheol with the ideas in Two-thumb quasimodal multitouch interaction techniques (p. 1765), Interactive calculator (p. 2771), drag-and-drop calculator for touch devices (p. 1045), Interactive geometry (p. 508), $1 recognizer diagrams (p. 1264), Multitouch livecoding (p. 122), and Dercuano drawings (p. 64), if those ideas work out.

The Magic Kazoo

Maybe you can build a synthesizer kids of all ages can play by humming into it, and that would be a big hit; that's what The Magic Kazoo: a synthesizer you stick in your mouth (p. 1873) is about.

Ultralight tunnel personal rapid transit

In Ultralight tunnel personal rapid transit (p. 706) I calculate the performance of a new kind of rapid transit system, one so much more efficient and frugal that it would enable entirely new kinds of urbanization, combining the benefits of suburbia with the benefits of dense cities; Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) speculates on the kinds of sustainable, resilient community living that could result.

Topics

Incremental roller comb forming

Kragen Javier Sitaker, 2019-11-27 (4 minutes)

If you drag a weighted rake over gravel, it forms the gravel, sort of plastically, leaving little valleys behind the rake tines. Similar phenomena occur with many soft materials and metamaterials: sand, styrofoam, oil films, wet clay, wet concrete, foamed metals, foamed waterglass, hot glass, mashed potatoes, and so on. With sufficient pressure, you can get it to happen in plastic materials like wax, aluminum, steel, and so on; lubrication and a super-hard, well-polished comb may be necessary.

Suppose that you put rounded wheels, shaped like rollerblade wheels but made of harder materials, in the tips of the comb tines. This should allow you to do this kind of plastic or pseudoplastic forming to fairly hard materials by virtue of increasing the force you can feasibly apply before the friction becomes prohibitive. Sheet-metal "spinning" is done this way, though usually with only a single handheldd wheel.

By itself, this is not a very interesting capability, because it just makes parallel lines on the surface, in fact with a fixed spacing. You can peen metal this way, but the standard ways to do that (with a hammer or with shot peening) are almost certainly better. But now suppose the comb tines are *movable* under precise servo control?

By varying the heights of the comb tines as you move over the surface, you can form it into any heightfield shape within some limits: you can't get inside radii shorter than the wheel radii or, in the other direction, features finer than the tine spacing; and you can only deform the surface within the plastic limits of the material. By making multiple passes over the surface you can achieve deeper deformation, especially in materials without too much work hardening. In relatively hard materials, it may be more useful to servo-control the pressure (i.e., the mechanical impedance) rather than the position of the roller; this should allow dimensional precision in the finished product that is superior to the measurement precision of your servomechanism.

Unsupported metal and paper sheets are among the easiest materials to plastically deform, but in addition to work hardening, they suffer from a limitation: it's hard to deform them into multimodal curves, because if you push down on both sides of an area, the middle goes down with them. There are several possible solutions, including supporting the sheet with a second comb of rollers, with wood, with some other deformable foam, or with an incompressible fluid, as used in hydroforming.

As with deep drawing, work hardening can be handled by normalizing or annealing the metal between stages of the process. In the particular case of sheet aluminum, which can anneal almost instantly, this can be achieved in a continuous-flow process with an air impingement oven like those used for cooking pizzas, with the air at a carefully controlled temperature a safe distance below the aluminum's melting point.

Extremely plastic materials like wet clay or wet cement might be better formed with an elastic spatula running across the ends of the

comb tines rather than wheels or rollers. A similar approach might work for cutting metals or wood using an elastically-deformed cutting tool.

In all of these cases you will need some degree of FEM material simulation in the toolpath planning and perhaps even the real-time feedback system. I suspect that this has been the major reason for the great popularity of metal-cutting processes such as milling and lathe turning: although the resulting product is weaker than a forged product, the necessary planning and control is simple enough to be done by the humans' brains and massive, stiff metal frames.

Topics

• Materials (p. 3560) (112 notes)
• Manufacturing (p. 3558) (50 notes)
• Digital fabrication (p. 3411) (42 notes)

Pulley generator

Kragen Javier Sitaker, 2016-09-05 (2 minutes)

Could you use a low-friction pulley system to directly drive a tiny micro-DC motor as a micro-generator from a hanging weight? A typical US$3 micro-DC motor can handle about 1–2W of power and produce .49 mN·m of torque, which at a wheel radius of 5mm, is 9.8mN, the Earth weight of 10g. If your weight is, say, 10kg, you need a 1000:1 pulley system; if it is raised through 2m of distance, a simple block-and-tackle would mean you need 500 pulleys at the top, 500 below, and 1000 iterations of thread running between them, for a total of 2000 meters of fine thread. (The thread would ideally be impractically fine: 28 microns of most materials would be sufficient, for a total cross-sectional area of 2.4 mm², and much less of high-strength materials.)

To avoid this massive number of parts and threads, it's probably better to use a multi-stage system: five sequential 4:1 reductions, for example, using gears or pulleys or whatever. A regular 40/3 cotton thread at 4 grams per denier should be able to support 2.7 kg, so maybe 4 to 16 threads running between 4 to 16 pulleys, followed by a 64:1 to 256:1 reduction gearbox or belt drive. (A 32:1 belt-drive reduction could perhaps be done in a single stage, reducing losses, with a 32-cm wheel and a 1-cm wheel; two 16:1 stages would provide 256:1.)

Topics

- Pricing (p. 3646) (89 notes)
- Energy (p. 3438) (63 notes)
- Mechanical things (p. 3569) (45 notes)

Weregild

Kragen Javier Sitaker, 2019-03-24 (3 minutes)

As a measure aimed at reduction of wars, we should perhaps accept weregild again. The traditional Scandinavian weregild rate was 200 solidi for a churl, the least noble category of person for which weregild was due, which works out to 900 g of gold, about US$40k at today's gold prices. I think the current Saudi diyya is a hundred thousand riyals for the noblest category of victim (Muslim men), which is a bit less than US$30k (US$26.7k as of 2015). These prices are a bit lower than modern wrongful-death awards in US courts, which I think are commonly in the hundreds of thousands of dollars, sometimes up into the millions. Presumably the Saudi prices do not apply to members of the royal family.

In Scandinavian times somewhat larger amounts were due for nobler victims: 600 solidi for a duke or archbishop, 300 solidi for a low-ranking cleric (or 400 if they were reading mass at the time), or at another time 1200 for a nobleman, 15000 for an archbishop, or 30000 for the king. The Mercians paid a lower weregild of 110 solidi for Welshmen, or less if they owned no property.

For modern times, probably the prices measured in gold should be higher, partly since we live in a society that is much wealthier, but also because we aspire to egalitarianism; we don't consider women to be worth half a man, as Saudi Arabia does and Iran used to, or Christians to be worth half a Muslim, or dukes to be worth three commoners. Perhaps a reasonable number would be 64 bitcoin, which is currently about US$240k, but will fluctuate with the value of bitcoin. Bitcoin is still very short-term unstable but will probably be a stabler measure of value over the next century or two than the dollar or gold.

Traditionally, wergild (and diyya, wrongful-death awards, and similar concepts) are payable to family members of the victim. Modern sensibilities would demand this to be under the voluntary control of the victim: the payment should be made to the heirs of the victim whoever they be, whether family members or not, as surely it would pile injustice upon injustice to, for example, award the wergild for a transgender hate crime victim to their disowned transphobic parents.

Topics

Nonlinear bounded leaky integrator

Kragen Javier Sitaker, 2019-09-11 (8 minutes)

An integrator or prefix sum is a linear low-pass filter with no rolloff: 6 dB per octave all the way down to dc, where it diverges. Sometimes it's desirable to have a similar kind of filtering action with BIBO stability, and there are a couple of standard linear ways to do that. A nonlinear approach just occurred to me, which is what this note is about.

The integrator and its standard variants

The discrete integrator is the linear filter

$y_i = y_{i-1} + x_i$

also known as the prefix sum, plus-scan, summed-area table, and other terms; it's the discrete analogue to the antiderivative operator. It amplifies frequencies by a linear gain factor proportional to their wavelength, which means that it's not BIBO stable: a bounded input sequence with any dc bias will eventually produce an arbitrarily large magnitude of output. It's marginally stable — if you stop stimulating it with a dc bias, its output won't keep growing. Its impulse response is the Heaviside step function.

It's extremely inexpensive to compute.

If you're doing this on a computer, sometimes this instability can be bad. In C, it can be undefined behavior. In integer arithmetic, it can overflow from positive to negative values or vice versa, which is a problem under some circumstances. In floating point, it results in a gradual loss of precision which eventually becomes total, although, for a signal with 16 bits of dynamic range being represented in a 64-bit IEEE-754 float, loss of precision doesn't begin until you have processed at least 2^{37} samples. In arbitrary-precision arithmetic, which is generally not used for signal processing, it starts to use more memory and become slower.

If you're using the integrator as a model of some physical system, such as a capacitor charging from an operational transconductance amplifier, you have a potentially more serious problem: the system almost certainly has some kind of physical bounds on its response, and if your linear model has unbounded behavior, that means its approximation of the physical system is going to be unboundedly awful under some circumstances.

The really long boxcar and its variants

Consider composing the integrator above with the following sparse FIR comb filter, producing a filter that is overall FIR and thus BIBO:

$y_i = x_i - x_{i-20480}$

This boxcar has the same output as the integrator on signals of less than 20480 samples and on frequencies whose wavelength is much shorter than 20480 samples, but for lower frequencies, its gain has an asymptote of 20480, although the gain isn't very well behaved; it has sharp nulls at $2\pi = \omega$ 20480 samples and its harmonics.

You could soften this a bit to something like

$y_i = 3x_i - x_{i-33137} - x_{i-20480} - x_{i-12657}$

so that you don't have any really sharp nulls like that, just some 1.8-dB attenuations.

However, you still have striking time-domain artifacts in the form of echoes: one at 33,137 samples, one at 20,480 samples, and one at 12,657 samples.

If you pass a signal through a filter with a Gaussian time-domain response, followed by an integrator, you get a unstable filter with a sigmoid impulse response, a sort of softened Heaviside step function. You can approximate this closely with four integrators and three combs like the comb above. If you delay the output of this filter, scale it down to unit magnitude, and subtract it from an integrator, you can similarly tame the integrator and get it to be FIR and thus BIBO stable, without much echo except at low frequencies and, I think, without any sharp nulls; the impulse response of the combination is a pulse with a sharp beginning, a flat top, and a slow sigmoid decline to zero. It costs one multiply, five adds, and four subtracts per sample.

The stable approximation of an integrator provided by these LTI hacks may be adequate for many purposes.

Exponential leakage

A simpler way to make the integrator BIBO stable without altering its high-frequency response and LTI nature is to add a little bit of exponential decay:

$y_i = k y_{i-1} + x_i$

Here k is a decay factor between 0 and 1, say something like 0.99 or 0.999, analogous to a bleeder resistor across an accumulating capacitor. The impulse response of this filter is a pulse with a sharp onset followed by an exponential decay back to zero with a time constant $\tau = -1/(f_s \ln k)$.

This has no echoes or sharp nulls, but it's not FIR like the boxcars.

Nonlinear leakage through saturation

Suppose we harshly clip our integrator output as if it were the output of an ADC:

$y_i = -k \vee y_{i-1} + x_i \wedge +k$

(Here $a \vee b$ is a if $a > b$, b otherwise, and $a \wedge b = -(-a \vee -b)$.)

This guarantees that it's BIBO stable because its output is bounded to $[-k, +k]$, no matter what the input is. (This approach is commonly used to limit integrator windup in PID controllers.) It gives up linearity, and in the process creates all kinds of potential for interaction between frequencies.

An interesting thing about this approach is that if the integrator is floating around near its limit when some high-frequency oscillation starts, say with amplitude 0.1, that would force it beyond the limit, the first quarter-cycle of that oscillation gets clipped; but thereafter the whole oscillation proceeds without incident, having added enough of a negative step function (a component at dc!) to make room for the rest of its waveform below the saturation level.

Another interesting thing about it is that there are a lot of natural phenomena that behave to some degree like this, including saturation

in transformers and ionic polarization in dielectrics (see Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033)); the optical Kerr effect can manifest such effects at near-exahertz timescales.

Softer saturation

Suppose we'd like to get some of this effect, but with gentler nonlinearity; we'd like to smoothly tilt the playing field for x so that it can move y back toward zero a little more easily than it can move y further away from zero. This way, maybe we can get the BIBO stability of the hard saturation thresholds, the same no-cutoff-frequency low-pass filtering action of the pure integrator, and minimal waveform distortion, except where these three conflict.

Maybe something like

$$y_i = (1 - kx_i^2)y_{i-1} + x_i$$

would do. But I feel like this is maybe *too* nonlinear, since it fails at BIBO. Also requiring three multiplications per sample seems like a lot, since the standard LTI exponentially-leaky integrator only requires one.

$x_i y_{i-1}$ is negative when x_i is trying to decrease the magnitude of y_i, and positive when it's trying to increase them. So an alternative, simpler approach might be to use this factor to adjust the gain on x_i smoothly:

$$y_i = y_{i-1} + (1 - kx_i y_{i-1})x_i$$

This still requires three multiplications per sample. The parameter k, maybe in the range 0.0001 to 0.1, sets the maximum amplitude of the output, which also depends on the frequency (relative to the sampling rate) and the other existing frequencies.

This is imperfect — in particular, the gain goes negative if $kx_i y_{i-1} > 1$ — but it seems to be giving reasonable results on some simple test signals. It can produce extreme harmonic distortion with undesirable values of k.

```
def nlf(x, k=.025):
    y = x.copy()
    for i in range(1, len(y)):
        y[i] = y[i-1] + (1 - k*x[i]*y[i-1])*x[i]
    return y
```

Topics

• Programming (p. 3658) (286 notes)
• Algorithms (p. 3310) (123 notes)
• Digital signal processing (DSP) (p. 3419) (60 notes)
• Prefix sums (p. 3645) (18 notes)

Bitstream dsp

Kragen Javier Sitaker, 2015-09-03 (updated 2019-06-23) (3 minutes)

I've been wondering if you can do DSP operations usefully with bitwise operations on infinite streams of bits, perhaps representing a signal in PDM, with a pipelined graph of binary-signal combinators. This is tempting both because you can get 64-way or 128-way parallelism on modern CPUs and because you can perhaps conserve area in FPGA implementations.

For example, to detect a frequency, you could generate two 1-bit-deep square waves in quadrature at the frequency, XOR them with the input bits, and then run a population count of the bits. This has the disadvantage that you're getting contaminated with the correlation with odd harmonics of the square-wave probe signals; I think you can maybe reduce this problem with upconversion?

Upconversion, of course, is also XOR.

Population count can be done with streaming full-adders on the bitstreams, after ANDing them with wavelength-2, wavelength-4, wavelength-8, etc., square waves and their complements, then delaying the output of the complemented version by 1, 2, 4, etc., bit times, before adding it back to its counterpart. This crudely converts the bitstream into PCM, but the full-adders can remain oblivious to the PCM sample boundaries. Thresholding the PCM can perhaps use a single bit test in each sample (generated by AND with a low-duty-cycle wave of the same frequency). AND and OR may be adequate approximations for min and max.

For other practical operations, a perfect-unshuffle operation would be useful, converting one bitstream into two bitstreams at half the speed.

A FIR filter might be a fixed-length word that you XOR against the bitstream at every bit offset, emitting perhaps the majority-rule of the XOR output bits.

2019 update: it turns out that there is some substantial research on a bitstream approach, but it uses very different primitives than I was thinking. The fundamental operations are NOT (1 - X), AND (X·Y), selecting bits at random from either of two bitstreams (½(X+Y)), and of course delay with a D flip-flop. The random-selection idea is very clever! My thinking was stuck in the deterministic paradigm, which is perhaps an unnecessary constraint when we're talking about bitstreams that by necessity incorporate a lot of error.

(I wish I could remember the names of the bitstream DSP papers or researchers.)

I've also read more about oversampling 1-bit ADCs and DACs since I wrote the above (mostly in Horowitz & Hill) and what I read leads me to believe that this approach could be more effective than you might think, since a third-order ΔΣ converter can hit an ENOB of 16 at only a 64× oversampling ratio, only 4× worse efficiency than PCM. I don't know that the noise shaping of these incredible devices will survive the kinds of operations described here.

If not, a sort of ΔΣ converter might itself be in some sense a safer way to do these computations. For example, if you want to attenuate

the signal represented by a ΔΣ bitstream by 9dB, you could very reasonably build an all-digital feedback system using a couple of up/down counters which attempts to maintain the input counter at 8× the value of the output counter.

Topics

- Performance (p. 3621) (149 notes)
- Electronics (p. 3430) (138 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Fpga

Median filtering

Kragen Javier Sitaker, 2019-01-17 (11 minutes)

I want to do some median-filtering image processing. First I want to do it in a janky way that will work for 2×2, 3×3, 4×4, and 5×5 square filter kernels, to see if the results I get are reasonable. Then I want to see if there's a better way to do it.

Median filtering by brute-force sorting

The janky way is to copy the pixels in the window into a buffer, sort the buffer, and then take the middle pixel, or like an average of the two middle pixels. These buffers are small enough that insertion sort is probably the way to go. Insertion-sorting a 5×5 square requires 300 comparisons, though, which is enough to make it slow.

A weighted-median filter could be an interesting way to cut down on resonance artifacts: count the pixels toward the center as having more weight than they really do. One way to do this is, after sorting, you calculate the CDF of the weights in the sorted order, then take the pixel with the CDF in the center of the range.

Median filtering by quickselect

Quickselect should be about 6× as fast, but it will still do 50 comparisons, I think.

Subjectively, quickselect on my laptop becomes unbearably slow once the window is bigger than about 12×12, at which point it's doing 548 comparisons per window; I think that with the usual partition algorithm it would be doing 288, but I'm using Lomuto's algorithm because it's simpler.

Incremental median filtering

Can you reuse some of the work from one window position to the next window position? For example, if you're shifting a 5×5 square (or whatever 5-pixel-tall window) to the right by one, can you replace the original 5 pixels with the new 5 pixels and then move them to the right place? On average they'll move 12 positions, so this involves about 60 comparisons — probably still slower than quickselect.

Incremental quickselect

How about incrementalizing quickselect itself? Once you've found the median, the buffer is partitioned between less-than-median and more-than-median items. If you replace 5 of them, they may need to move to the other side of the buffer, pushing the median over a bit — but you don't a priori know which element from that expanded side is the new median. You have to run quickselect on it again, which is only a speedup of 2, giving you about 50 comparisons.

Median of medians

What if you calculate the median of each pixel column and then take the median of their medians? This turns out to also save you only about two thirds of the work, down to half in larger cases. Here are how the partitioned items relate to the median of medians, having sorted the columns by their medians

```
? ? + + +
? ? + + +
- - = + +
- - - ? ?
- - - ? ?
```

That is, one of them is the MOM itself, two medians to the left are known to be less, and the two to the right are known to be greater. Transitively, the elements earlier in the left two columns are also known to be less than the MOM, because they're less than their own column median, which is less than the MOM. Analogous remarks apply to the right. But we don't know anything about the other 8 elements; any of them could be the overall median.

I'm not sure how to get from this to the actual median in an efficient way, although this might be a good approximation to the actual median.

Merging for incremental sorting

What if we sort the new column and then merge it into the sorted order of the non-deleted pixels in a single pass, 1950s business DP style? This seems like it is going to be a win; sorting the new column with insertion sort takes 10 comparisons, and then merging it into the sorted order takes I think 25, for a total of 35 or so.

Heaps for incremental quantiles

What if we maintain a max-heap of the elements in the window that are less than the median and a min-heap of those that are greater? We can replace the old pixels with new pixels and then sift them up or down as appropriate, possibly shifting elements through the median to keep the heaps the same size. Consider the 5×5 case; with 12 elements in each heap, we will typically sift about 2 steps, for about 10 steps per window shift.

In the 12×12 case, quickselect needs about 548 comparisons in my current implementation. The heap approach would have 72-element heaps, which would be mostly 6 levels deep, so we would typically sift each element by about 3 steps, for 36 comparisons.

Incremental overhead

The data structure for such an incremental approach probably needs to be a little more complicated than a simple array of pixel values, because we need to be able to efficiently find the 5 samples to replace when we shift. We need one map from (x, y) pairs to buffer positions so that we can find the buffer positions to replace and a second map from buffer positions to (x, y) pairs so that we can access the actual pixel value or swap the positions of two pixels. When you swap two positions, you have to update both arrays. I don't know if there's a simpler way than this. This will probably impose enough update overhead that the crossover from non-incremental quickselect will come at a larger size than 5×5, since non-incremental quickselect can work on mere pixel values.

Histograms

Suppose you have only, say, 256 different possible color values in the image. Then you could represent the distribution of those values in a window with a histogram, an array of 256 numbers counting the

number of pixels that has each value. The prefix sum of the histogram, the empirical CDF, would cross half the size of the window precisely at the median.

A desirable property of the histogram is that it can be updated incrementally quite efficiently: to shift a pixel out of the window, decrement its value in the histogram, and to shift one in, you increment its value in the histogram.

Instead of maintaining all 256 values, you could maintain a reduced histogram with, say, 16 buckets; and instead of maintaining just a count in each bucket, you could maintain an array of the colors that fell into that bucket. The bucket at which the empirical CDF crossed the median would tell you approximately the median, but to find it precisely, you would need to compute some arbitrary order statistic over the pixels in the bucket, which could be done with quickselect. Removing a pixel from the bucket now requires a linear search through the bucket for its value, but the number of pixels in each bucket is small.

Perhaps, in the form described above, the ideal number of buckets is roughly the square root of the window size; for example, for a 12×12 window, perhaps 8 to 16 buckets would be best. This is because for each new output pixel we must recalculate the empirical CDF, which takes time linear in the number of buckets, and then we must calculate an order statistic inside some bucket, which (under dubious but probably approximately correct assumptions) takes time inversely proportional to the number of buckets.

An incremental prefix sum data structure would change the cost function to favor a larger number of buckets, and the recursive binary-tree form in particular can update the prefix sum in logarithmic time; in its sparse form, it reduces to a binary trie over pixel values in which each trie node knows the total number of descendants it represents.

(As an alternative to going all the way to the trie structure, you could use some structure other than a linear list inside each bucket. But nothing occurs to me that would be a real speedup.)

The bucketing and trie techniques allow the histogram approach to scale to large color spaces, such as 12-bit grayscale or 24-bit color.

The complexity cost of this technique is nontrivial. My quickselect implementation is 18 lines of C and worked the first time; my prototype trie implementation, which crashes, is 157 lines, although to be fair it was probably a premature optimization to use node pools before I had it working at all.

Rank order filters

An interesting thing to note about median filtering is that it can be generalized to a family of quantile filters; you could use any arbitrary percentile rather than just the 50th percentile. The result is a filter that tends to suppress local noise and leaves uniform-color areas of the image unchanged, but tends to contract or expand brighter areas according to whether the percentile is less or greater than 50. In the extreme cases of 0 and 100, where we take the minimum or maximum of the neighborhood, the filters are the familiar erosion and dilation operations from mathematical morphology; the other percentiles provide a more or less smooth variation between erosion and dilation, which are the maximally contractive and expansive

filters of the family, respectively, and the most sensitive to noise, while the median filter itself is neutral between contraction and expansion and the least sensitive to noise. With a 25-element kernel with equal weighting, there are 25 quantile filters in the family, three of which are erosion, median, and dilation.

These are commonly known as "rank order filters" or sometimes "order-statistic filters".

The data structures discussed above for median filtering are all applicable to these more general filters, but their efficiency properties vary. The median-of-medians mentioned above provides a better estimate in any case other than the median case, for example:

```
? + + + +
? + + + +
? + + + +
- = + + +
- - ? ? ?
```

And in the extreme cases of erosion and dilation, it has no unknown pixels, amounting to the standard decomposition of a rectangular erosion or dilation kernel into horizontal and vertical kernels.

The trie algorithm, in particular, can effortlessly handle arbitrary rank-order filters.

Topics

Hot wire saw

Kragen Javier Sitaker, 2015-12-28 (updated 2019-06-02) (10 minutes)

An idea based on watching Grant Thompson's description and demo video of his "Styro-Slicer" hobby hot wire cutter.

As an alternative to laser-cutting wood or MDF, you could cut it with a hot wire, like the one in a hot-wire styrofoam cutter. Some hot-wire styrofoam cutters are already capable of cutting soft woods with some difficulty.

The conductive wire bandsaw

I am envisioning a sort of bandsaw in which an endless loop of thin wire is run through the kerf, heating up on its way into the kerf and then being quenched as it plunges into the material, dumping its heat into the wood. Two conductive rollers, one above and one below, feed electricity through it; the distance between the rollers is so much shorter than the distance around the other part of the loop that the other part of the wire is not dangerously heated.

To be concrete, let's consider 80-micron-thick (40 AWG is 79 microns) nichrome wire cutting 3mm MDF with 50 watts of power, leaving, say, a 100-micron-wide kerf, with the two electrical-feed rollers at a distance of 45mm between them and an additional 900mm in the loop. Nichrome's resistivity is about a μΩm and apparently 800 mA is enough to keep an 80-micron wire radiating at 1100°, which is plenty hot enough. In this case the resistance of the 45mm should be about 1 μΩm * 45mm/((80μm/2)²π) or about 9 ohms, so you need about 2400 mA to get to 50 watts:

```
You have: micro ohm meter * 45 mm / ((80 microns/2)^2 pi)
You want: ohms
        * 8.9524655
        / 0.11170107
You have: sqrt(50 watts / (micro ohm meter * 45 mm / ((80 microns/2)^2 pi)))
You want:
        Definition: 2.3632718 A
```

This means your power supply needs about 22 volts, which is eminently feasible.

The resistance of the 900mm part is 20 times greater, or 180 ohms, so it will be carrying 120 mA and dissipating about two or three watts --- enough to require management but not an overwhelming problem.

This machine seems nearly feasible, but a problem with this is that the exposed 45mm of wire is going to be not only radiating like a motherfucker, perhaps losing most of its energy into the void, but also possibly getting rather hotter than the 1400° that it can in fact withstand before melting. And don't forget that the MDF needs to have air blowing on it the whole time to keep it from catching on fire, which is also going to cool off the wire, although maybe this problem can be limited by careful management of the air blowing, so that it's only blowing in a very thin layer.

So consider a revision: 127-micron-diameter (36-AWG) wire

cutting a 150-micron-wide kerf (still at only 50 watts) with only a 12-mm gap between the conductive rollers. This drops the resistance per mm by 2.5 and the distance by 4x, getting us down to about 930mΩ, which means that we now need 7 amps running through the wire, powered by about 7 volts, and a power-supply output impedance of below an ohm or so to keep efficiencies reasonable. That's still hot enough to melt the wire if there's no thermal load on it, but I feel that it's possible to keep it running through the rollers fast enough to keep it passively safe, even without rapid-response electronic control to keep it from burning. And it's mechanically strong enough that it might survive encounters with things that unexpectedly fail to burn on contact with it.

(180-micron (33-AWG) nichrome wire is commonly used for styrofoam-cutting machines like the popular Argentine brand "Segelin". Thicker gauges are common; thinner gauges are somewhat exotic. One Aldo Vignolo sells them on Mercado Libre here in Buenos Aires.)

What do we do about splicing the wire? One possibility is to simply not worry about it, instead merely reversing direction when the time comes, maybe turning off power to let the wire cool in between.

How much thermal energy can be stored in the gap before the wire melts? Nichrome has a specific heat of 450 J/kg/K and a density of 8.4 kg/liter, which works out to 3.8 kJ/liter/K. At 127 microns, that's 48 microjoules per millimeter per kelvin, or 67 millijoules per millimeter, or about 800 millijoules in all, to melt the wire in the gap. The situation is slightly worse because nichrome's resistance increases slightly with temperature, so it can experience a kind of thermal runaway where hotspots heat up faster, and also slightly better because the hottest spots are also radiating away energy most rapidly. Disregarding these, this means that keeping it passively safe at 50 watts involves feeding a gap's worth of wire onto a quench roller about every 16 milliseconds, a speed of about 750 mm per second. But that's just to barely keep the wire from melting.

I'm not sure that shortening the gap was a big improvement. It does lower the amount lost in the other part of the loop, but it also reduces the heat capacity of the wire in the gap, thus requiring faster response if we decide to use active feedback for safety, and so it doesn't actually affect the necessary passive-safety wire speed. And, because it lowers the resistance, it requires a higher-quality power supply and heavier wiring. And, because it raises the required current, it also increases the possibility of melting the wire --- resilience is likely of more value than efficiency here.

A third revision, where we increase the gap size to the accursed 25mm, leave the wire thickness at 127 microns, and drop the power to 40 watts, requires less current, only 4.5 amps:

```
You have: sqrt(40 watts / (micro ohm meter * 25 mm / ((127 microns/2)^2 pi)))
You want:
        Definition: 4.5020328 A
```

This is low enough that it is *almost* passively safe even for copper, which melts at 1084°, even without the quench rollers. The resistance in the gap is up to 2Ω, which means you need 9 volts, a very

convenient number indeed. You no longer need quite such big hairy power cables or a high-quality power supply; Duracell's spec for their MN1604 nine-volt battery lists a 1.7Ω impedance, so even that would be almost good enough; a dozen or so in parallel would do, though obviously a line-powered power supply would be more cost-effective. The gap can now hold 1.7 J of thermal energy before melting:

```
You have: 450 J/kg/K * 8.4 kg/liter * ((127 microns/2)^2 pi) * 25 mm * 1400 K
You want: J
        * 1.675935
        / 0.59668186
```

Lowering the power to 40 watts means that it takes 42ms to reach melting temperature in the absence of any heat-loss mechanism, such as radiation, conduction into the workpiece, or the quenching roller. For the quenching roller alone to be sufficient, the wire needs to move at 600 mm/s:

```
You have: 25 mm * 40 watts / (450 J/kg/K * 8.4 kg/liter * ((127 microns/2)^2 pi) o
o* 25 mm * 1400 K)
You want:
        Definition: 0.59668186 m / s
```

The 150-micron kerf I expect from this is not quite as good as the 100-micron kerf I see from laser cutters. Contrast this with the Olson No. 2 spiral scroll saw blade, which nominally gives you a kerf of .035 inches in US folk units (890 microns), losing six times as much material and thus with six times as much opportunity for imprecision.

I don't know if it will work as well as a 40-watt laser cutter at cutting wood, even if you blow air down the wire. I think the laser cutter is usually heating up a tiny point rather than a whole contact surface, so it might heat the wood or MDF to a higher temperature. But maybe this will instead work better, since it has some mechanical sliding action and won't be impeded by smoke.

CNC foam cutters commonly use inconel rather than nichrome.

US patent 5,429,163 is relevant; it claims 600° to 800° is sufficient for cutting wood, and suggests using either a reciprocating or an endless wire. It says the wood itself need only heat up to 240° to 270°, and that irregularities in the wood pretty much require closed-loop temperature control, and claims 10 to 21 mm/sec of cutting speed. Unfortunately it doesn't go into any detail about power usage, preheat gap size, wire speed, or wire diameter.

Induction heating

As an alternative to heating the wire by running electricity through it lengthwise, you could run it through a high-frequency induction heater and thus induce eddy currents in it, particularly if it's ferromagnetic. Rudnev et al.'s *Handbook of Induction Heating, Second Edition* tells us on p. 38, in §2.2.4, that induction heating of wire is commonplace, using frequencies from 10 to 800 kHz and wire speeds through the inductor around 2 m/s. They cite a particular system they built for Radyne that heat-treats 1060 carbon-steel spring wire in

the 4–8-mm diameter range with a 400 kW, 10 kHz inverter for heating up to its Curie temperature, then a 420 kW, 200 kHz for heating above its Curie temperature; but they don't give a wire speed.

Topics

Fast mathematical optimization with affine arithmetic

Kragen Javier Sitaker, 2019-09-15 (5 minutes)

Suppose that, like most days, we're trying to find a local minimum of a univariate function $f(x)$ we can automatically differentiate with a continuous derivative. By using affine arithmetic, we should be able to get both quadratic convergence to the minimum, under Newton–Raphson-like conditions, and a guarantee of global optimality.

Newtonish–Raphsonish root finding with affine arithmetic

Suppose that the derivative is continuous (though not necessarily Lipschitz) and our automatic differentiation procedure supports reduced affine arithmetic, so that we can evaluate the function's derivative $f'(x)$ at "points" that are really affine forms describing some interval $[x_0, x_1]$, such as an interval with a zero in it, and get back an affine form that gives a linear approximation $k + a_0x$ and a fairly tight worst-case error a_1 from that approximation $x \in [x_0, x_1] \Rightarrow \exists\varepsilon \in [-1, 1]: f'(x) = k + a_0x + a_1\varepsilon$. (Affine arithmetic is discussed at some length in An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) and Affine arithmetic optimization (p. 2801).)

This allows us to bound the zero of the derivative, and thus the critical point of $f(x)$, to a generally much smaller interval: x must be between $-(k - a_1)/a_0$ and $-(k + a_1)/a_0$. In fact, *every* zero of the derivative within the original interval is guaranteed to be within that new interval, so in cases where the new interval is larger or fails to be sufficiently smaller than the original, we must be prepared instead to subdivide the interval and recurse on each subinterval.

Rate of convergence and global optimality

I was thinking about trying to do a rigorous proof here but instead I'm just going to handwave because it's almost 4 AM. Generally the (vertical) error bounds affine arithmetic gives you are proportional to the second derivative of the function you're calculating (which is itself a derivative in the above), unrelated to its first derivative, and inversely proportional to the square of the interval width. This is pretty much the same situation as Newton–Raphson iteration or the method of secants, so you should expect it to converge quadratically under the same conditions that they do.

If you can figure out how to make an initial interval that is big enough to contain all the local optima, then this approach is guaranteed to find all the local optima and in fact all the critical points; you can use branch-and-bound to avoid recursing too deeply in areas where the local optima are not as good as the worst case in other areas where you are also searching, so even functions with a large number of critical points may be tractable.

Indirect multidimensional optimization, and root-finding in general

This can, of course, be used to find zeroes of functions that aren't derivatives of anything of special interest, and I think that's where I got the idea — from people using this approach for raytracing of implicit surfaces, specifically Gamito and Maddock's paper from 2004 or 2005, "Ray Casting Implicit Fractal Surfaces with Reduced Affine Arithmetic", §4.3, p. 6, fig. 2. But it seems like an approach that could be extremely fruitful for mathematical optimization. In Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) I suggested cutting a multidimensional function along a gradient line to get a one-dimensional function that's easier to optimize, an approach I haven't tried yet, but it seems like this approach would work for it.

Direct multidimensional optimization

Alternatively, it might be feasible to use this approach more directly for multidimensional optimization. There's a paper out a few years back about a correct algorithm using interval arithmetic for multidimensional optimization, using a branch-and-bound approach, but as I understand it, it has the problem that it is very slow on high-dimensional spaces. Suppose you look for a zero of some gradient $\vec{g}$ as $\Sigma_i g_i^2$ or $\Sigma_i |g_i|$ using a variant of the method above, using reduced affine arithmetic to compute bounds on this (squared) gradient magnitude or other gradient norm over some multidimensional box; I think this should take time merely proportional to the dimensionality.

The magnitudes of the coefficients in the resulting affine form should tell you how sensitive the gradient is to your location within the box along each dimension; to some extent you can trade off shrinking the box along one dimension against shrinking it along another dimension, but it's hard to know which dimension is optimal to shrink most in.

Maybe if we automatically differentiate our reduced-affine-arithmetic program, we can get the gradient of the error bars on the (original function's) gradient norm, with respect to the bounds of the box we're evaluating the gradient over. This would tell us which dimensions of the box are most important to shrink in order to reduce the uncertainty in the gradient.

Topics

- Programming (p. 3658) (286 notes)
- Performance (p. 3621) (149 notes)
- Algorithms (p. 3310) (123 notes)
- Math (p. 3564) (78 notes)
- Mathematical optimization (p. 3611) (29 notes)
- Interval and affine arithmetic (p. 3528) (24 notes)
- Newton–Raphson iteration ("Newton's method") (p. 3595) (6 notes)
- Automatic differentiation (p. 3336) (6 notes)

Techniques for, e.g., avoiding indexed-offset addressing on the 8080

Kragen Javier Sitaker, 2019-07-20 (updated 2019-07-24)
(27 minutes)

Reading the 8080 instruction set and watching David Givens's recorded livestream of writing a text editor for CP/Mish, I'm struck by the nonexistence of indexed-offset addressing modes, and the relatively large cost of emulating them; so I was thinking about ways to avoid this cost.

8080 indexed-offset memory access

If your program wants to access a one-byte variable the C compiler has allocated at offset 4 from the stack pointer in its stack frame, it needs to do something like the following:

```
LXI H, 4     ; HL := 4; 10 cycles; "X" means 16-bit
DAD SP       ; HL += SP; 10 cycles; "X" is implicit
MOV B, M     ; B := M[HL]; 7 cycles
```

This takes 27 clock cycles, which is 13.5 µs at the 8080's 2MHz maximum clock speed. In some cases, the MOV at the end might be replaced with something like `INR M` to increment the variable (10 cycles, 5 more than incrementing a register) or `ADD M` to add it to the accumulator without loading it into a register first (7 cycles, 3 more than adding a register). So you could reasonably argue that the cost is something like 23 cycles rather than 27.

Just as fast as `MOV Y, M` is `LDAX` or `STAX`, which load and store the 8-bit A register from or to the address in BC or DE in only 7 cycles; unfortunately, you can't store addition results in BC or DE.

(I haven't looked at the actual code generated by ACK or BDS C, and I'm not that familiar with 8080 assembly language, so I might have gotten something wrong here.)

The code is pretty much the same if you're indexing a record† rather than a stack frame, just that the base address doesn't come from SP:

```
LXI H, 4
DAD B        ; or D
MOV B, M
```

And it's almost the same if you're indexing into an array (without bounds-checking), except that you might need to multiply the index by the array-item size; for fetching the first 8 bits of a 16-bit item, for example:

```
LXI H, 2834H ; the array base address
DAD B
DAD B        ; an extra 10 cycles
```

```
MOV B, M
```

By contrast, if the variable is at a fixed location in memory, you can avoid the DAD bit, cutting the cost from 27 or 23 cycles to 17 or 13, depending on how you figure it:

```
LXI H, 2082H ; HL := 0x2082; 10 cycles
MOV B, M     ; B := M[HL]; 7 cycles
```

But wait! You can do better! If you're willing to load it into A rather than some other register, you can instead use the 13-cycle LDA (resp. STA) instruction, which takes up 3 bytes:

```
LDA 300H     ; 13 cycles
```

If you're doing indexed-offset addressing, sequential reads can be significantly faster, because you can increment or decrement HL in only 5 cycles. Here the initial indexed load takes 27 cycles, but the subsequent sequential load takes "only" 12 more:

```
LXI H, 4     ; 10 cycles
DAD SP       ; 10 cycles
MOV B, M     ; 7 cycles
INX H        ; HL++; 5 cycles
MOV E, M     ; E := M[HL]; 7 cycles
```

Chasing pointers involves loading 16-bit values; the LHLD and SHLD instructions (16 cycles each) load and store the value of HL at fixed addresses. Loading it from an address pointed to by a register is more involved; you can load it one byte at a time, for a total of 23 cycles (if you want the result to be in HL and not, say, DE). For an apples-to-apples comparison with the 8-bit situation, it's 43 cycles if we first point HL at an offset into the stack frame:

```
LXI H, 4     ; 10 cycles
DAD SP       ; 10 cycles

MOV E, M     ; E := M[HL]; 7 cycles
INX H        ; HL++; 5 cycles
MOV D, M     ; D := M[HL]; 7 cycles
XCHG         ; HL, DE := DE, HL; 4 cycles
```

Or you can use SP as a pointer and then POP H, taking 10 cycles. Pointing the stack pointer at a random address with SPHL only takes 5 cycles, but that probably requires you to save the stack pointer ahead of time so you can restore it later. Unfortunately, I think the only way to access the old value of SP is with DAD SP, so the whole sequence is gnarly and takes 52 cycles:

```
XCHG         ; HL, DE := DE, HL; 4 cycles (save old HL)
```

```
    LXI H, 0      ; HL := 0; 10 cycles
    DAD SP        ; HL += SP; 10 cycles
    XCHG          ; HL, DE := DE, HL; 4 cycles (save old SP)

    SPHL          ; SP := HL; 5 cycles
    POP H         ; HL := M[SP]; SP += 2; 10 cycles

    XCHG          ; HL, DE := DE, HL; 4 cycles (restore old SP)
    SPHL          ; SP := HL; 5 cycles
```

But the `SPHL, POP H` sequence in the middle is only 15 cycles, so if you need to follow a chain of pointers, that's probably a faster way to do it. However, in the middle of this mess, you don't have access to the old stack pointer, which would further complicate access to local variables allocated in the stack frame.

The 8080 implicitly uses the stack to handle interrupts, so under most circumstances, you'd need to disable interrupts to use the above trick; otherwise an interrupt at the wrong moment will overwrite stuff before the pointers you're trying to chase.

Finally, you could use self-modifying code, which takes 32 cycles — slower and more code than just doing byte-at-a-time access, but doesn't trash DE:

```
    SHLD $+4      ; store HL into the address field of next insn; 16 cycles
    LHLD 0        ; load HL from address to be inserted; 16 cycles
```

This has the potential advantage that the two instructions can be at separated places, and in particular you might be able to set the address once and load from it many times.

For some special cases, there were faster ways to access data on the stack, the most obvious being simply by popping it, but there were others. For example, Alan Miller's *8080/Z80 Assembly Language: Techniques for Improved Programming* suggests that if you have passed an argument to a subroutine on the stack:

```
    PUSH B        ; i.e., BC
    CALL FOO      ; i.e., PUSH PC and then JMP
```

Then that subroutine best can get the argument (inconveniently hidden beneath its return address) into a register using `XTHL` (p. 38):

```
FOO POP H         ; i.e., HL; 10 cycles
    XTHL          ; M[SP], HL := HL, M[SP]; 18 cycles
```

This also works for return values.

In summary, on the 8080, it's dramatically faster to load data from memory at statically allocated addresses than at addresses on the stack or in records or arrays:

address	bits	cycles to read into register	bytes of code
static	8	13	3
in HL	8	7	1
offset from SP, BC, or DE	8	27	5
static	16	16	3

in HL	16	23	4
offset from SP, BC or DE	16	43	8

The first CP/M machines used the 8080, but the backwards-compatible Z80 was the CPU most CP/M machines used. It had index registers IX and IY which apparently ameliorated these problems noticeably, but did not remove them entirely. I haven't tried these exercises on the Z80.

† A record is called a "struct" in C-derived languages, a "tuple" in ML-derived languages, and an "object" in Smalltalk-derived languages.

Dynamic scoping with shallow binding in LISP

Many other old computers had similar problems, and I think this is the reason for the conventional wisdom among 1970s LISP implementors that, although lexical scoping was a good idea in theory and simplified the understanding of programs, in practice, the performance cost was too high, relative to then-conventional dynamic scoping with shallow binding, in which the current value of each variable was stored in a fixed memory location, but upon entering and exiting a subroutine that has it as a local variable, its previous value is pushed onto a stack, then restored upon exit.

Similar considerations, plus the then-popular technique of storing subroutine return addresses in the return instruction through self-modifying code rather than using a stack, prompted the omission of recursion from COBOL and older versions of FORTRAN.

`static` variables in C

At one point (about 45' into the livestream, I think), Givens gets a noticeable speedup in redrawing his screen by changing a couple of 16-bit stack-allocated variables (`auto`, the default storage class in C) to the `static` storage class, thus enabling the use of static-address instruction sequences like those above rather than (presumably) the offset-from-SP sequences.

At first, this optimization introduced a bug, since local `static` variables are initialized upon the first entry to the subroutine, while `auto` variables with initializers are initialized upon every entry.

You could imagine an optimizing source-to-source translation that would simply add a `static` to every implicitly `auto` local variable in your program and separate its initialization into a separate statement. This transformation would be sound — it would not break previously correct code — except in the case of recursive functions, or more specifically variables in recursive functions whose values are read after at least one recursive call without being written to again first.

This translation improves things, but it has a few problems. First, you can't declare function parameters `static` in C. Second, it could result in your program using more memory than before, because while previously you only needed enough space, on the stack, for the functions in the single deepest call chain (weighted by activation record size), now you need enough memory for all the activation records to be alive at once, because functions can no longer share memory with other functions that aren't active concurrently. Third, access to variables in recursive functions is still slow.

(How important are recursive functions? They enormously simplify recursive-descent parsing, some computations on trees and

graphs, interpreters, and simplified regular-expression matching, as well as many mathematical computations like Aryabhata's pulverizer algorithm, so they are an important feature to have in the language, although they are hard to use safely. But typically only a small minority of code is inside the recursive loop, and it's not performance-critical code. Many programs entirely lack recursive loops; Givens's qe.c is 956 lines of C which, despite its use of function pointers, is entirely nonrecursive.)

Compile-time stack allocation

We could imagine instead statically allocating the activation records of a nonrecursive program on a sort of stack, at compile-time. You could allocate the activation record of `main` at some address `local_variable_start`, and all other activation records at one more than the greatest address used by any of their callers' activation records. This allocates a single static address to every local variable.

So, for example, given call chains main[6] → a[3] → b[5] → c[2] and a → d[7] → c, where the number in brackets is the number of bytes needed for each activation record, you would allocate main at 0, a at 6, b and d each at 9, and c at 16. b and d overwrite the same memory, but that's okay because they're never running at the same time. When b calls c, two bytes are left unused, but that's also okay, because c isn't getting its local variables' addresses from b; they're compiled into it.

To extend this to recursive programs, we can break each recursive loop, or potential recursive loop, by introducing a "trampoline" function into it at some point; a greedy approach may not be optimal but is likely adequate. The "trampoline" has the job of saving the variables from the functions participating in the recursive loop onto a run-time stack somewhere, then forwarding the call to the next function in the recursive loop, then restoring the saved variables when it returns. There may be some variables that don't need to be saved because their saved values are statically never used after the recursive call. It may be worthwhile as an optimization to simply `memcpy()` a relevant chunk of the compile-time stack rather than enumerating all the necessary variables.

(Incidentally, since it only gets invoked once per recursive loop and knows exactly how much stack space it needs, the trampoline is in a position to efficiently check for stack overflows and report them in a usable fashion, something C runtimes usually fail badly at.)

C, however, has another feature that complicates this: you can take the address of a local variable and pass it to other functions. (In C, you can also store it in data structures; the Pascal family including Oberon (see A review of Wirth's Project Oberon book (p. 431) and IMGUI programming language (p. 103)) instead provides a "var parameter" mechanism analogous to downward funargs, which, however, poses precisely the same potential problem for this mechanism.) It is expected that such an address will remain valid until the function it's in returns, despite possible recursion. If such a variable occurs in a recursive loop, it needs to be immediately allocated on a run-time stack, rather than being initially located at a static address and possibly saved and restored later.

For calls via function pointers, the function pointer type, rather than a specific function, can be a node in the call graph which "calls"

all the functions whose addresses are taken and coerced to that pointer type. More conservatively, we could consider all function pointers to be a single node in the call graph.

This approach thus gives us the full semantics of C or Oberon at a much lower run-time cost on machines like the 8080. However, it requires whole-program analysis to precisely calculate which functions potentially participate in a recursive loop, and that might pose some difficulties for self-hosted development.

You could pretty much solve this problem with a linker, though. Each reference to an in-memory statically-allocated local variable gets relocated by the function's activation-record base address, and the linker is responsible for assigning those base addresses at link time and fixing up the relocations, as well as inserting trampolines, which would probably have to copy entire activation records rather than just the "live" parts. Probably you also have to runtime-stack-allocate every variable whose address gets taken, too, and you need to expose function-pointer types to the linker.

This is a nontraditional sort of linker, and it has to do more relocations than the usual kind, but it seems like it still ought to enable fairly powerful self-hosted development with separate compilation, because the object files should be about the same size as before.

It can even solve the problem of parameter passing — the caller gets relocations for the callee's activation record so that it can poke the arguments into the proper locations in memory and get results from the proper place, assuming you aren't passing arguments in registers.

This might still be useful on more modern small computers like the AVR and the STM32, because they are usually run in environments where they don't have a sensible way to report a failure. (See Patterns for failure-free, bounded-space, and bounded-time programming (p. 925).) Under normal circumstances, there is no way to statically bound the stack usage of a C program, because C supports recursive loops, and ruling them out requires whole-program analysis. If your stack overflows into your heap, which can easily happen on an Arduino, you may get incorrect results or your program may just crash.

As an even more nontraditional alternative, you could use a BibTeX-style two-stage compilation process; first, you compile each module based on certain assumptions, such as "function foo is nonrecursive", "function bar is possibly recursive", and "function baz takes an int and a char* and returns a char*", and "function quux preserves the contents of the BC register", which can only be validated by a whole-program analysis, and annotate the compiled module with the assumptions that were used, as well as the useful properties that were discovered (for example, foo calls bar). Then, at link time, you collate all the properties discovered into a database of useful properties, and check whether all the assumptions still hold true. Any module compiled with assumptions that turned out not to hold is then recompiled using the newly collated database, and the link step is repeated. If the properties discovered don't depend on the assumptions made, this will converge after just two iterations.

This approach has the additional advantage of eliminating the need for C header files.

Context switching with "buffer-local variables"

Multics Emacs was the first Emacs to be scriptable in Lisp, during the 1970s period I mentioned above. In Emacs, there are a lot of frequently-accessed variables that are local, not to a function call, but to a particular editor buffer; if you open two files at once, for example, you probably want to maintain independent cursor positions in them. The modern approach to doing this is to store all those variables in a record, allocate a record for each buffer, and maintain a pointer to the "current buffer" record, and associate a buffer pointer with each open window on the screen. Switching buffers is achieved by changing the value of the current-buffer pointer. This requires, as explained above, indexed-offset addressing to all these variables.

To avoid this extra cost, Multics Emacs used an approach similar to shallow binding: all the variables for the *current buffer* are stored in constant places in memory, and when you switch buffers, those variables are copied into the record for the old buffer, then copied out of the record for the new buffer. The rationale for this was that accessing things like the current cursor position is so much more frequent than switching buffers that it doesn't make sense to slow down access to the cursor position in order to speed up switching buffers.

(I suspect GNU Emacs uses the same mechanism even today, but I haven't looked.)

This principle is applicable to many kinds of records that enjoy a certain sort of locality of reference. It's common for a program to do a number of things to one file before switching to doing things to another file, for example, and many GUI programs, if they even use more than a single window at all, do many things in sequence to the same window. Image-processing programs frequently do many things in sequence to the same image, parsers frequently do many things in sequence to the same input stream, network programs frequently do many things in sequence to the same network socket, and database programs frequently do many things in sequence to the same table or cursor.

Such programs can probably work better on an 8080 by using the copy-in/copy-out approach used by Multics Emacs for its buffer-local variables. They may even be able to do this as an optimization — for example, your file access functions could take a file-number argument, maintaining the "current file" state entirely internal, but checking that argument against the current file number upon entry.

However, there are other uses of records that do not work as well in this approach. Pretty much anything that repeatedly walks a tree or linked list of the same kind of records is going to be slower rather than faster this way, including abstract syntax trees and database query plan execution.

Self-modifying code

The 8080 has no instruction cache, so there is no extra cost to self-modifying code, except your sanity. By inserting some instruction bytes between the data fields in a record, you can make the record executable. For example, this subroutine loads B, C, D,

and E with four bytes in 38 cycles (plus 17 in the CALL instruction or 5 in the PCHL instruction to access it, for a total of 55), and occupies 9 bytes of RAM:

```
MVI B, 33h    ; 7 cycles each
MVI C, 31h
MVI D, 12h
MVI E, E8h
RET           ; 10 cycles
```

By poking new bytes into the appropriate locations (overwriting the immediate operands, not the MVI opcodes) you can update the data structure.

55 cycles ÷ 4 8-bit variable reads = 13.75 cycles per 8-bit variable read, dramatically less than the 27 cycles each you'd need for an indexed-offset read. However, an indexed-offset read followed by three 12-cycle sequential reads totals only 63 cycles, so the improvement is less dramatic than it would seem at first.

In some cases, the data can be paired with a more useful operation than merely loading into a register; here we have a bitmask and a bitfield that set the low three bits of A to 5:

```
ANI F8H       ; A &= 0xf8; 7 cycles
ORI 05H       ; A |= 0x05; 7 cycles
```

For function-local variables in nonrecursive functions that are only read in one place, storing the variable value as immediate data reduces the cost to read it from 17 cycles to 10 for 8-bit variables, or from 16 cycles to 10 for 16-bit variables, while adding no extra update cost. If the variable is read in more than one place, the other places can use LDA or LHLD or the LXI/MOV sequences mentioned earlier to read it from within the instruction where it's an immediate operand; alternatively, if it's read more often than it's written, the write operation can update multiple immediate operands.

This seems like a quite significant performance advantage, although it does require whole-program analysis during compilation to be used in a language like C; and this "field padding" in the records can be expensive when you only have a 64-KiB address space. For local variables, though, storing variables inline reduces space usage rather than increasing it.

Earlier I pointed out that chasing a 16-bit pointer chain costs 23 cycles per pointer, or 37 cycles to swizzle SP plus 15 cycles per pointer. But chasing a pointer represented as a JMP instruction costs only 10 cycles; a CALL/RET pair, however, costs 27 cycles.

Avoiding 16-bit variables

16-bit variables are pretty expensive on the 8080, not just to store but also to manipulate, so it's worth avoiding them as much as possible.

The 8080 has some limited 16-bit ALU operations; as noted above, it can do 16-bit addition, fetch, and store, but it can also do 16-bit increment and decrement. However, these operations are

much slower than the 8-bit variety, and its 8-bit repertoire also includes subtraction, addition and subtraction with carries and borrows, bit rotations, and bitwise AND, OR, XOR, and NOT. If you want 16-bit AND, you need to synthesize it out of two 8-bit ANDs, taking 28 cycles:

```
MOV A, D  ; 5 cycles
ANA B     ; A &= B; 4 cycles
MOV D, A
MOV A, E
ANA C
MOV E, A
```

The 8080 also has very limited register space — 88 bits of architecturally accessible registers, not counting the PSW, 16 of which are the PC. (Compare to the 16 32-bit general-purpose registers in most modern CPUs: 512 bits.) So handling a 16-bit value adds a significant amount of register pressure.

This means that in most cases you should avoid having large arrays; limit them to 256 items. If you can 256-byte-align the arrays, you can avoid doing any arithmetic to index them:

```
MVI B, 28H      ; array base address; 7 cycles
MOV C, A        ; index array with A; 5 cycles
LDAX B          ; chase pointer; 7 cycles
MOV C, A        ; index array again
LDAX B
MOV C, A        ; and again
LDAX B
```

This takes 12 cycles to follow each 8-bit pointer after the initial 7-cycle setup. For an additional 7 cycles, you can load a second array base address into D and alternate between them. And by loading C (or E) with a field offset and chucking A into B (or D), you can make the pointers point to 256-byte "memory pages".

If your arrays aren't 256-byte-aligned but don't cross 256-byte page boundaries, you can do an indexed-offset thing:

```
LXI B, 2821H    ; array base address; 10 cycles
ADD C           ; A += C; 4 cycles?
MOV C, A        ; set up address in BC; 5 cycles
LDAX B          ; load from array; 7 cycles
MVI C, 21H      ; fix base address; 7 cycles
ADD C
MOV C, A
LDAX B
```

On the first iteration, though, this is 26 cycles, barely any faster than the more flexible 27-cycle sequence I started with. Subsequent iterations get down to 23.

Givens's qe in particular uses 16-bit pointers fairly extensively so that it can treat the editor buffer as an undifferentiated, flat sequence

of bytes, which simplifies the description of editing operations and file access, and would especially simplify search if he implemented it.

Topics

Notes on the STM32 microcontroller family

Kragen Javier Sitaker, 2018-06-30 (updated 2018-11-12) (42 minutes)

I think I am switching from AVRs to STM32s for seven principal reasons.

• Speed: STM32s are much faster, from almost twice the clock speed to over ten times the clock speed, with registers that are four times as wide, including a single-cycle 32×32→32 multiply — which would be 10 8-bit multiplies and 7 8-bit adds. And AVRs without an external crystal are limited to 8 MHz, while STM32s can run at full speed on their internal ±1% RC oscillators.
• Power: With an STM32, you can run 32 32-bit instructions on the energy the AVR needs to run a single 8-bit instruction. AVRs, despite their "picoPower" marketing, use about 7500 picojoules per instruction (pJ/insn), and those are 8-bit instructions. The STM32F0 uses about 1500 pJ/insn, and the STM32L0 uses about 230 pJ/insn.
• Cost: The STM32 costs more than the AVR, but only slightly, and less per pin or per MIPS. The 48MHz STM32F031x4, with 39 GPIOs, costs US$1.30 in quantity 1 on Digi-Key. The cheapest reasonable† AVR is the 20MHz ATtiny13A, which costs US$0.40 and has 6 GPIOs.
• Size: The STM32L is *tiny*, 2.133 mm × 2.070 mm in the WLCSP package, with 21 GPIOs (two of which you can run I²C on). The AVR ATtiny4/5/9/10 is physically the same size, but only has 8 total pins and 4 GPIOs.
• Scalability: STM32s have a much higher ceiling; AVRs top out at 20 MHz with a limited range of peripherals, while STM32F7s run at 216MHz and have all kinds of crazy things.
• ADC: STM32s' ADCs are 12-bit and run at 1Msps and up, while AVRs' ADCs are 10-bit and run up to 15ksps at maximum resolution.

• Debugging: AVRs' DebugWIRE protocol, in a big FUCK YOU to users of free software, is undocumented, while STM32s have a debugging interface called "serial wire debug" which is supported in OpenOCD.

† The ATtiny4/5/9/10 are cheaper at 17¢ but are almost useless.

Introduction

I think the STM32F031 is going to be my new favorite microcontroller, replacing the AVR. It's a 32-bit 48MHz Cortex-M0, it costs US$1.30 in quantity 1 on Digi-Key, and there is already Arduino support for it, which should make it easy to get started. Like the ATMega328, it has 32 (or 16) kilobytes of Flash and 4 kilobytes of SRAM. And, in addition to easier-to-handle sizes with shitloads of pins (up to 39 GPIOs!), it comes in a 2.4-mm-square WLCSP25 form factor. And it uses one fifth the power of the AVR at the same clock speed!

(Hmm, the above mostly refers to the STM32F031x4/6, but I might have gotten some details from the STM32F030x4/6/8/C

mixed in there. The 4/6 is 16 or 32K; 8 and C are 64 and 256 K, and more interestingly, up to 32K of SRAM. These larger chips also come with up to 55 GPIOs. There are also STM32F3 and STM32F4 families, which are progressively more awesome; the F4 runs at 168MHz and has three 5Msps DACs, and both have floating-point.)

On the bad side, it still doesn't have a radio (unlike the ESP32) and it only runs up to 3.6 volts — 5 volts is right out.

There's an overview by Satoshi NM.

Instruction set

ARMv6-M with Thumb-2, ONLY. RAWK. Nothing comes close for machine code density. And it has a 1-cycle 32×32 → 32 (☹) multiply, which is pretty astounding even if it does discard half the result.

The STM32Foxxx Cortex-M0 Programming Manual (PM0215) doesn't include instruction timings at all except to say that the multiplier takes a single cycle, actually.

There's a separate stack pointer for interrupt handlers if you enable it, so that the interrupt handlers (I think there can be up to 7 on the stack at once) don't eat into the worst-case stack space of every thread.

Software support

OpenOCD even supports the "serial wire debug" interface, which apparently allows you to turn the microcontroller into the puppet of your workstation using two pins.

GPIOs and 5V

It does have 5V-*tolerant* I/O pins, though not all of them. Also, its GPIOs *can* be configured as open-drain rather than push-pull, which has two huge benefits:

• They can communicate with 5-V systems if there's an external 5V pullup;
• Perhaps they can switch somewhat heftier loads?

Supposedly the pins can source or sink 20 mA each, and up to 80 mA over the whole chip, which I think is inferior to the AVR's 200; still, if I could hook a 12V 100mA motor directly to four open-drain GPIO pins, that would be pretty awesome. I don't think I can because they get configured as input pins during and after reset, and those aren't supposed to be subjected to more than 4 volts.

Some other STM32s have a special pin for sinking a lot of current from an infrared LED.

The GPIO pins can also be configured with internal pullups, pulldowns, or neither.

Hmm, wait, it says the 5V-tolerant pins can be subjected to V_{DDIOx} + 4.0 volts as long as the pullups and pulldowns are disabled. What's V_{DDIOx}? Oh, I guess that's the 2.0V to 3.6V Vdd. So 7.6V is reasonable. But anything over 5.5V is outside "general operating conditions".

The smallest and cheapest ones only have 15 GPIOs, which is still pretty decent; that's under 9¢ per GPIO!

The massive number of GPIOs can toggle at up to half the clock

speed, i.e. 24MHz, supposedly. I don't quite understand how this works; do you just run a sequence of instructions that loads immediate 16-bit values into some CPU register and then writes them to the memory-mapped I/O register `GPIOx_ODR`, I guess? Or is there some way to feed the GPIOs with the DMA controller?

Analog

Its ADC is 12-bit rather than the AVR's 10-bit, and it's supposedly 1µs! Does that really mean 1Msps? It apparently does… and it has 9 or 10 external input channels. And it supports DMA, and there's a "DMA circular mode" (`DMACFG`=1) to just dump stuff into a ring buffer — ideal for seeing pre-trigger data!

It doesn't seem to have an analog comparator input, unfortunately. You have to fake it by configuring the ADC to do conversions constantly and fire an interrupt when they hit a threshold.

Another drawback compared to the AVR is that the ADC's full-scale reading is always Vdda, which must be at least as high as the power supply — at least 2.4 V. It has a calibrated internal bandgap voltage reference to measure against, but it can't set that as the ADC reference voltage directly. This means that the 4096 counts in the output are spaced at least 580 µV apart, and 880 µV at 3.6 V, while the AVR's 1024 counts can be spaced 1.07 mV apart. Some other STM32 chips support a separate VREF+ (alluded to in ST AN2834), but the STM32F031x4/6 does not. Only STM32s with 100 or 144 pins do. ST suggests solving the problem with a preamplifier.

The ADC requires at least 2.4 V, although most of the rest of the chip will run down to 2.0 V. And it uses 1 mA at its normal speed.

It's possible to configure the ADC for lower bit depth (10, 8, or 6 bits) to speed it up (to 928, 785, or 643 ns, respectively), which suggests you could crudely digitize signals up to 780 kHz with one STM32. The sample-and-hold is 1.5 cycles of the 14MHz ADC clock, so about 107 ns. This seems like it is likely to sharply attenuate frequency components above about 10MHz even if you can get several chips sampling them at once.

The Vdda pin is supposed to draw only about 0.9 mA, so putting it on a separate regulator might improve precision.

Like the AVR, it has an inaccurate temperature sensor: ±5°, better than the ±10° on the AVR. (Hmm, at least on the STM32L, the Vsense linearity with temperature is at worst ±2°, and at least 1.48 mV/°, which you would think would give you 1° accuracy? Maybe it's just `TS_CAL2` that is ±5°?)

In terms of PWM, well, it has 9 timers and 11 (?) PWM channels, which can run off 16-bit timers instead of an 8-bit one, which should give a lot more dynamic range. It even has a 32-bit timer!

Some of the timer counters can be run off an external quadrature input instead, which should enable quadrature feedback up to fairly high pulse rates.

The PWM counters have an adjustable "auto-reload value" and can count in either sawtooth, inverted sawtooth, or triangle form. I think you can run them off a 48 MHz clock, which ought to enable you to generate phase-correct PWM waves at 24 MHz and factors thereof (24 MHz, 12 MHz, 8 MHz, 6 MHz, 4800 kHz, 4 MHz, 3428 kHz, etc.) Or maybe 12 MHz.

They also have a dead-time mode specifically designed for

H-bridges.

Buses

It has I²C (including wakeup from Sleep, 10-bit addressing, multiple 7-bit slave addresses, and 1 Mbps "Fast Mode Plus") and an 18Mbps SPI interface. These, like the ADC, support DMA, so you don't have to handle interrupts after every byte like on the Arduino.

Its USART, which also supports DMA, can run at up to 6Mbps, which could be handy for RS-485.

The DMA support means that the CPU can sometimes be delayed in its access to memory, which seems like it could cause some problems for cycle-accurate bitbanging. The CPU doesn't seem to have a cache (why would you have a cache when all your RAM is zero-wait-state?), but it does have a 12-byte instruction prefetch buffer, which could also cause unpredictable timing; it can be disabled with `FLASH_ACR`.

The larger STM32F030x4/6/8/C has two I²C interfaces instead of one and two SPI interfaces instead of one. Other STM32 chips have a wide variety of peripherals available.

Clocks

In addition to an external crystal, in theory, like the AVR, it has an 8MHz internal oscillator — with an optional 6× PLL to bring it up to 48MHz.

A neat thing is that if the external crystal fails, it automatically switches to its internal oscillator. (But then it fires an NMI, and your ISR needs to clear the CSSC bit in `RCC_CIR` to escape the NMI loop, so it's not totally transparent.)

Power

Supposedly it normally uses like 20mA at 3.6 V and 48 MHz or more like 12mA if all the peripherals are turned off. This works out to something like 1.5 nJ per 32-bit instruction if we figure on 1 instruction per cycle. Strangely, the datasheet doesn't supply current consumption figures at 2 V.

This scales linearly down to about 3 mA at 8 MHz, then down to about 1mA as frequency approaches zero.

It has four modes: Run, Sleep, Stop (3.2–55 µA), and Standby (0.7–3 µA), which last is sort of turned off actually, losing all the memory contents. It can run down to 2V. Stop mode still has a bunch of possible interrupt sources as wake sources, though I'm not sure if timers are among them. (I think its RTC may be capable of doing this.) Presumably getting out of Standby involves rebooting, and it says the RTC can do this. Also the watchdog timer can wake it from Stop or even Standby.

The RTC has a separate supply and consumes 0.5–2.1 µA when enabled. It can continue working down to 1.65 V.

Wakeup from Stop mode takes 2.8 to 9 µs; wakeup from Standby takes like 50 or 60.

All of this would seem to imply that it is sensible to use Stop mode for duty cycles down to about 0.1% as long as the wakeups are for at least, say, 20 µs of execution — 1000 instructions or so. Which means that you can reasonably wake up every 25 ms for 1000 instructions,

using about 2 µJ while awake and another 1 µJ while asleep, for a total of 120 microwatts, 600 times lower than full power consumption of 72 mW.

72 milliwatts is about three red indicator LEDs.

(I'm not yet sure how clock speed depends on supply voltage.)

Standby mode might use 5 microwatts, and so makes sense for duty cycles down to 0.007%, but requires 100 µs or so (5000 instructions) of execution. I mean, you could do less execution, but then you would be spending way more energy on rebooting than on actually getting anything done. At this level you're waking up no more often than every 1.4 seconds or so. In this mode nearly all the GPIOs are tristated.

I think there may be some wrinkles around stabilization time of the system clock PLL I may be missing here. Like, if the PLL is disabled, you only get 8 MHz, not 48. But if the PLL is enabled, it might take a while to get sync (up to 200 µs, according to the datasheet). And the PLL automatically gets disabled whenever you enter Stop or Standby, though not Sleep, so you always wake up at 8MHz.

Note that erasing the Flash takes 20–40 MILLIseconds, during which time presumably the thing is using as much power as normally, or even more.

A 3-gram Energizer CR2032 lithium coin cell could power the STM32 directly, and its 240mAh down to 2 volts works out to 2.2 kJ. This would power the STM32 at full speed for 8 hours (if it can deliver a reasonable amount of current; its internal resistance is 20Ω until near the end, which in theory means it can deliver 150mA), in Stop mode 99.9% of the time waking up 40 times a second for 6 months, or in Standby mode waking up every 1.4 seconds for 14 years. (In either case you can use the IWDG independent watchdog or the RTC real-time clock to wake up at a scheduled time.)

To power the STM32 at full speed for a full second directly from a capacitor would require a 16000 microfarad capacitor. A 1-farad supercapacitor could theoretically run it for a full minute. If you had a buck converter and a boost converter, you could use a higher-voltage electrolytic. For example, the last inkjet printer I took apart has a 100µF 50V electrolytic on its main board, which is a bit smaller than my last pinky joint; this works out to ⅛J, or 1.74 seconds at full power.

The larger STM32F030 uses about 10% more power at the same high speeds.

1.5 nJ/insn is slightly higher than the 0.9 nJ/insn of the MSP430 family, but much higher than the 0.3 of an LPC1110 (see Keyboard-powered computers (p. 2220)). Atmel has a line of "picopower" ARM microcontrollers that are around 0.25 nJ/insn (see Low-power microcontrollers for a low-power computer (p. 2602)). Much-lower-power microcontrollers exist but are not widely used (see Keyboard-powered computers (p. 2220) and Low-power microcontrollers for a low-power computer (p. 2602).)

However, the most relevant comparison is the ATmega328, which (despite sharing the "picopower" moniker) gobbles 7.5 nJ/insn (see Low-power microcontrollers for a low-power computer (p. 2602)). So the STM32 family uses one fifth the power to process four times as many bits!

Relevant comparisons include: that an E-ink display uses about 3

mW during continuous reading at a comfortable rate; the ADC also uses 3 mW while running; and 3 mW from a 12%-efficient solar cell is about ¼ of a square centimeter.

Memory

It has a bootloader that can reprogram its own flash. Also it can somehow boot from its own SRAM. Erasing a 1KB page of the flash takes 20–40 ms, and programming a halfword takes 40–60 µs, which is like two or three thousand clock cycles.

Of course ARM is not Harvard, and so not only is C programming somewhat simplified, but also, it's possible to execute code from SRAM. In fact, that's mandatory if you want to keep executing code while erasing or even writing to the Flash, and the SRAM is zero-wait-state.

You don't get the full benefit of the implicit AND operation provided by NOR flash — the micro tries to protect you from yourself by refusing to program non-erased locations — except that you *can* overwrite arbitrary halfwords of the flash with 0x0000 without erasing entire pages. I feel like this could be useful.

Board support

The popular Blue Pill STMDuino is a Chinese design built around a STM32F103C8, which is a somewhat higher-end STM32: 72 MHz, 64 or usually 128 KB of Flash, and 20 KB of RAM. This is a Cortex-M3 rather than M0. (M4 is the one with the FPU, saturating arithmetic, and MAC.). I think it's the same size as an Arduino Micro or Nano, I'm not sure. It's 53.0 mm × 22.5 mm, just slightly larger than an Adafruit Feather.

The WLCSP25

The 2.4mm-square WLCSP25 that the STM32F031x4/6 can come in is particularly appealing to me. It's just so incredibly tiny; it's only 2.458 mm × 2.36 mm × 585 µm at most, and that includes its balls. It should weigh under 10 mg, since it occupies 3.4 µl. However, a few sacrifices are made to get it down to 25 pins, or rather solder balls. Port A has pins 0–10, 13, and 14, but is missing pins 11, 12, and 15. Port B has pins 0–1, 5–7, and that's it; port F has just pins 0 and 1, which are for the external oscillator. These 20 GPIOs, plus Vdd, Vss, Vdda, BOOT0, and NRST, suck up all the pins.

Port A and PB1 have the analog input channels, so it's fully supplied there. Port A also has the USART, the SPI interface, and the I²C interface, and most of these have alternate pins among those included from port B. PA13 and 14 have the SWD debugging interface.

It's missing the battery-backup pin for the real-time clock, it's missing the connections for a 32.768kHz watch crystal, and 8 of the pins aren't 5V-tolerant (namely, PA0–7), but it doesn't actually seem to be missing anything essential except moar GPIOs.

So, if you want to max out your GPIOs, you have 20 GPIOs, or if you want to preserve debuggability, you have 18 GPIOs and two SWD pins (PA13 and PA14). If you want to use it as a debuggable I²C slave to twiddle and maybe process GPIOs, you additionally dedicate either PA9–10 or PB6–7 to I²C, and you have 16 GPIOs. If

you need high timing precision, you put a crystal on PF0–1. If you want SPI, you can have it (at 3V) on PA4–7, sacrificing three or four analog pins. And you have PWM outputs on all the GPIO pins except PF0–1 (the oscillator pins) and PA13–14 (the SWD pins), except that I think a couple of those pins are actually just used as triggers for the PWM waveform (PA0 and PA5 are TIM1_CH1_ETR and TIM2_CH1_ETR, which I don't understand.)

Holy *SHIT*, the STM32L

I didn't realize this at first, but since 2015, there's *another* line in the STM32 lineup called the "access line ultra-low-power" STM32L, the low-power processors. And these are even *smaller* and also lower-power. And their ADC is *even faster* at 1.14 Msps, with DMA. *And* they have two analog comparators instead of zero. They still have SWD, I²C, and SPI ("only" 16Mbps, and then only above 2.7V; at 1.71V it can manage 12Mbps, and at 1.65V 8Mbps). They're a bit slower, though, at 32MHz, but they're still Cortex-M0+ with a single-cycle 32×32→32 multiplier.

The STM32L011x3/4 goes for US$1.50 on Digi-Key, and it has a WLCSP25 package (STM32L011ExY) that is 2.133 mm × 2.070 mm. Miraculously, it has *more* GPIOs than the larger STM32F0's WLCSP25: sort of 21 rather than 20, because the BOOT0 pin can be configured as PB9, though input only. It has less memory (the 3 has 8K of Flash, the 4 has 16K, both have 2K of SRAM) and it can operate down to 1.65 V, or 1.8 V if the brownout reset circuit is enabled.

The WLCSP25 pinout is gratuitously different from the STM32F031x4/6 WLCSP25 pinout, but since it's a different physical size, you couldn't use it in the same circuit board layout anyway. Almost the same set of pins is mapped: pins 0–10, 13, and 14 on port A, pins 0–1, 5–7, and 9 on port B, and pins 14 and 15 on port C, rather than 0 and 1 on port F, but those are still the external oscillator pins.

The friendliest way to get started with the STM32L might be using the LQFP32 package, which is 7 mm square and should be easy to hand-solder, since its pins are every 0.8 mm.

The GPIO currents are a bit lower: 16 mA sunk by three-volt pins, 22 mA sunk by five-volt pins, and 16 mA sourced by either.

STM32L ADC

The 12-bit ADC not only runs at 1.14 Msps down to 1.65V, it only uses 25µA at 10ksps and 200µA at 1Msps, and it supports up to 256× hardware oversampling in order to fake up to 16-bit sampling. So you could get 16-bit samples at 4.4 ksps.

STM32L power

The front page of the datasheet claims 0.95 DMIPS/MHz and "down to" 76 µA/MHz in Run mode. At 1.8V, if we ignore the "down to" clause, this would work out to 0.144 nJ/insn, 144 pJ/insn, dramatically lower than the STM32F0; even at its max of 3.6V it's only twice that, 290 pJ/insn. The aforementioned 3-gram Energizer CR2032 lithium coin cell with its 240mAh down to 2 volts (2.2 kJ) could thus power it for 3200 MHz-hours — 100 hours at 32 MHz, 200 hours (8 days) at 16 MHz, 400 hours (17 days) at 8 MHz, 3200 hours

(4 months) at 1 MHz.

Perhaps more interesting still, given a 1-farad 2.7-volt supercapacitor, which has 2 J and 0.9 coulombs down to 1.8 V, it could run for over 3 MHz-hours; with a 90%-efficient buck-boost regulator running the supercap down to 1.3 V (2.8 J), it could run 18 billion instructions, 4.9 MHz-hours. Such a capacitor as the 104µℓ 4Ω 1F Nichicon JUWT1105MCD (see Capacitors: some notes on tradeoffs (p. 134)) has a time constant of 4 seconds, so it can mostly recharge within 4 seconds and 95% charge within 12 seconds.

Digging in a bit more, above 16 MHz it needs at least 1.71 V, and there are two other "power consumption ranges" up to 16 MHz and 4.2 MHz, which apparently affect the number of wait states (perhaps to Flash only?). Power consumption is about 1 mA in sleep mode at 16 MHz with all peripherals off, and there are seven different low-power modes. In Standby mode, it consumes 180 nA without the RTC enabled or 410 nA with the RTC. It has a "low-power run" mode which can run at 32, 65, or 131 kHz, which, at 25°, runs at 5.7–17µA from RAM or 18–32µA from Flash, which works out to 210 pJ (1.65 V, 131 kHz, RAM) to 400 pJ (3.6 V, 32 kHz, Flash) per instruction; typical consumption for normal run mode ranges from 320 µA (4 MHz, running while(1) from RAM or with prefetch off) to 5.4 mA (32 MHz Dhrystone from Flash). Likely the most efficient is 1.95 mA for 16 MHz from RAM, which is further explained as "HSI16 clock source (16 MHz), Range 2, VOS[1:0] = 10, Vcore = 1.5 V, Flash switched OFF". If we figure that this is 15.2 MIPS and is being powered at 1.8 V, then that's 3.5 mW, 230 pJ/insn.

So I can't figure out where they get the "76 µA/MHz" claim from, because the best I can find in their measurements is 122 µA/MHz. Oh, wait, I think I see how to get partway there — the 1.95 mA is "code with data processing", which I think is Dhrystone, and `while(1)` uses about 30% less power. That gets you to 85 µA/MHz but not to 76.

This is still enormously better than, say, the 900pJ MSP430F2001.

A tricky bit here is what kind of suspend mode you can get away with using when there isn't stuff to do. You have Sleep (1 mA at 16 MHz), Stop (with or without RTC, retaining RAM contents), and Standby (with or without RTC, losing RAM contents). Aside from the wakeup time, the problem is that in Standby you can only be woken by the brown-out reset, power-on reset, RTC "tamper", something completely undocumented called "Auto WakeUp (AWU)", the independent watchdog, and two of the GPIO pins. By contrast, in Stop mode, you can also be woken up by the programmable voltage detector (PVD), the analog comparators, the LPTIM, all the GPIOs, the RTC (if it's running), the USART, the low-power UART, and I²C, though not SPI. And it only takes 5 µs to wake up instead of 65 µs.

In exchange for the wakeup limitations, you cut your power; at 1.8 V:

mode	µA	µW	time to drain	time to drain	duty
		1.8 V	CR2032 2.2 kJ	1F 2.8J	cycle
			coin cell	supercapacitor	
16MHz Run	1950	3500	175 hours	13 minutes	

16MHz Sleep	450	810	31 days	58 minutes	23%
131kHz Flash run	32	58	14 months	14 hours	1.66%
131kHz RAM run	17	31	27 months	25 hours	0.87%
Stop w/RTC	0.54	0.97	72 years	33 days	0.028%
Standby w/RTC	0.41	0.74	94 years	44 days	0.021%
Stop, no RTC	0.29	0.52	134 years	62 days	0.0149%
Standby, no RTC	0.18	0.32	220 years	101 days	0.0092%

The "duty cycle" column is the duty cycle at which the power used in low-power mode equals the power used in run mode at 16MHz. At greater than this duty cycle, the majority of power used is in run mode, so the low-power mode's power consumption is in the minority; at less than this duty cycle, run mode's power consumption is in the minority.

This makes me think that as long as your duty cycle is above 3% or so, you should just use low-power run mode instead of suspending, and above 0.1% or so, it isn't worth worrying about the differences between the different suspend modes, so you should just use stop with RTC; but below 0.03% or so, the differences between them could potentially give you a factor of three improvement in lifetime.

The Sleep mode doesn't give a huge improvement on power, but it's very cheap and only takes 7–10 clock cycles to wake up from.

The wakeup latency means that, above a certain frequency of wakeups, you no longer get the benefit of a low duty cycle. At the 5 µs wakeup time for Stop, for example, you get to the 0.028% duty cycle at 56 Hz, even if all you do after the wakeup is just go right back to Stop.

(The wakeup time even from Stop actually depends on the clock speed and the voltage regulator configuration and can range from 5.1 µs to 260 µs (!), and Standby wakeup can range from 65 µs to 3000 µs.)

Such a small device will probably have appreciable power usage in its communications with other devices. In "Range 2", the I²C interface uses 8.2 µA/MHz, GPIOA uses 6.3 µA/MHz, and GPIOB uses 4.1 µA/MHz. If all three are turned on, they total 18.6 µA/MHz, or 300 µA at 16 MHz, adding about 15% to run-mode power consumption, but increasing stop-mode power consumption by a factor of over 600. But maybe you could enable just I²C while stopped.

If it's communicating, though, in addition to running its own GPIO port, it needs to drive the input impedances of the other chip and the lines in between. Different input pins have different specified input leakage currents: ±50 nA, 200 nA, 500 nA, and they're specified as having a typical input capacitance of 5 pF, which means that charging them up to 1.8 V will cost you 9.0 picocoulombs, the same as you spend on the leakage current in 180 µs, 45 µs, or 18 µs, according to which leakage current you figure with.

9.0 picocoulombs on half the bits you transmit (the ones that differ from the previous bit) at 1.8 V gives you a cost of 8.1 pJ per bit, which triples to 24 pJ for separately-clocked serial protocols like I²C and SPI.

8 or 24 pJ per bit is relatively significant compared to 230 pJ/insn, since instructions generally produce 32 bits, which means 7.2 pJ per output bit. But it's far from the overwhelming cost I was fearing.

Local availability

I can get an STM32F103C8T6 here in Buenos Aires for AR$154 (US$5.39) or some kind of STM32F030 from */while1/* in Liniers for AR$99 (US$3.46). Arduinos based on the STM32F103C8T6 are even more common and cost from AR$200 to AR$300 (US$7–10.50). Large volumes would therefore seem to depend on importation.

The 103 has a Cortex-M3, USB, 72MHz, CAN, two 1Msps 12-bit ADCs, 20K of SRAM, 64K of Flash, three 16-bit timers, two I²C interfaces, JTAG, and 1.25 DMIPS/MHz. It runs down to 2.0V. It seems to come in LQFP100, LQFP64, and LQFP48 packages. The "C" in the part number means these are 48 pins, which means 37 GPIOs.

In particular "Arduino Arm Stm32 Cortex-m3 Stm32f103c8t6 Mona" is Monarca Electronica in Flores for $220, with male headers. Monday to Friday 10:00–13:00, 14:00–19:00. Gavilán 58, C1406AWA Buenos Aires, 011 4634-2407, info@monarcaelectronica.com, according to Gooble and Mercadoshops. Also WSAP 1162241486. I could go there TOMORROW.

Fun projects

You should totally be able to bitbang NTSC or PAL color from this bad mofo, or for that matter AM radio. You should even be able to do AM radio with their PWM outputs, since that's the 540 kHz to 1610 kHz range. The PWM waves unfortunately have relatively few power levels available, but the slightly lower frequencies actually within the AM radio spectrum have more; 545.5 kHz is subharmonic 44, so you should have 22 meaningfully different power levels available.

Given the 24MHz max GPIO toggle rate, a TV typewriter could in theory manage 400 kilopixels per 60Hz NTSC field, or 800 kilopixels on the screen. But NTSC is actually limited to about 6MHz of bandwidth, so it's more like 100 kilopixels per field or 200 kilopixels per frame — about 400 × 500. This is enough for 60 lines of 80 columns.

Rebraining a calculator with one of these bastards should give you access to virtually unlimited computational power.

The 8MHz HSI RC oscillator has an 8-bit HSITRIM register which adjusts its frequency up or down in steps of about 40kHz, about 0.5%. If scaled up into the FM radio range, these are steps of about 0.5 MHz — too big to get usable FM radio out of. But maybe there's another way to bend the HSI frequency by adjusting the voltage, current, or temperature of the chip. You're lacking about a factor of 4 in speed to do direct digital synthesis of FM radio waveforms without an analog front end, but maybe you could generate, say, a 14 MHz square wave and filter out its 7th harmonic. This would be about 3.4 CPU clock cycles per full oscillation, so it would be kind of tricky, but maybe you could do it.

All the trimming and calibration and precision in the ADC and clocks, plus the various push-pull/open-drain/pullup/pulldown optionson the GPIOs, should enable a much better version of the M328, especially if you use a crystal. And 3.6V or 2.4V should be a lot safer for the DUT than 5V. (Does the M328 run at 5V?)

The AVR could only sample from its ADC at very low sampling

rates. Even the cheapest STM32 can do 1 Msps and so you should be able to digitize 500 kHz waveforms. This gives you about 2.5% of a decent oscilloscope, which ought to be enough to debug slow circuits.

If you could gang up several STM32s, you could perhaps make 35% of a decent oscilloscope, or even a bit more. 14 STM32s adequate to the job should cost under US$10.

Alternatively, it ought to be feasible to build an SDR for broadcast AM or FM radio using that megasample of capture bandwidth.

Audio communication and touch detection should be easy.

The Magic Kazoo should definitely be built around one of these bad boys. So should lots of other musical instruments.

The 25-to-55-kΩ pullups and pulldowns should source or sink a few hundred μA. In theory you could use this to very dimly light an LED, or charge a capacitor or reverse-biased diode by a very small amount to measure light, heat, radioactivity, or vibration. Into 10 pF of capacitance you should get a few tens of mV/ns (MV/s), or a bit under a volt per 48 MHz cycle. According to the datasheet, a Vishay 1N4001 is about 15 pF and has a reverse leakage current that rises exponentially with temperature, from 0.2 μA at 50V at 25° up to 5 μA at 100° and 180 μA at 150°. Fortunately or unfortunately, it's also near exponential with voltage, being something like 0.03, 1.5, and 20 μA respectively at the 7% of peak reverse that 3.6 V represents. Unfortunately the input leakage current is given as up to ±0.2 μA in analog mode, which would overwhelm the smaller numbers in this list. 1.5 μA at 15 pF is 100 kV/s or 100 mV per 1μs ADC sample time. (The STM32 pins themselves weigh in at 5 pF.) A lower-voltage diode might be a better choice as a temperature sensor.

You should be able to use the hi-Z input state for detecting electrical fields, given this 0.2 μA number. Maybe a capacitive divider would be useful.

It should be feasible to do a pretty fast 16-bit-wide logic analyzer using one of the 16-bit GPIO ports. I don't know how often you can read them but it seems like it should be at least 10 million times a second, producing 20 megabytes per second. Probably you could stream such data out to an SPI peripheral at the much slower 18 megabits per second the SPI interface supports.

For parallel computing, it should be feasible to implement the PAPERS Beowulf barrier synchronization primitive on a cluster of these guys using their open-drain/pullup configuration. As each microcontroller reaches the barrier, it writes a 1 to its pin, but continues to read a 0 from it until the last open-drain output on the bus gets set to 1. Given the 20 mA sink current capability and the 25 kilohm minimal pullup resistance, it should be possible to synchronize over 100 microcontrollers within microseconds this way. If only a small number of microcontrollers enable their pullups, it should be feasible to scale to thousands. This approach is actually used in the Gestalt FABNET bus.

It should also be possible to compute a crude majority rule function in this way, by using the pullups and pulldowns to "vote".

Parallel computing is especially appealing given the STM32L0's six times lower cost per instruction, with the limitation to 16MHz. You could plausibly build a 3×3×3×3 81-processor hypercube with 162 kilobytes of RAM and 1.30 megabytes of Flash, capable of 1.2 billion

instructions per second, that weighed 100 milligrams and ran (for embarrassingly parallel problems) on 1.8 V, 160 mA, 280 mW, for a BOM cost of under US$100. It might measure 7 mm × 7 mm × 7 mm if the circuit boards and capacitors don't take up too much space. Without any external switches, it could scale its power usage down to 26 microwatts by moving all the processors to Standby; with one, it could power the unused processors off entirely, leaving a power of 0.32 μW for the one left in Standby.

It should be straightforward to do the DSP necessary for short-range ultrasonic communication even at hundreds of kHz, if you can find adequate transducers.

I think the key to inventing ubiquitous computation, if found in a literary genre, is going to be fantasy, not science fiction. Sterile tropes of human-like computers with voice interfaces do not help us to imagine how we might actually interact with these devices. If we can give the objects in our environment new rules — whatever new rules we want — what rules would we like to give them? If you can enchant the objects in your environment to act however you want, what would you have them do?

Perhaps you would have your possessions follow you around, put themselves away when you got home, and scream or fight back when they were being stolen. Your bullets and arrows would never miss, and your clothes would become hard as rock when the bullets of others were about to hit them. Your utensils would change color or cry out in pain if they recognized a poison in your food. Your clothes would always keep you dry and at a comfortable temperature, despite being thin and light, and would never rip or stain. You would know everything about whoever you met before they told you, and you would know whenever you were in danger. You could reach the top of the highest tree or through the narrowest keyhole, and hear a whisper across the city if it had your name in it. You could climb walls like Spider-Man, fly through the air like a bird, or run faster than a horse; you could send objects flying to wherever you wished. You could call down lightning and fire to destroy your enemies, or turn them to stone, mud, or pillars of salt. Your house would clean itself, and it would change its appearance as you wished.

The GD32

There's a similar line of ARM SoCs from a Chinese company called GigaSomething (GigaDynamics?) with part numbers that seem to mirror the STM32 line: GD32F107 and so on. I wouldn't be surprised if these were manufactured as drop-in STM32 compatibles with some competitive advantage: lower prices, being able to speak to the company only in Chinese, more predictable stocking, shorter lead times, or something.

Topics

Binate and KANREN

Kragen Javier Sitaker, 2018-12-02 (3 minutes)

You can construct the relation describing your desired program by composing it from binary relations.

I just woke up from a nightmare where I was scheduled to give a talk tonight at one of my two employers about some work I had just done, when the other employer informed me that they considered that work confidential to them and covered under my NDA to them, so I wasn't going to be able to give the talk. Since neither employer actually exists, this document briefly describes the work.

Binate is a particularly terse way of writing down relations by algebraically composing them from more primitive, binary relations. It can only express binary relations, but it can express binary relations among tuples, so that isn't actually a limitation.

In particular, you write relations in terms of more primitive relations using conjunction or intersection `,`, union `;`, converse or inverse `~`, composition (concatenation), transitive closure `*`, either negation `!` or set subtraction `-`, and a form of N-ary Cartesian product: `{x: a, y: b, ...}` produces a relation from the intersection of the domains of a and b etc. to a set of tuples which are the domain of the new relations x and y etc., whose codomains are the codomains of a and b etc. respectively, constructed in the only reasonable way. Literals are treated as relations from the entire universe to that literal. This turns out to be sufficient to express anything you can express in Codd's N-ary relational algebra or relational calculus.

Kanren is a family of relational programming languages in which you program not by constructing functions but by constructing more general relations, which is to say that when you invoke them on different occasions, you can change your mind about which arguments are inputs and which are outputs. This sounds like Prolog, but because Prolog contains a lot of extralogical operators, its ability to generalize is limited.

When using Prolog semantics, the major advantage of Binate over Prolog is that programs are dramatically terser because they are point-free. For example:

```
ancestor = parent parent*.
father = parent, "male" ~sex.
sibling = parent ~parent.
cousin = parent sibling ~parent.
```

The major advantage of Kanren and relational programming over functional programming is that, in cases where you need to use both a function and its inverse, you avoid having to write the inverse explicitly, which sort of doubles your programming power. Indeed, a function of multiple arguments may have many inverses, and you only have to write it once. Additionally, it's often much easier to describe a predicate you would like to find instances to satisfy than it is to describe an algorithm to generate them — this is the general virtue of constraint-oriented programming or programming with solvers — and at times you would like to change the search or solver

algorithm without changing the predicate you are trying to satisfy.

So, the trade-secret insight from my dream was this: by composing your program out of binary relations, you can describe the computation you want in a somewhat terser fashion than any previous language. You can beat APL for terseness.

Topics

wood and stone personal digital assistants

Kragen Javier Sitaker, 2007 to 2009 (6 minutes)

Polished-Stone Handheld Computers

So I've been thinking about making a handheld computer with the look and feel (shininess, irregularity, weight, seamlessness) of a polished semiprecious stone.

One way to do this would be to embed the electronics in polyester resin poured into a mold, with an embedded induction coil for charging, some embedded lead shot for weight, and a dark, but not quite opaque, surface layer to hide the interior except for when it was glowing. Input would probably be piezoelectric, localizing surface taps or using rhythm. (See earlier kragen-tol post magic boxes and secret knocks.) Output could be through embedded LEDs shining through the surface layer or through audio, especially if you held it against a window.

(How much lead shot would you need? Lead has a density of 11.3g/cc, against quartz's 2.6g/cc and the polyester resin's 1.11g/cc, so only 14.6% of the volume would need to be lead to equal quartz's density.)

It would be shockproof, waterproof, crushproof, not particularly prone to damage from ESD, and it would feel really good in your hand. Some hard silicone around the outside might improve its thermal conductivity. (There are hard silicone resins with high thermal conductivity, right?)

Beatrice suggested that you could use an actual polished semiprecious stone instead; cut out a circle from one side, drill out a cavity underneath, put the electronics inside, pot them with epoxy, replace the circle, wipe off the excess epoxy, and then polish the result.

Wood-Block Handheld Computers

Another "everyday object" kind of electronic device case: a block of wood. Some time ago I saw a web page about a wooden clock. It seems to be widely available now; for example, http://svp.co.uk/products-solo.php?pid=4989&ref=froogle&ci_src=o o18615224&ci_sku=8028 advertises it for £93.99. It explains:

> A totally minimal block of wood with digital numbers floating across the surface. These clever clocks have a very thin layer of real maple wood veneer that permits the LEDs to shine through.
> Each one is slightly different due to the natural variation in wood grain.
> Dimensions: 208 x 90 x 90mm Weight: 1.2kg

Another page says:

> TO:CA 'wood' LED clock designed by kouji iwasaki in 2002. this 'wooden' LED clock won top prize at the asahikawa international design fair in 2002.

A third page says they're actually made of MDF under the maple veneer, and has a photograph of the back that seems to confirm this, and a fourth page says the manufacturer is "Takumi of Japan".

I think a handheld computer that looks like a block of wood would

be pretty nice too. Something the size of a business card (3.5" x 2", or 89 x 51 mm) but fairly thick (say, 15mm), with veneer on at least one side. The resolution of the display would be limited by the light blurring on the way through the translucent veneer; each spot of light would have a radius on the order of the thickness of the veneer. Veneers are typically 0.8mm but are available as thin as 0.3mm.

If spaced 1.6mm apart, you could get almost 1800 pixels in a rectangular array into the business-card size. You could do a little better with a hexagonal array: if the distance from the center of a regular hexagon to the center of one of its sides is r, then the distance to one of its corners is about 1.15r, which is the same as the length of each side; and its area is 1½ * 1.15r * 2r = 3.45r², which is 14% smaller than a square circumscribed around the same size of circle. In the case of r=0.8mm, you'd have 2.2mm² per pixel instead of 2.56, so you'd get about 2000 pixels. But then you'd have to deal with the hexagonal array in your software.

1800 pixels is enough for about 45 letters in a traditional 5x8 single-bit-deep font, which is pretty cramped; my cheap two-year-old US$30 cellphone has something like 65 letters'worth of space on its display. But it's enough to be useful. It's a lot more than any of the under-US-$10 devices I picked up for the "cheap electronics dissection project" in 2006, and they are useful for some things.

I don't know how easy or hard it is to populate a PC board with 1800-2000 LEDs. I know I wouldn't want to do it by hand.

You could hollow out the middle of a block of wood with just a drill and jigsaw; a keyhole saw or wire saw might work in place of the jigsaw. Cutting all the way through it would be a lot easier than just chiseling out a hollow in one side of the block; then you'd need to put veneer on both sides instead of just one. To add strength and keep it from sounding hollow, you'd probably want to pot the whole interior with epoxy or something.

You could have a couple of finishing nails visible on one end if you wanted to charge it through actual electrical contacts rather than with induction.

Other Everyday Items

You could also embed handheld computers in the following: oyster shells; bricks; pens (I suggested this previously on kragen-tol); ceramic tiles; beanbags, pillows, and stuffed animals (like the Chumby and the Furby).

References

Eager Plastics, aka Eager Polymers, has an "EP4117 General Purpose Polyester Laminating Resin" with a density of 1.11 g/cc.

In April 2002, I posted "magic boxes and secret knocks" to kragen-tol.

The article Using Veneers describes the different kinds of wood veneers available today.

In 2006 I wrote a web page about my "cheap electronics dissection project", where I bought a bunch of cheap electronics and looked inside them.

Topics

The internet is probably not going to collapse for economic reasons

Kragen Javier Sitaker, 2016-09-06 (9 minutes)

Occasionally I run into articles about how the internet is doomed for economic reasons, even today, like this article by John Michael Greer. But they're wrong. Greer's basic argument is that internet services like Facebook are being heavily and unsustainably subsidized by money-losing venture capitalists and that the internet as a whole is heavily and unsustainably subsidized by fossil fuels.

Consumer monthly internet service pays the majority of internet costs

Every time I've mentioned the future of the internet on this blog, I've gotten comments and emails from readers who think that the price of their monthly internet service is a reasonable measure of the cost of the internet as a whole.

It actually is a reasonable measure, because the majority of the cost of operating the internet as a whole is the cost of operating the so-called "last mile" service. The total cost is maybe twice that.

For a useful corrective to this delusion, talk to people who work in data centers.

I do, all the time. They pay substantially less for their bandwidth and their computers than residential internet users do. This should be unsurprising, because they are able to drive bargains with equipment and bandwidth providers that residential users can't, and they can place their computers right next to the internet backbone instead of stringing cables through urban neighborhoods.

Amazon may be the biggest retailer on the planet, for example, and its cash flow has soared in recent years, but its expenses have risen just as fast, and it rarely makes a profit. Many other content provider firms, including fish as big as Twitter, rack up big losses year after year.

Twitter's current operating expenses are around two billion dollars a year, with 302 million active users. That's six dollars per user per year, or 50¢ per user per month. This is very small compared to the cost of the price of monthly residential internet service.

Facebook's operating expenses are around ten billion dollars a year, with which they serve 1.44 billion active monthly users, 936 million active each day. Again, that's about six dollars per user per year.

Wikipedia's operating expenses are only US$58.5 million, but of course that doesn't pay the editors, just the server infrastructure. Because of Wikipedia's privacy policy, they don't know how many people read Wikipedia every day, but comScore says it's about half a million. That means Wikipedia spends about 10¢ per user per year, or about 1¢ per user per month.

Now, it's true that getting some money from the users who benefit from an internet service into the pockets of a company who provides it is a pretty big problem, and one that Greer correctly points out many companies are all too happy to put off solving. But that doesn't imply that the total costs per user are high — as we've seen above, Twitter, Facebook, and Wikipedia (three of Alexa's top 10) are spending 50¢, 50¢, and 1¢ per user per month. Indeed, in all

likelihood, if Twitter and Facebook weren't desperately struggling to get more users, their operating expenses would probably be lower. But Twitter's only been able to put off profitability for so long because their cost per user is so low.

(If we extrapolate these costs down to a small three-person company, with each person costing US$100k per year and equipment costing a similar amount, we end up with US$600k / 50¢ ≈ 1.2 million users as the smallest viable online service these days.)

Once this happens, the companies that dominate the industry have to stay in business the old-fashioned way, by earning a profit, and that means charging as much as the market will bear, monetizing services that are currently free, and cutting service to the lowest level that customers will tolerate. That's business as usual, and it means the end of most of the noncommercial content that gives the internet so much of its current role in popular culture.

I often hear nonsense like this from people who are relatively new to the internet and therefore don't remember what it was like before privatization. Noncommercial "content" has a history on the internet far deeper than what is suggested here.

What we're seeing in practice is that, when you lower the cost of communication sufficiently, commercial "content" like Encarta or Encyclopedia Britannica or Microsoft Windows has a really hard time competing with noncommercial "content" like Wikipedia or FreeBSD (the free software MacOS is built from). The big websites like YouTube, Wikipedia, Facebook, and Twitter are not sources of content; they're just ways for people to talk to each other that happen to be more convenient and efficient, and no more expensive, than running peer-to-peer software on their own computers. The companies that currently "dominate the industry" (if by that we mean Facebook and Twitter and not Comcast and VSNL) are only able to do so because they are successfully able to deliver people to each other.

As long as some people can control the software running on their own computers and get those computers to talk to each other, companies like Facebook will have to compete with "free". (And as one of the commenters on Greer's post points out, BitTorrent already accounts for the majority of internet traffic.)

Non-fossil-fuel energy is abundant

the internet is simply a cheaper and more convenient way of doing things that people were doing long before the first website went live, and a big part of the reason why it's cheaper and more convenient right now is that internet users are being subsidized by the investors and venture capitalists who are funding the internet industry. That's not the only subsidy on which the internet depends, though. Along with the rest of industrial society, it's also subsidized by half a billion years of concentrated solar energy in the form of fossil fuels.

We've already shown above that the degree to which internet users are being subsidized by money-losing investors and venture capitalists is very small, but the energy subsidy question here is interesting, and Greer's position is spectacularly wrong.

It's true that our fossil fuels accumulated over half a billion years, but that does not mean that they are half a billion years of concentrated solar energy; the vast, vast majority of that solar energy was reflected or re-radiated into space and lost forever, not stored away in coal and oil to await humanity.

Current proven coal reserves are 861 billion tonnes, which at 24

GJ/tonne are about 21 zettajoules. Oil and gas reserves are somewhat less. The earth's radius is 6400 km, and it receives about 1000 W/m² of solar energy, which is about 128 petawatts over its circular cross-sectional area.

That means that current proven coal reserves are 45 hours of concentrated solar energy, not half a billion years, not a billion years, not a million years, not a thousand years, not a century, not a year, not even a week. Every two days, we receive from the sun as much energy as all of our currently-known coal reserves.

Historically, it has proven difficult to use solar energy — in part because, as Greer correctly notes, fossil fuels are more concentrated and therefore hard to compete with in an economic sense, but also because of our limited materials and fabrication technology, much as nitrate mined in Chile was a crucial strategic ingredient for explosives until the Haber process was invented.

But using solar energy is no longer particularly difficult, if we measure difficulty in dollars — Juan Cole collated several stories of recent utility-scale solar energy, among which is the news that Nawaz Sharif in Pakistan just inaugurated the hundred-megawatt Quaid-e-Azam Solar Plant in Punjab province built "by China's Tebian Electric Apparatus Stock Co Ltd (TBEA) [in] a year…at a cost of $190 million". That's US$1.90 per watt. Spain is already at 70% carbon-free electricity, mostly nuclear, hydroelectric, and wind.

Speaking of nuclear, the amount of energy available in easily fusible elements in seawater is comparable to the amount available in solar, if burned over a reasonable period of time; thorium is somewhat smaller, while uranium is substantially smaller. Fusion energy, hot-dry-rock geothermal (aka "enhanced geothermal systems"), and solar are each individually capable of getting human civilization to Kardashev Type 1.

An internet is a more energy-efficient way to communicate

Topics

- Pricing (p. 3646) (89 notes)
- Energy (p. 3438) (63 notes)
- Economics (p. 3424) (33 notes)
- The future (p. 3746) (20 notes)
- Networking (p. 3594) (7 notes)

Emacs22 annoyances

Kragen Javier Sitaker, 2007 to 2009 (4 minutes)

So I just upgraded to Emacs 22 in April, despite Debian Etch not supporting it. It solves several of my daily annoyances with Emacs 21: - It recognizes "Password: " as a password prompt, so ssh and sudo get the benefit of me not having to manually type M-x send-invisible. - I can paste Unicode text into it from a web browser, including asymmetrical quotes, real apostrophes, and em dashes, and have it save them to a UTF-8 file without fuss. (Although it still displays the quotes in an obnoxious double-width fashion until the file has been saved and reloaded.) - TRAMP works out of the box. - The documentation is included, unlike in Debian. (There's a licensing dispute over whether the GNU Free Documentation License is free enough to satisfy the Debian Free Software Definition.) - comment-region now asks what comment syntax to use if it doesn't know. - When I run e.g. "darcs" by itself in shell-mode, occasionally Emacs used to take quite a while to display its output usage message, because it was reading it one character at a time. This has been fixed.

I also anticipate joy using MuMaMo, but I haven't actually tried that yet.

There are some changelog/news entries that sounded pretty good:

...if you set `set-mark-command-repeat-pop' to t. I.e. C-u C-SPC C-SPC C-SPC C-SPC ... cycles through the mark ring. Use C-u C-u C-SPC to set the mark immediately after a jump. [Haven't tried this yet.]

```
...M-% typed in isearch mode invokes `query-replace' or
`query-replace-regexp' (depending on search mode) with the current
search string used as the string to replace.  [Haven't tried this
yet.]

You can now customize the use of window fringes.  To control this
for all frames, use M-x fringe-mode or the Show/Hide submenu
of... [so now I can have two 80-column windows on my screen at
once, which is awesome]

A new minor mode `next-error-follow-minor-mode' ... In this mode,
cursor motion in the buffer causes automatic display in another
window of the corresponding matches, compilation errors,
etc. [Haven't tried this.]

The new command `multi-occur' is just like `occur', except it can
search multiple buffers. [Useful. Also I didn't know about
`occur`.]

The grep commands provide highlighting support. Hits are fontified
in green, and hits in binary files in orange.  Grep buffers can be
saved and automatically revisited.  [This is in fact extremely
awesome.]

In addition, when ending or calling a macro with C-x e, the macro
can be repeated immediately by typing just the `e'. [This sounds
```

```
nice, but the F3 and F4 macro keybindings are better.]

The new package longlines.el provides ... "soft word wrap" [like
actual word processors have since the 1970s.  Turns out to be
fantastic.]

SES mode (ses-mode) is a new major mode for creating and editing
spreadsheet files.  [Haven't tried this yet.]

The new package table.el implements editable, WYSIWYG, embedded
`text tables' in Emacs buffers  [Haven't tried this yet.]

The new package flymake.el does on-the-fly syntax checking of
program source files.  [Haven't tried this yet.]

savehist saves minibuffer histories between sessions.  [Haven't
tried this yet.]

isearch in Info uses Info-search and searches through multiple
nodes.  [This is fantastic.]

Atomic change groups: To perform some changes in the current
buffer "atomically" so that they either all succeed or are all
undone, use `atomic-change-group' around the code that makes
changes.  [Sounds like a fantastic idea, but I haven't tried it
either.]
```

So far I've only noticed two new annoyances: one is that it uses its own python-mode that I don't like as well as the one that comes with Python, and the other is that C-x C-f RET no longer reverts the file to the version in the filesystem (assuming the buffer wasn't edited); now you actually have to type the filename.

The stuff in the NEWS file (C-h N) looks pretty innocuous. Nothing is terribly exciting, though.

Topics

- History (p. 3500) (71 notes)
- Emacs (p. 3435) (4 notes)

ISAM designs for Tahoe-LAFS

Kragen Javier Sitaker, 2016-09-07 (2 minutes)

Tahoe-LAFS is, among other things, a content-hash-addressable store for use as a networked filesystem. https://www.tahoe-lafs.org/trac/tahoe-lafs/wiki/FAQ question 13 describes a "medium distributed mutable file" facility for storing mutable files as balanced Merkle trees of 128-kilobyte chunks. There's also a "large distributed mutable file" facility proposed but not yet implemented, which instead stores sequential logs of deltas like Mercurial's revlog file format.

In the Tahoe IR channel I wrote (edited for format):

https://tahoe-lafs.org/pipermail/tahoe-dev/2012-October/007750.html is a fairly different proposal from the "Mercurial revlog" proposal. It seems to me like B-trees might be a better fit for storage that has significant latency per random read, like disk or network filesystems, which I guess is how MDMF works. Has anybody tried running some kind of ISAM implementation on top of MDMF?

It seems like there might be a somewhat higher cost to running some kind of ISAM thing on top of something like MDMF than just journaling your ISAM updates directly onto a write-once block store, since journaling allows you to write just the updated records at first. Postponing the task of copying them together with the unchanged records into a single new block (to restore locality of reference) until less of the unchanged records are unchanged, but I don't have a good estimate for how large or small the cost of the extra abstraction layer (which also serves to implement regular mutable files) would be. A log-structured filesystem scavenger can choose to scavenge the segments with the smallest amount of surviving data, thus optimizing the cost/benefit ratio of segment scavenging.

MDMF could do that too, though! Is there more detailed performance data somewhere? The fact that SSD FTLs have a hard time with random writes, though, makes me think that the cost of hiding nonsequential writes at an MDMF-like low level is probably very significant. So what I'm suggesting is somewhere in between the "balanced Merkle tree" and "Mercurial revlog" approach, since balanced Merkle trees have potentially arbitrarily bad write performance, while Mercurial revlogs have potentially arbitrarily bad read performance.

Topics

- Databases (p. 3400) (20 notes)
- Decentralization (p. 3404) (13 notes)
- Filesystems (p. 3455) (8 notes)
- Content addressable (p. 3389) (8 notes)
- Log-structured merge trees (LSM-trees) (p. 3555) (4 notes)
- Write-once read-many (WORM) memory (p. 3779) (3 notes)
- Merkle DAGs (p. 3573) (2 notes)

Square wave synthesis

Kragen Javier Sitaker, 2014-02-24 (2 minutes)

An integral of a square wave is a spline approximation of a sine wave. The third integral is a cubic-spline approximation, which has very low harmonic distortion. The amplitude of the approximated sine wave is the amplitude of the square wave scaled by the Nth power of the period; for a third-order spline, it's scaled by the cube of the period.

This is interesting because the discrete analog of integral, running sum, is linear, time-invariant, and very cheap indeed to compute (one accumulator and one addition per sample), and sine waves are useful primitives for composing many signals.

In the case of audio synthesis, in particular, I'm thinking you can take some square waves (or envelope-shaped square waves), add them together, and take their running sum a few times, to get a mix of sinusoidal signals.

To be concrete, suppose you're synthesizing 16ksps audio, and you want to be able to cover the audible range down to 20Hz and up to the Nyquist frequency at 8kHz. A 20Hz square wave has a period of 16000/40 = 400 samples on both the top and bottom; this results in amplifying the original square wave by about 400^3 for a third-order spline, or 64 million. (This is not quite correct because of discretization; the actual number is 10.6 to 10.7 million.) This means you need 32-bit integer math for your accumulators (and for the amplitudes of your higher frequencies), but that 32-bit math gives you a dynamic range at 20Hz of 10dB $\log_{10}((2^{31}/10.6M)^2)$ = 46dB, which is quite respectable.

At higher frequencies, you have correspondingly more dynamic range; at Nyquist, you have 31 bits of dynamic range, since the square wave *is* the sine wave, or 187 dB.

So you have square waves, which are more or less cheap to compute (at least as long as you're far from Nyquist and can therefore disregard jitter), and which you sum; and then you integrate them (three times) to get a sum of (cubic) spline approximations of sine waves. You have to scale the amplitudes of the square waves as mentioned above, proportional to the Nth power of the frequency.

Topics

Wang tile addition

Kragen Javier Sitaker, 2017-02-16 (3 minutes)

Suppose you wanted to construct a set of Wang tiles that performed base-10 addition. Like, you would start with a row of interlocking puzzle-piece tiles that said " 3280+834 ", with a flat edge along the top, left, and right, but a ragged edge along the bottom, and all the possible ways to complete a smooth-edged shape would have the correct answer "4114" on the bottom row.

This involves four kinds of top-row tiles: digits, left boundary, right boundary, and "+". (Or 13 kinds, if we count each digit separately.) These can mostly all fit together in an arbitrary order, but the left and right boundary have smooth edges that won't interlock with anything else. And if you put more than one "+" sign in there, you will have created an unsolvable puzzle. (You could use two different kinds of digit tiles, one for the left addend and one for the right, in order to make this impossible.)

The first order of business is to get the corresponding digits lined up, so that you get some kind of representation of "(3+0), (2+8), (8+3), (0+4)". This involves introducing an extra "0" to the "834", and then shifting digits over, one per row, until they reach their final position. Then, once the corresponding digits are lined up, you need to reduce the pairs to sum digits, one by one, and finally once everything is a sum digit, you can finish off the bottom with a nice smooth-edged line.

Introducing extra leading zeroes

Here we're copying the original problem down, row by row; so we have "copy digits" whose tops interlock (only!) with the bottoms of the corresponding digits above them, and similarly a "copy left edge" and "copy right edge" and "copy plus sign". Except that there are 120 kinds of extra "digit insertion tiles" which allow you to, under the right circumstances, shift a number over by one tile to the right. Each of these kinds of digit insertion tiles has a digit printed on it that is determined by its left and bottom edges, which are the same, and a right edge that is the same as its top edge. There are top-edge versions for each of the possible digits that could be above, and also the "+" tile and the right-edge tile.

The "copy down +" and "copy left edge" tiles have a special right edge that allows an "insert leading 0", so you can add an extra-leading-zero tile to shift over either the left or the right operand by one. The "copy down" tiles are not "shift tiles", so you can only use them directly below the thing they're copying, so you can only insert one leading 0 per row.

Shifting corresponding digits left

Topics

Designing an archival virtual machine

Kragen Javier Sitaker, 2016-05-12 (6 minutes)

Software has a very variable lifetime. There are a few programs written in the 1970s that are still in use today, some 40 years later: many Unix utilities (including vi, now nvi), BRL-CAD, SPICE, Maxima, TeX, Smalltalk, and so on. There are many programs from the 1980s that are still in use today, 30 years later: much of X-Windows, MS-DOS, GNU Emacs, GCC, other GNU utilities, LaTeX, parts of Microsoft Windows, and so on.

In many of these cases, little of the original code remains; 30 or 40 years of maintenance have changed it considerably.

In order for our programming efforts to become part of the intellectual heritage of humanity, rather than be forgotten, they apparently need to be continually maintained. blah blah blah.

Urban Müller's BF, inspired by Wouter van Oortmerssen's False, is a virtual machine design with eight instructions that you can implement in half an hour in a page of code. It's probably close to the simplest virtual machine design that you can write real programs for; Linus Åkesson has written a Game of Life for it in a bit under four kilobytes. People have compiled text adventures and fractal renderers to it.

Unfortunately, BF is a terrible machine to program for. It has no subroutine-call mechanism, limited ability to index into memory, and limited arithmetic; straightforward implementations are exponentially inefficient, and even advanced implementations can be inefficient.

I'd like an archival virtual machine meeting the following requirements:

• within an order of magnitude of BF in difficulty to implement — which is to say, less than five hours and ten pages of code;
• within an order of magnitude of native code in performance when implemented in that way;
• within an order of magnitude of native code in programming difficulty, say, with an assembler;
• with high probability that a reimplementation from the specification will be compatible.

A simple reading of #1 suggests that the VM should have as few instructions as possible, and shouldn't have more than about 80 different instructions, but really it depends a great deal on how complex the instructions are. Some instructions can be implemented in a single line of code in the VM, while others might require a page of it.

A simple reading of #2 suggests that you need to keep your interpretation overhead down to no more than about, say, 9 instructions per virtual machine instruction. Unfortunately, this is really difficult! Threaded-code systems like Forths do manage to do it in about 5, at the expense of needing to compile to threaded code first, and of things like type-checking and bounds-checking.

We also need to ensure that the VM's instruction set is sufficiently expressive that you don't need three VM instructions to do what one native-code instruction could do. Forths often suffer from this; it's easy to need to do `OVER OVER -` where a single subtract instruction would suffice on a three-address machine.

The Chifir virtual machine, designed to run an emulator for the Smalltalk-72 virtual machine, has one register and 15 three-address instructions:

- jump (PC ← M[A])
- conditional jump (if M[B] = 0, then PC ← M[A])
- store program counter (M[A] ← PC)
- move (M[A] ← M[B])
- load (M[A] ← M[M[B]])
- store (M[M[B]] ← M[A])
- add (M[A] ← M[B] + M[C])
- sub (M[A] ← M[B] - M[C])
- mul (M[A] ← M[B] × M[C])
- div (M[A] ← M[B] ÷ M[C])
- mod (M[A] ← M[B] % M[C])
- cmp (M[A] ← M[B] < M[C] ? 1 : 0)
- nand (M[A] ← ~(M[B] & M[C]))
- refresh the screen
- block until a character is available from the keyboard and store it in M[A]

Requirement #3, that it not be too much of a pain to program, also exerts pressure in favor of a large, expressive instruction set.

Requirement #4, like #1, exerts substantial pressure on simplicity, but also runs strongly counter to including facilities like division (what do you do on division by zero?), floating-point arithmetic, signed integer arithmetic, and possibly other accesses to memory during the same instruction as a memory write (what order do they happen in?). All of these provide dangerous opportunities for implementations to diverge.

The Ethereum Virtual Machine has an interesting feature that could help to ameliorate some of these tensions: its registers (in its case, on a stack) are 256 bits, 32 bytes. You could imagine a virtual machine with similarly wide registers, but with SIMD instructions, like 3DNow, SSE and NEON; in some cases, not only would this allow a single instruction to do the work of several native instructions, it would allow the programmer to omit writing an explicit loop.

Another way to ameliorate these tensions somewhat is to combine several different operations into a single instruction. For example, if we have a register that always contains 0, another that always contains 1, and a third that always contains -1, then a single three-register A += B*C instruction (known as MAC or sometimes FMA) provides ADD, SUB, MUL, and clear-register as special cases; if the instruction works in complex cases, it is guaranteed to work in simple cases as well. Similarly, A ^= B&C provides AND, XOR, NAND, ANDNOT (BIC), and NOT as special cases.

Topics

A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day

Kragen Javier Sitaker, 2017-06-15 (4 minutes)

In http://canonical.org/~kragen/comida.html I calculated the costs of some minimally adequate diets on 2012-08-16. So now I want to update the cost calculation for at least one of them to current prices.

The "minimal budget for balanced macronutrients", for 2482 kcal per day with 52% from carbohydrates, 18% from protein, and 30% from fat, consisted then of 200 grams per day of soybeans, 5 grams per day of salt, 33 grams per day of sunflower oil, and 430 grams per day of white flour. Coto Digital offers 500 g of soybeans for AR$14.99, 1 kg of Cañuelas 000 white flour for AR$7.72, 1 kg of salt for AR$41.65, 1 kg of medium-fine salt for AR$17.19, and 1.5 ℓ of sunflower oil for AR$49.75.

Food	size	AR$	price/g	g/day	AR$/day	% of cost
Soybeans	500	14.99	0.02998	200	5.996	56.055567
Flour	1000	7.72	7.72e-3	430	3.3196	31.034366
Salt	1000	41.65	0.04165	5	0.20825	1.9468932
Oil	1400	49.75	0.035535714	33	1.1726786	10.963169
Total			0/0		10.696529	100.

#+TBLFM: $4=$3/$2::$6=$4*$5::$7=100*$6/@6$6::@6$6=vsum(@2..@5)

So at supermarket prices, this diet would now cost you AR$10.70 per day. Maybe if you buy in bulk the prices would be lower; and on MercadoLibre, you can buy 30 kg of soybeans for AR$466, which works out to half the price of the supermarket.

However, this is 3¾× as much as it cost when I originally did the calculations, in Argentine pesos. (I don't have a good set of CPI figures to adjust the inflation with, because during the previous administration the integrity of the statistics bureau was destroyed, and they were forced to publish false numbers.) At the time, the dollar sold for AR$6.26, and this diet cost AR$2.86 per day, which works out to US$0.46 per day. Now, the dollar sells for AR$16.00, more or less, so this works out to US$0.67 per day. This represents a substantial loss of buying power in Argentine necessities for the dollar, and the popular feeling is that hours of labor have also lost substantial buying power.

With only three foods (not counting the salt), the proportions of the foods are set by the macronutrient balance requirements I've chosen; there is no wiggle room to reduce the price by changing the proportions slightly. But it might be the case that some other combination of foods is now the optimum — I doubt it, though, because the proportions are almost the same, except that now the oil

makes up a bit more of the cost (11% instead of 6%) and the flour correspondingly less. So probably none of these foodstuffs have increased in price proportional to the market as a whole, and I think the flour is maintained at an artificially low level by government subsidies paid to supermarkets, reducing malnutrition at the cost of encouraging people to eat too much flour.

This implies that a family of four will be malnourished if less than about AR$42.80 per day is spent on their nutritional needs, at supermarket prices; this works out to AR$1302 per month for the four.

For an actually healthy diet, you almost certainly need some fresh vegetables, even though the flour is supplemented with vitamins and minerals to prevent deficiency diseases. If bought retail, this adds substantially but not overwhelmingly to the cost; for example, carrots cost AR$17.90 per kg at Coto, so 100 g of carrots would add an additional AR$1.79.

Topics

Dehydrating processes and other interaction models

Kragen Javier Sitaker, 2018-12-28 (updated 2019-01-01) (36 minutes)

I was thinking about the problem of displaying graphical user interfaces and reacting to clicks and other user events in them, and I think I found an interesting unexplored design space, which seems more and more promising as I think about it.

To talk about this, I need to talk about a concept that I don't have a good name for. I guess I'll call it "interaction models". This is not exactly about user interface paradigms as conventionally understood — e.g., command-line software, graphical user interfaces, hypertext — although it's not entirely orthogonal to them. It's more about the model within which programs using those user-interface paradigms interact with not only the user, but also each other and the rest of the computing system. This document is about a new interaction model which I am exploring.

Background

Most existing programs fit fairly cleanly into one of several existing interaction models: "apps", "REST", "batch processing", "REPL", "notebook", and "open operating system", though of course there are gray areas.

Apps

The now-conventional ("apps") model for this is that, behind the window, there's a running program, typically on a multitasking operating system in which it runs in a "process" (a virtual computer), which has a thread of execution and a stack and a virtual memory space and whatnot; and, at times, it responds to user interface events, modifying its internal state and possibly its external appearance.

This works, and it can be extremely efficient, but it has some drawbacks:

• It's very difficult to return the process to a previous state ("undo"), for example because a user interface action had an effect you didn't like, or because you wanted to explore some other alternatives.
• It's very difficult to update the code of the process, because its internal memory state is entangled with the assignment of addresses to pieces of its code in a way that can be difficult to undo.
• Data tends to get "siloed" in particular programs, reducing users' control over the data and causing them to spend a substantial fraction of their time figuring out how to get data out of one program and into another.
• It uses a lot of memory space and, often, CPU time.
• It can be very difficult to migrate the process from one computer to another, which is a serious problem now that we're all carrying lots of computers around.

REST

Another approach is REST. In REST, the "window" is a

document sent to your user-agent by a server, containing your entire session state and including links to other resources that may be relevant. All applications use the same user-agent and the same resource link namespace. REST has a number of advantages over the apps model:

• The uniform link namespace enables a certain degree of interoperability;
• previous session states can be stored as "bookmarks" and returned to at will, or sent to other people as links;
• upgrading the code on the server is completely seamless in most cases.

However, because the server is physically far away and because the user-agents we are using are unbelievably inefficient, current software is shifting away from the REST model and toward the apps model.

Teletypes, typescripts, and the REPL model

"Apps" are actually somewhat older than graphical user interfaces — in a sense, an "app" is what you get as soon as you load a program onto a general-purpose computer and start it running, even if you don't have an operating system. The machine, in essence, starts to manifest in the physical world the behavior of the abstract machine embodied in the program — it becomes an avatar of the program, though perhaps likening programs to devas or gods is blasphemous. Traditionally, with or without timesharing, once the program starts running, barring the invocation of debugging features such as interrupt keys or single-step switches, your interaction is with the program, not the computer or the operating system — or, at least, that is the desired illusion.

In the app model, you launch a program, which puts your user interface to the machine into a special mode where it behaves differently until it exits — or, if you have a multitasking user interface, until you switch tasks. This is the same way Instagram interacts on your cellphone in 2018, the same way Sketchpad worked in 1962, and the same way Mel Kaye's Blackjack program worked on the LGP-30 in the 1950s.

The LGP-30 was a very simple computer indeed; it contained barely over a hundred vacuum tubes, and it ran far too slowly to produce a useful display on an oscilloscope screen. So the user interface on the LGP-30 was a Friden Flexowriter, a kind of teletype.

A Teletype™ (or, generically, a teletype) is an electric typewriter that could be driven either by the computer or by its operator. (Or, in its original application, by another teletype, perhaps in another city.) This meant that the user interface only needed enough electronic or electromechanical memory for the single byte currently being input or output, rather than the dozens of bytes required for the HP 9100's calculator display, the kilobyte or more required for character-cell terminals, or the many kilobytes to megabytes required for framebuffers. Some form of teletype was the primary form of user interface for BASIC, JOSS, and APL in the 1960s, and then Unix in the 1970s, and it remains with us today in the Unix terminal interface, which even today uses the abbreviation "tty".

On the teletype, many programs could produce their final permanent results just by showing them to the user, perhaps without

even storing them in the very limited memory of the time. For example, if you had a series of calculations you wanted to carry out in LISP, APL, JOSS, or BASIC, you could type the formulas into the computer, one after the other, and on the paper you would have the formulas, each followed by its result, for ease of future reference. This form of interactivity is known today as a REPL, for "read-eval[uate]-print loop", from its name in Lisp, and it remains nearly a sine qua non of new programming languages.

The REPL facility meant that you could write useful interactive programs in LISP or APL that, paradoxically, contained no code for processing user input; they merely provided a vocabulary of procedures for the user to invoke from the REPL, leaving the interaction to the REPL.

The teletype user interface is very limiting — you can't even erase a character once it's on the paper, and it was common to use baud rates as low as 110 baud, or 11 characters per second, because the mechanics couldn't print any faster anyway. But it does have some real advantages over framebuffers and character-cell terminals: you can get a large character repertoire and also boldface and underlining by overprinting multiple characters in the same position, and, most interestingly, the user is left with a complete and indelible record of their interaction, typically on continuous-feed fanfold paper, showing both everything they did and everything the computer typed — a record they could annotate with pencil or pen.

This made it possible to easily produce not only printed textual documents, but also banners, data tables, and data plots.

Batch processing

As an example of app-model glass-teletype interaction, Chapter 6 of the TeXbook (originally published in 1984 and documenting the 1983 version of TeX that supplanted the 1978 version) talks about how to start the TeX compiler and then interact with it once it's started:

log in; and start \TeX. \ (You may have to ask somebody how to do this on your local computer. Usually the operating system prompts you for a command and you type '`tex`' or '`run tex`' or something like that.)

When you're successful, \TeX\ will welcome you with a message such as

```
This is TeX, Version 3.141 (preloaded format=plain 89.7.15)
**
```

The '`**`' is \TeX's way of asking you for an input file name.
% Incidentally, 89.7.15 was Jill's 50th birthday.

…

Now you're ready for Experiment~2: Get \TeX\ going again. This time when the machine says '`**`' you should answer '`story`', since that is the name of the file where your input resides. \ (The file could also be called by its full name '`story.tex`', but \TeX\ automatically supplies the suffix '`.tex`' if no suffix has been specified.)

…(Previous \TeX\ systems required you to start by typing '`\input story`' instead of '`story`', and you can still do that; but most \TeX\ users prefer to put all of their commands into a file instead of typing them online, so \TeX\ now spares them the nuisance of starting out with `\input` each time.)…

You can see that TeX is interacting in the app model. However, it seems that for TeX users, app-mode interaction was often a nuisance that they wanted to minimize. They preferred to spend their time preparing the input file for TeX, then processing it as non-interactively as possible.

This leads us to the "batch-processing" interaction model, which is almost as old as the app model — or perhaps older, if you count running punched paper tape through a Teletype. In batch processing, you start a program, which runs for a while, reading input prepared ahead of time and producing output. Then the program terminates, without having interacted with you at all while it was running. Batch-processing operating systems predate interactive operating systems by a little bit.

Originally, in batch processing, you used a separate, non-computer machine to prepare your input, in the form of a deck of punched cards punched on a keypunch or of a reel of punched paper tape punched on a Teletype. Perhaps this was an incremental process where you gathered punched cards from different places, and even a non-mechanized process where you selected and ordered the punched cards by hand. (The equivalent with paper tape involved, as I understand it, knives and scotch tape.)

Why would you prefer batch processing to interactive processing?

For many years, a primary reason was economic; computers were very expensive compared to employees. A computing job that required one minute of computer time, if done interactively, might require the operator to go through ten minutes of machine setup and five minutes of teardown, during which time the computer spent all its time waiting on the operator, for a total of 16 minutes. If, instead of *operating* the machine manually, you used an *operating system*, you could do it in maybe a minute and a half, without requiring any multitasking. This approach was still occasionally used into the 2000s on supercomputers, because even though they run multitasking operating systems, you can sometimes run two jobs faster one after the other than if you divide the machine's resources between them.

The queue lengths were sometimes long, and programmers tell of needing to wait a day or more for their (printed) job output after submitting their job for processing, back in the 1970s. So if your one-minute job contained five bugs, each of which was hidden by the previous bug, it might take you a week and a half to get it to run in the batch system, while doing it interactively might have taken you 18 minutes. In such cases, the batch system would save computer time — it would consume perhaps 4 minutes in this case rather than 18 — at the expense of human time. So it paid off to be extremely cautious about correctness, and moreover, using a time-shared ("multitasking") system rather than a batch system could boost your productivity by an order of magnitude or more.

Time-shared operating systems got much the same computational efficiencies as batch operating systems — while you're sitting at your terminal thinking about what to do next, the computer is running someone else's job — and much the same human efficiencies as interactive operation without an operating system. But they required a terminal for each concurrent user, and when implemented with a virtual computer per user, they also required much more memory than a batch operating system.

However, even in the time-shared environment that Knuth took for granted in the above quote, batch processing was part of the picture — not just to use less terminals and memory, but also to simplify the programming and make the system easier to use.

The wonderful thing about batch processing — ignored for decades

in favor of mere economic considerations — is its *reproducibility*. A batch program starts up, accesses some input data, produces some output data, and exits. Normally, we have the guarantee that it will produce the same output data if given the same input data (a guarantee Knuth went to extreme lengths to provide in a cross-platform way in TeX, for reasons of preserving access to the scholarly record) and we can modify the input data incrementally to see what changes result.

Reproducibility is extremely valuable for testing and debugging, and it's also very useful for *caching*. A significant fraction of programming, perhaps the majority, consists of "optimizing" — not in the mathematical sense of finding the minimum of a function, but in the sense of making programs do the same thing in less time and/or memory. Caching — storing a previously obtained piece of data so that it's available again when needed — is the most important optimization in the world, and perhaps the majority of optimizing and even software design consists of tweaking what to cache, when, and how.

Reproducible computations can be cached. Irreproducible computations can't. By casting a computation into the batch-processing model, we make it reproducible and therefore cacheable. (Umut Acar's revolutionary "self-adjusting computation" work, as I understand it, is a matter of carrying this principle down to a microscopic level.)

The biggest change in mainstream programming practices over the last 20 years is the near-universal adoption of automated testing, which amounts to testing your software in batch mode, so that the test results are reproducible, rather than interactively.

Batch processing doesn't need "undo", because it just produces output data from input data. If you change the input data, you get new output data, but the old output data doesn't disappear. By incrementally changing input data, we can achieve a sort of "undo and redo" without burdening our program code with extra complexity — make a change to a part of the input that the program accesses early, and it's like undoing everything the program did after that, making the change, and then redoing all the other things in the new context. It's wonderful.

TeX users prefer to keep their interaction with TeX batch-mode because it becomes reproducible and thus gains these flexibilities. And TeX, despite unusually extensive interaction facilities for a compiler, depends on the batch-processing model — it cannot back up to a previous page and change it, for example, or save an editable form of a document so you can edit it again later, both of which would be core functionality for an app-model word-processing system. You can write a document entirely interactively in TeX — that was the Experiment~1 omitted from my TeXbook quote above — but it's impractical.

The Unix shell, and software tools

Shortly after the above quote from the TeXbook, there's a note about command lines:

\danger Incidentally, many systems allow you to invoke \TeX\ by typing a one-liner like '`tex story`' instead of waiting for the '`**`'; similarly, '`tex \relax`' works for Experiment~1, and '`tex &plain story`' loads the plain format before inputting the

`story` file. You might want to try this, to see if it works on your computer, or you might ask somebody if there's a similar shortcut.

This implies that, on some computers where people ran TeX, the only way to specify input files was through the app-model interaction described earlier, which meant that reproducing a TeX run required a human interaction.

This was an affliction suffered not just by TeX but by many systems of that time period, and indeed by many app-model programs nowadays. (For predatory companies such as Facebook, the owner of Instagram, this is an advantage, not a drawback, although internally they have systems for driving the Instagram app in batch mode for testing.)

Unix was a new operating system designed in the 1970s and late 1960s, using teletypes, based on a non-app, indeed anti-app, design — though Unix itself was unredeemably interactive, most of its programs were almost entirely batch-mode. A Unix user in the 1970s might use dozens of programs at different times during a session, but only *interact with* three of them — login, the "shell" (an interpreter for an ad-hoc programming language optimized for launching other programs) and ed, the editor.

In essence, the Unix shell user interface is a REPL for launching batch jobs, and the programs are designed to be composable in useful ways. Unix facilitates this by allowing programs to treat other programs' unfinished outputs as input files, as long as they only access them sequentially from beginning to end, a facility known as "pipes" — the reader process falls asleep until the writer process produces some data, just as with a disk or magnetic tape. The stream of input from the terminal can also be treated as an input file, and normally is, providing a certain degree of interactivity to normally-batch-mode programs and a certain degree of automatability to normally-interactive programs.

The power of the Unix "REPL" must be put in context in its place in the 1970s: it ran on a PDP-11, which might typically have a quarter-megabyte of RAM, of which each process could access no more than 64 KiB at a time. Nevertheless, in a few seconds, you could assemble a command line that strung together four or five processes to use the whole quarter-megabyte of RAM to process a file of a megabyte or more, fairly efficiently.

Some of the designers of Unix used the metaphor of "software tools" to explain the difference between Unix programs and traditional app-model programs: rather than climbing into a bulldozer to interact with it exclusively for a while, you could pick up a shovel in one hand while wrenches and hammers hung from your belt. The software-tools approach became popular with programmers on non-Unix systems as well, but without the Unix shell and pipes, it was much clumsier. Too, the heavy costs of launching new programs on non-Unix systems made the software-tools approach much less efficient than on Unix.

The `make` program added to Unix in the late 1970s provides an automatic caching system for such reproducible batch computations, the paradigmatic example being recompiling a file of source code and relinking an executable when that file has changed. With `make`, the invocations of the compiler are usually automatic. You can also use it to cache a wide variety of staged data processing.

The architecture of the Unix C compiler itself also took advantage of the pipe facility to separate a macro preprocessor, the compiler as such, and the assembler into separate concurrent processes, making C considerably more expressive than other minicomputer programming languages of the 1970s.

After a few years, the "C shell" from Berkeley added filename completion and command-line editing to the Unix "REPL", enabling incremental cut-and-try construction of command lines. As an example, in my shell history on this laptop, I have the following sequence as I incrementally excluded more types of files from my command-line listing:

```
ls
ls | grep -v 'mkv$|vtt$'
ls | egrep -v 'mkv$|vtt$'
ls | egrep -v 'mkv$|vtt$|webm$'
ls | egrep -v 'mkv$|vtt$|webm$|mp4$'
```

The `ls` and `egrep` programs need not be interactive to support this kind of incremental experimentation; they need only take their input from the Unix shell command line rather than interactively.

The C shell also added "job control", which allows a user using a teletype-emulating terminal to switch back and forth between one or more interactive apps and the shell, just as a GUI windowing system does.

The Unix shell, however, suffers some serious weaknesses. It's almost useless for anything involving graphics, although you can do things like this:

```
$ gnuplot -e 'set term dumb size 60,15; plot x**2'

  100 **+----------+-----------+------------+---------+**
   90 +**          +           +            +         **+
   80 +-**                                 x**2 *******-+
   70 +-+ **                                      ** +-+
   60 +-+  **                                   **  +-+
   50 +-+    **                                **    +-+
   40 +-+      **                            **      +-+
   30 +-+        **                        **        +-+
   20 +-+          ***                   ***         +-+
   10 +-+          +  ****     +     ****   +        +-+
    0 +-+----------+------***********-------+---------+-+
     -10          -5           0            5           10
```

Instead of drawing blocky ASCII art, gnuplot can write your plot to a file or open it in a window, but in neither case do you get the alternating list of queries and answers you normally get in a REPL typescript.

The shell is also useless for anything that requires feedback with latency under a second, like adjusting spline control points of a vector graphic.

Finally, the shell typescript session inevitably contains errors and false starts mixed in indiscriminately with the actual useful results, limiting the usefulness of the typescript without further editing. On a teletype this was inevitable, but not on a computer screen.

Notebooks

Like most fashionable programmers, I do a lot of my work nowadays in IPython notebooks (more recently renamed Jupyter notebooks). These are Python REPL transcripts enhanced with a few major features:

• As in the Unix shell or the Python REPL, you can edit previous command lines. Unlike those systems, by default, the edited version replaces the original, and its output also replaces the original output. If you want both versions in your notebook, you can copy and paste into a new "cell".
• The output from the command lines is not restricted to being text. It can include graphics, TeX markup for equations, and data tables, and often does. In fact, it can even support further graphical interaction through callbacks.
• Because the input and the output are distinguished — although they alternate, as in traditional REPL transcripts — it's possible to re-execute some or all of the cells with new input data, or to attempt to reproduce the results after an incremental change. You can think of the contents of each cell as the input to a batch job. Reproducibility is fairly limited, though, for a variety of reasons. First, the IPython session state is an additional, invisible, constantly-changing input to the "batch job". Second, the installed software and filesystem state are also potentially inputs, and they are also invisible and potentially changing.

You can also invoke other programming languages (including the Unix shell), include comments in Markdown with LaTeX equations, and a few other things, but those are not the most important advantages over the bare Unix shell in a terminal.

That is, the notebook is an alternative model for interacting with and integrating programs, different from apps, REST, batch processing, and teletype REPLs including the Unix shell.

The open operating system model

Before proceeding to talk about the new interaction model I'm exploring, I want to point out a few significant systems that don't fit into the above interaction models: Smalltalk, Emacs, Cedar, and Oberon. In these systems, many separately developed "programs" are loaded into the same memory space and can interact with one another using function calls, and the user can interact with them by invoking functions from any of them at any time, and can also load more programs.

In Smalltalk, the system developed at Xerox PARC in the 1970s in which object-orientation was invented, not only each window on the screen but each letter is an object, which is to say, a tiny virtual computer with its own code and state, and each window's code decides how to display it and how to react to events such as clicks. The difference from the apps model is that there's no separation between the windows — it's trivial to embed one "program" in another, call functions of one "program" from another, pass complex data objects from one "program" to another, and so on — like Windows OLE, but simple.

Cedar is a system developed at Xerox PARC in the late 1970s and early 1980s as a sort of improved alternative to Smalltalk and Interlisp,

but with strong static typing and separate compilation. Cedar is not as well known as Smalltalk, in part because the software was never published, but its successor language Oak is unfortunately extremely popular.

After spending a couple of sabbatical years at Xerox PARC working on Cedar and its predecessor Mesa, Wirth went back to Switzerland and, with Gutknecht and others, wrote his own version of it from scratch, starting in 1988, based on his language Modula-3; this system is called Oberon, and it is free software.

XXX Inferno?

Emacs also uses this interaction model. Emacs Lisp is not object-oriented, but it has the notions of "major modes", "minor modes", "keymaps", and "buffer-local variables", which permit different windows to respond differently to interaction; and there are a number of popular programs written in Emacs Lisp, including Org-mode, Gnus, Magit, Eshell, Ediff, and Hyperbole, in addition to the basic IDE functionality Emacs was written for. Although Emacs has been suffering substantially in popularity in recent years, some of these programs are, or once were, best-in-class applications.

Lampson and Sproull published a 1979 paper in which they called a version of this system architecture an "open operating system" or "open system", a term which has been reused for other purposes; they emphasized other aspects.

Mummified embedded apps

With that background out of the way, I've been hacking on Yeso, a simple framebuffer graphics input-output library, with the purpose of making it easy and direct to write graphical programs in any language that run pixel-identically on a variety of platforms. So far I only have X11 and Linux framebuffer console backends, but one of the next things I want to write for it is Wercam, a display server that uses Yeso itself, rather like Xnest, 8½, or Rio — it will allow Yeso apps to connect to it using a simple protocol and draw windows on its screen, and route events to them as appropriate.

This led me to ask: what kind of window management should it use? Should it use overlapping windows, like X, or alternate between giving different apps full control of the screen, like Android and IOS normally do?

And, of course, I'd been using notebooks for years, and Perry Lorier told me about a notebook-style shell he's writing using curses. Unlike Jupyter/IPython, in which only one cell can be evaluating at a time (except that they can respond to callbacks), in his notebook-shell, each process gets its own pseudo-terminal, and it can continue running freely while you're interacting with other processes.

So it occurred to me that maybe the window manager should be such a notebook-shell. If you start a program and it merely produces textual output and takes textual input, that would appear underneath your command, as per normal; but if a program opens a Yeso window, the window would be embedded in the notebook there, by default. Perry's notebook-shell takes a program full-screen when it issues a clear-screen escape sequence; perhaps some similar action would be useful.

The Yeso calling interface is in rapid flux, changing about once a

day for the last three weeks, but putting a colored rectangle on the screen can be done in C something like this:

```
#include <yeso.h>

int main()
{
  yeso w = yb_open("hola, mundo!", 64, 64, "");
  yi_fill(yb_framebuffer(w), 0xcc99cc);
  yb_flip(w);
  for (;;) yb_wait(w, 0);
}
```

On the Linux framebuffer console backend, that last line is somewhat unnecessary; it just makes the program hang until you kill it. On the X11 backend, it's kind of mandatory, because otherwise the window closes immediately as the program exits; it probably won't even be on the screen long enough for a screen refresh.

But preserving output from programs that are no longer running is sort of the sine qua non of REPLs and notebook interfaces. Maybe if Wercam uses a notebook interface for its window management, it should preserve the graphical output of the program, just as IPython preserves the graphical output of matplotlib plot commands that have plotted data. (Unless the user deletes them.) Wercam could actually do better than just save the last frame of output — because Yeso sends a sequence of full frames to the display, Wercam could record all the frames (up to some limit, by default) and allow you to fast-forward and rewind through the impromptu screencast.

Horcruxes or continuations

Having graphical output there in the notebook without a running process behind every scatterplot seems useful. But what if you want to zoom in on one of those plots later? How can you supply event-handling callbacks without a running IPython kernel or other process sitting there waiting to answer them?

Well, in the early days of the Web, clicking a link might launch a CGI program to generate the response document. Wercam could do something similar. What if the program left behind at its exit not only an image but also a sort of last will and testament, or perhaps a horcrux, giving a command to run to handle any new events, and perhaps a set of event types to ignore, such as mouse movements?

(Of course, for security reasons, that command should run only with the authority of the original program.)

At one point it would have been absurd to spawn a new Unix process for each mouse movement — it just took far too long. But my forkovh test on my laptop takes about 130 µs to fork a child process, written in C, which immediately dies, and then reap it. And httpdito takes about 30–50 µs to accept a connection, fork a child process, and then reap it. (The children handle HTTP requests, but that mostly happens on other cores.) Httpdito is faster probably because it's not linked with the C library, so its memory map is much smaller. Linking forkovh with glibc boosts its latency to 670 µs.

Exec is significantly more expensive, although possibly a better operating system would help. Forking, executing, and waiting for a hello-world C program takes about 3 ms with glibc on my laptop

(whether statically or dynamically linked), or 2 ms with dietlibc. On the Linux framebuffer, the Yeso PNG viewer takes 4 ms on small images. (Pentium N3700 1.6GHz at probably 500 MHz, Linux debian 4.4.0-21-generic #37-Ubuntu.)

These latency numbers are not insignificant, but on a 60fps display like the one on my laptop, 2 milliseconds out of a 16.7 ms frame is not immediately fatal. I mean, USB mice are normally polled every 10 milliseconds, and 1 millisecond is the shortest polling interval USB supports.

But suppose respawning the program when there's an input event, or maybe even after a specified timeout, is fast enough at the kernel level. Now the program needs to know the "session state" — if it's redrawing a scatterplot in response to a zoom, for example, it needs to know the data points and plot colors. If it's an interpreter, it needs the parsed program. If it's a PDF viewer that renders nearby pages in the background to avoid delays, it needs access to the prerendered cache. And it needs all of this without having to redo precious milliseconds of deserialization every time.

It turns out that arrays of binary integers or floating-point numbers, written to files in native byte order, and memory-mapped, works really well for this kind of thing. On my laptop, mmap() on an already-open file takes about 4–6 µs. For more elaborately structured data, something like FlatBuffers, CapnProto, or SBE enables you to get similarly zippy speeds without having to pay a startup deserialization tax.

So perhaps the "session state" or "horcrux" in the notebook cell should be a filesystem directory with the crucial state saved in files that the program knows to open and map into its memory when restarted; essentially they are a mummified version of the live program's non-ephemeral memory state, ready for instant rehydration and return to life. This is pretty similar to how Android saves the state of an Activity (an app, in Androidese) in a "bundle" in onSaveInstanceState and then passes that bundle to onCreate when it restarts the Activity; but Android "flattens" whatever you put into the bundle into a "parcel", which is extra overhead that should probably be avoided in this context.

links to programs rehydrating with new versions of code declaring which UI events to handle delegating to other programs file managers publishing outputs in formats usable by other programs Launching transactions that may fail editing text files reducers, replay, and distribution

Topics

- Reproducibility (p. 3682) (3 notes)
- The LGP-30 computer (p. 3547) (3 notes)
- Jupyter (p. 3535) (3 notes)
- Teletypes

Kerr snow display

Kragen Javier Sitaker, 2019-11-12 (3 minutes)

Snow, like titanium dioxide paint, is brilliant white because it's full of randomly oriented interfaces between transparent media of different refractive indices. This causes entering light rays to execute a low-loss random walk which ultimately comes back out of one side or the other of the metamaterial.

Kerr cells are, as I understand it, normally used to slightly retard the phase of light beams passing through a medium with a strong Kerr effect by applying a few thousand volts across a thin layer of it, thus slightly altering its refractive index.

A much more visible effect of the Kerr effect would result if the Kerr medium were mixed with transparent particles of a different phase, no Kerr effect, and a slightly lower refractive index. By matching the indices of the two materials, the cell would vary between transparency and translucency; when the Kerr effect caused their indices to match perfectly, the phase boundaries would cease to refract or cause total internal reflection, becoming perfectly invisible. This amounts to a sunlight-readable display, like a super-fast high-contrast reflective LCD.

Ideally the two media would have the same permittivity to prevent the electrical field from varying according to the local concentration of Kerr medium, but I don't know if that's possible.

In a variant of this approach, instead of particles, the other transparent medium occupies a triangular section of the Kerr cell, while the Kerr medium occupies a different triangular section, and light travels through the Kerr cell lengthwise, entering and exiting at the edge rather than parallel to the electric field. A time-varying electrical field will produce a time-varying refraction angle where the light beam strikes the oblique interface between the two materials, thus deflecting the beam through varying angles, perhaps including total internal reflection (at which point the angle becomes insensitive to further voltage changes). Possibly many different "pixels" can be obtained, each deflecting a different part of the same beam through an independent angle, with bandwidths up into the tens of MHz or more. In this case there is no particular requirement for their indices to be very close. If feasible, this is a better solution to the problem described in Lenticular deflector (p. 2612). Heck, this is probably a product that already exists.

This device as described can also provide amplification; like a MOSFET, its switching action is voltage-controlled and nondissipative, so it can shunt around much larger amounts of power than those needed to control it.

Topics

- Physics (p. 3632) (119 notes)
- Optics (p. 3609) (34 notes)
- Light deflection (p. 3549) (2 notes)

Term rewriting

Kragen Javier Sitaker, 2017-07-19 (3 minutes)

Typical term-rewriting patterns are of the form

```
x(?y, z) -> foo(?y, ?y)
```

where the "?" identifies the `y` as a variable (or, we might say, a metavariable). This poses a certain amount of difficulty for metacircularity, such as a program to determine whether a set of rewrite rules is confluent (normalizing); you might want to look for the literal pattern `?y` or especially `??y` for any `?y` in some rewrite rule you are trying to manipulate.

Prolog, where all this originally came from, is even trickier along this axis, since it spells `foo` and `?foo` as `foo` and `Foo`, respectively.

One possible solution to this problem is to declare the metavariables out-of-line:

```
(rewrite (y) (x y z) (foo y y))
```

or

```
metavariable y;
x(y, z) -> foo(y, y)
```

Another aspect of term-rewriting languages is that they don't accommodate extensible data structures very easily. Suppose you have a bunch of rules of the following form:

```
?x + constant(0) -> ?x
```

Now, you decide that constants should additionally include the line and column number where they were found, thus improving your debugging information. This forces you to modify the above rule as follows:

```
?x + constant(0, ?_) -> ?x
```

And you must do the same thing for all similar rules.

Suppose instead that the nodes of your graph instead have arbitrary sets of named properties, perhaps namespaced to avoid unfortunate property collisions, and with the proviso that extra properties not mentioned never cause a match failure. Then we can write the above instead as follows:

```
{addend1: ?x, addend2: {constant_value: 0}} -> ?x
```

This version will survive such an augmentation without modification.

This is the same data model as Binate, and suggests that you should be able to write such rewrite systems as Binate relations, making them point-free (and solving the above problem of metacircularity in a different way): the transitive closure of some rewrite relation,

perhaps. That is, the above algebraic rewrite rule would be part of a relation called something like `algebraically_simplifies_to`, so we could ask whether

```
(4 + 0) algebraically_simplifies_to 4.
```

or indeed what a given expression simplifies to. The `algebraically_simplifies_to` arc or arcs from that expression would be in some sense on equal standing with the `addend1` node from it to 4. We can take the transitive closure of that relation and subtract a version of its own converse from it to exclude results that can be algebraically simplified further.

I am not sure if this is in practice a feasible thing to do with Binate, or what would be needed to make it feasible, or whether it would be practical, but it is a very appealing simplification and unification!

It also avoids the ugly ad-hoc extension to term-rewriting pattern-matching in Leler's Bertrand (p.53 of his book) where `aNumber` matches any number, etc.

I think one big problem here is that in Binate, equality is (I think) necessarily defined by identity rather than content: two point objects with the same `x` and `y` properties are not equal, because one of them might have some other property (such as `z`) which is not supposed to affect the behavior of queries that don't mention that property.

Topics

- Programming (p. 3658) (286 notes)
- Programming languages (p. 3656) (47 notes)
- Binate (p. 3343) (3 notes)
- Tree rewriting (p. 3757) (2 notes)

Comparison of the PCO-1810 and PCO-1881 plastic bottlecap standards

Kragen Javier Sitaker, 2014-05-25 (updated 2016-07-27) (2 minutes)

Plastic bottlecaps are mostly compatible with each other because their threads conform to a standard called PCO 1810, or occasionally PCO 28, which is being gradually replaced by a newer standard called PCO 1881. These threads are formed into the bottleneck (called a "finish") before the shape of the rest of the bottle is formed; companies like Jesper PET Preform will sell you the bottle "preform", with the thread formed but the rest of the bottle uninflated; then it's up to you to blow-mold the bottle into the shape you want.

The bottle caps (called "closures") are sold separately, and they're usually made of polypropylene ("PP").

PCO 1881 has a slightly shorter neck (17mm instead of 21mm) and was finalized in 2009 by the International Society of Beverage Technologists. It apparently implies no loss in performance compared to the PCO-1810 standard, despite the steeper thread pitch and the threads going around only twice instead of the three times you see on PCO-1810 bottles.

Because the bottle neck and cap are so much thicker than the rest of the plastic bottle, they account for a substantial fraction of the material use in the bottle, so the 1.4 gram reduction due to the 4mm-shorter PCO-1881 standard amounts to 27% of the total weight of the PET bottle, not counting the cap.

ISBT has put the PCO-1810 standard and the PCO-1881 standard engineering drawings online along with their other standards, and "ippe" posted an STL file on Thingiverse that prints to a working PCO-1810 bottle closure at least in PLA on the Prusa Mendel I have access to. (You may want to turn that file upside down for printing if you want it to seal, though; the top was too long a horizontal span to cohere properly.)

In addition to the preforms, you can buy entire preblown bottles if you don't have blow molding ("pressure molding") equipment; a 43-gram 1ℓ bottle costs around €0.36–€0.69 without the cap.

Topics

Supervisor children for fault-tolerant Unix command-line programs

Kragen Javier Sitaker, 2019-01-04 (3 minutes)

It's straightforward for a Unix process to spawn a child and then clean up when the child exits — it can use the wait(2) system call to sleep until the child is done. For things like changing the video mode, turning off cursor display and character echo in a Unix terminal, turning on power to a heating element, and so on, this is an improvement over the usual approach of doing the cleanup in the same process, because it happens even if the supervised process crashes, for example because of an OOM kill or a segfault. This requires that the supervisor process be less likely to crash, but that is often easy to arrange; an effective supervisor for particular scenarios can be written in ten or twenty lines of C.

This is such an effective way to handle errors that systems designed for high-reliability applications, such as Erlang, often use it as their only way to handle errors.

However, it's a problem for the adoptability of a C library if using it requires its users to launch their programs in a special way. So another possible approach is to spawn a supervisor *child* which waits for the parent to die. Unix doesn't have a general way to wait for non-child processes to die, but this can be taken care of easily with a pipe from the supervised parent to the supervisor child — the parent can trigger a cleanup either by writing a byte to the child or by exiting, including by way of an OOM kill.

The supervisor child is not entirely transparent to the application, for two reasons.

First, if you have no children, wait(2) or waitpid(2) with a -1 PID will return immediately with ECHILD; the supervisor child would convert this into a deadlock, as each of the child and the parent are waiting for the other to exit. This can be fixed with the double-fork trick usually used to spawn daemons.

Second, and more seriously, POSIX threads are pretty aggressively incompatible with fork(); it's unsafe, at least in theory, to call things like ioctl() in the forked child if the parent is multithreaded, presumably because ioctl() might try to acquire a lock that was held by another thread at fork time, a thread which doesn't exist in the child and which therefore can't unlock its lock. To work around this — and also perhaps to reduce memory consumption and the likelihood of OOM kills — the supervisor child could exec() a special-purpose supervisor executable.

Topics

• Programming (p. 3658) (286 notes)
• Systems architecture (p. 3691) (48 notes)

- Fault tolerance

Graph construction

Kragen Javier Sitaker, 2016-09-08 (updated 2017-07-19) (23 minutes)

(See also Circuit notation (p. 1161) and A stack of stacks for simple modular electronics (p. 1779).)

I think I have a new approach to concisely and responsively describing directed graph structures that is useful in many different creative endeavors that can be undertaken with computers. I don't yet have a working implementation; this article is only an exploration.

Here's an overview of different possible notations for edge-labeled directed graphs (and related structures), including some new applications.

As an example application, in one variant of the notation, this is a schematic for an energy-harvesting power supply circuit taken from a paper on the subject. Here , represents parallel connection, concatenation represents serial connection, and <> represents reversal, among other things:

```
{{<d> {{ac i, c}}=q <d>, {<d> q <d>, c}} s {<d>, l {c, {<v>, r} r}}}
```

However, this approach is not limited to circuit design. All of the following can be conveniently represented as edge-labeled directed graphs:

- regular expressions
- finite state machines
- bubble-and-arrow diagrams (although the bubbles are more commonly labeled than the edges)
- database queries
- database contents
- dataflow pipelines (or more generally dataflow graphs)
- circuit schematics
- control-flow graphs in programs
- networks of constraints

These techniques should be applicable to user interaction and generative programming in all of these areas.

Netlists

You can write these things as netlists or 3-tuples; an abbreviated example of a bubble-and-arrow diagram from the dot man page:

```
digraph test123 {
        a -> z [label="hi", weight=100];
        x -> z [label="multi-line\nlabel"];
        b -> x;
}
```

Or, here's a circuit netlist in Spice:

```
Multiple dc sources
v1 1 0 dc 24
v2 3 0 dc 15
```

```
r1 1 2 10k
r2 2 3 8.1k
r3 2 0 4.7k
.end
```

This requires naming every node; the nodes are named `a`, `b`, `x`, and `z` in the dot example, and 0, 1, 2, and 3 in the Spice example.

Here's a dataflow example by zahorjan@cs.washington.edu in MIPS assembly, which is at the same level of abstraction, although it reuses the "dataflow node labels" `$t0` and `$t1` — each time they are the first operand of an instruction, their previous value is overwritten:

```
srl   $t0, $t0, 1      # i/2
addi  $t1, $gp, 28     # &A[0]
sll   $t0, $t0, 2      # turn i/2 into a byte offset (*4)
add   $t1, $t1, $t0    # &A[i/2]
```

If we were to rewrite that with SSA-style subscripts (what Ada Lovelace use prefixed superscripts for) we get:

```
srl   $t0₁, $t0₀, 1
addi  $t1₀, $gp₀, 28
sll   $t0₂, $t0₁, 2
add   $t1₁, $t1₀, $t0₂
```

There are better alternatives, though.

Infix

The traditional solution for the arithmetic dataflow case, of course, is to divide the computation into unary and binary operations and write them in infix notation; the above MIPS assembly then renders as follows:

```
($t0₀ >>> 1 << 2) + ($gp₀ + 28)
```

This is considerably easier to follow, bringing related things closer together and eliminating most of the accidental names, ending up about three times shorter.

By itself, it only covers a very restricted range of cases: tree-shaped digraphs, in which, if interpreted as dataflow graphs, each value is used only once, producing a single output. But it isn't immediately obvious how to apply that to get more general kinds of graphs.

Kleene's Theorem

Stephen Kleene (whose name I mispronounced for 21 years as [klin], but TIL it's actually [kleɪni]) showed that any regular language, and thus the possible paths through a finite state automaton, can be generated by a regular expression. As I understand it, a regular expression on some alphabet Γ is recursively defined as the smallest language L such that

```
L = Γ ∪ ({"concat", "alt"} × L × L) ∪ (L × {"*"}) ∪ {∅, ε}
```

Hmm, that definition is shitty, I'll try again. A regular expression `Γre` over Γ is defined as the sum type

```
Γ re = Lit of Γ
     | Concat of (Γ re × Γ re)
     | Alt of (Γ re × Γ re)
     | Closure of Γ re
     | Impossible
     | Emptystring
```

The language generated by a regular expression is defined as follows:

```
lang(Lit(x)) = {x}
lang(Concat(a, b)) = {wa || wb for wa in lang(a) for b in lang(b)}
lang(Alt(a, b)) = lang(a) ∪ lang(b)
lang(Closure(a)) = {ε} ∪ lang(Concat(a, Closure(a)))
lang(Impossible) = ∅
lang(Emptystring) = {ε}
```

Here || is string concatenation. (Or word concatenation, as combinatorists apparently like to say. And who can argue? A C program as little resembles a piece of twine as it does an entry in the dictionary.)

So the nifty thing here is that this gives you a way to generate all possible finite-state-machine languages by building up a finite-state machine from simpler single-entry single-exit finite-state machines, which is to say, building up a directed graph from simpler two-terminal directed graphs. You can't express every possible graph in this way; in particular, there are a set of "irreducible control flow structures" which, if present, require duplication of nodes. (McCabe's 1976 paper "A Complexity Measure" explains in more detail in the context of program control flow.)

Traditionally in regular expressions we write Concat, Alt, and Closure as mere juxtaposition, infix |, and postfix *, respectively, although, in the context of Kleene algebras, alternation is traditionally written with infix + instead.

And this is somewhat more expressive; it can express any series-parallel circuit. This includes trees as a subset. If we use the traditional |* notation, we can express the above MIPS-assembly arithmetic graph as follows:

XXX try , instead of |?

```
(($t0₀ | 1) >>> | 2) << | ($gp₀ | 28) +) +
```

Here we are supposing that constants and variables obliterate whatever data was present on their branch of the dataflow digraph, that rejoining dataflows creates a node where both pieces of data are available (with the left and right branches distinguished!), and that the operators then reduce the two pieces of data available at their node to 1.

But now we can also represent the SPICE circuit I gave above, for example. One possible representation is as follows:

```
(v⁻¹[dc 24] r[10k] (r[8.1k] v[dc 15] | r[4.7k]))*
```

This contains all the useful information in the original netlist, but it is both much easier to write and, I would argue, even much easier to

read; it's easy to see what is in series and what is in parallel. Interpreted as a regular expression, it yields all of the possible traversals through the circuit in a certain direction (which could be the direction current is flowing, but happens not to be).

It's common to be able to express the majority of an electrical circuit in terms of such series-parallel circuits, with the occasional deviation from series-parallel constructions; the same is true for control flow in programs.

Postfix

The arithmetic expression above may look suspicious:

```
(($t0₀ | 1) >>> | 2) << | ($gp₀ | 28) +) +
```

It happens to be exactly equivalent to how you'd express this expression in postfix (RPN), but with stack relationships explicitly called out — every time some value produced by `b` is on top of some value produced by `a` on the stack, we have written `(a|b)` rather than just `a b`. If we adopt the more usual convention, this expression reduces to the following:

```
$t0₀ 1 >>> 2 << $gp₀ 28 + +
```

Can we apply this to the circuit diagram as well? It's simple:

```
{ 24 dc v⁻¹ 10 k r { 8.1 k r 15 dc v | 4.7 k r } }
```

Consider the expression as an RPN program to construct a circuit diagram, with a stack consisting of node identifiers and numbers. It begins with a single node identifier on the stack, and it uses the following operations:

- `{` duplicates the top item on the stack
- `dc` pushes some number on the stack used to indicate DC
- `v⁻¹` pops a voltage type and parameters from the stack, allocates a new node, connects the positive side of a newly created voltage source to the new node, connects the negative side to the node on the stack, which it pops, and pushes the new node on the stack
- `k` multiplies the number on top of the stack by 1000
- `r` is analogous to `v⁻¹` but creates a resistor instead
- `v` is exactly the same as `v⁻¹` but connects its items in the opposite order.
- `|` exchanges the top two items on the stack
- `}` shorts together the two nodes on top of the stack, causing them to become equivalent

It turns out that, in this context, this semantics for `{ }` makes them adequate both for Kleene closure `*` and to enclose a binary alternation `|` in the context of circuit schematic capture, because a wire forward from the beginning of a part of a circuit to its end is the same as a wire backward from its end to its beginning. In other contexts, like regular expressions, these two are not equivalent.

`v⁻¹` is a bit suspiciously ad-hoc; really we might prefer this operation of flipping a bit of circuitry around before connecting it to be an orthogonal operation, rather than separately defining `v⁻¹`, `d⁻¹`,

`electrolytic_capacitor⁻¹`, and so on. You could instead define `v` to allocate two fresh nodes and push them both on the stack, then use a separate `forward` or `reverse` operation to short two of them together and remove them from the stack. Alternatively, you could have < and > operations that enclose a piece of circuitry to be reversed — the first allocating a fresh node and pushing two references to it onto the stack, the second performing the equivalent of `rot }`. But really this is the converse relationship on binary relations, about which more later.

In general, I find postfix less readable and more error-prone than infix, so I'm not sure it's really the right thing, but it has a few advantages which might be relevant in this context:

• Its semantics are ruthlessly simple and thus easy to implement correctly.
• It's very clear to see how to extend it to define new parametric components or macros — pseudo-arcs that are really an entire subgraph — simply by attaching a name to a sequence of graph-construction operations and invoking them as if by copy-paste later on.
• It smoothly handles three-terminal and multiple-output devices; three-terminal devices can, for example, consume two items from the stack and produce a third (ideal for things like logic gates) or push fresh nodes for all of their terminals onto the stack. If you can define new component macros, you could define a bridge-rectifier macro that consumes two AC terminals and returns two DC terminals.
• For interactive graph construction and editing, not only does it require fewer UI actions than other means of schematic capture, it can also provide immediate simulation feedback while the circuit is being constructed, because almost everything is already connected when it's constructed.

I'm not sure that "edge-labeled directed graphs" are quite the right abstraction for electrical circuits. Yes, a battery or a diode is a directed relationship between two wire nodes/nets, so representing it as a labeled directed edge in a graph makes some sense. But a bipolar transistor is a relationship between three wire nodes/nets, each of which has a unique relationship to it, and a microcontroller is in some sense a relationship between dozens of nodes. These are more like hypergraphs, but the existing work on *directed* hypergraphs is pretty slim.

This mirrors the state of relational databases and logic programming, where most of the work is with some arbitrary number of finitary (N-ary) relations, rather than binary relations.

A bipartite directed graph, where wire nodes/nets are one part and components are the other part, would be an adequate model. In dataflow, the parts would instead be values and operations.

Despite all this stuff and despite being quite compact, postfix, even augmented with the { | } operations above, can't construct arbitrary edge-labeled directed graphs, just directed graphs whose traversals are equivalent to traversals of any given directed graph, and that only at the cost of a potentially exponential expansion in graph size. In particular, they can't construct any nonplanar graphs, and not even all planar graphs.

Variables/Labels

The usual solution for this in dataflow is to use (possibly immutable) named variables, and in control flow to use labels and `goto` — essentially escaping back to netlists, where connection is indicated by coincidence of names rather than proximity. Labels or variables are "wormholes" in the otherwise planar and recursive graph construction; they can connect anywhere. Usually you don't need very many of them.

The above discussion of regular expressions leads us to speculate on whether we should be labeling nodes, as is normally almost universal, or entire subgraphs. Consider the following SPICE model, from the same textbook page as the example I gave earlier:

```
fullwave bridge rectifier
v1 1 0 sin(0 15 60 0 0)
rload 1 0 10k
d1 1 2 mod1
d2 0 2 mod1
d3 3 1 mod1
d4 3 0 mod1
.model mod1 d
.tran .5m 25m
.plot tran v(1,0) v(2,3)
.end
```

Like bridge circuits in general, this is not a series-parallel circuit; you need a label. You can represent most of it as a series-parallel circuit; let's use the postfix form, and suppose we have the direction-reversing brackets < and > suggested earlier:

```
{ 15 60 ac v  { d < d > | < d > d } }
```

This expresses the entire circuit above except for the load resistance. If we add the new operation }= to define a label for a subgraph, defining the word following it to hook up the two nodes on the stack as if they were an arc, then we can write it as follows:

```
{ 10 k r }= rload  { 15 60 ac v  { d rload d | < d rload d > } }
```

This defines exactly the same circuit as the SPICE model in a third less text, although it doesn't specify the simulation parameters. Each time `rload` is invoked, it connects its environment to the *same* component, unlike things like `d`, which generate a new component each time they're invoked.

Alternatively, and more traditionally, we could define }= to simply define a name for the node on top of the stack, removing it from the stack; when invoked, the name shorts that saved node to whatever is on top of the stack at the time. Then we can define the circuit as follows:

```
{ 15 60 ac v  { d { 10 k r }= rload < d > | < d > rload d } }
```

Explicit series connection

Another alternative, which I mention here more for completeness rather than because I think it's necessarily better than the others

mentioned, is to make the series connection explicit rather than implicit. In an infix syntax, for example, using the hyphen for the series connection, perhaps R-C is a series connection of a resistor and a capacitor, while R|C is a parallel connection. Then we can do things like L-(C|L-(C|R)). Here, as in the notation at the beginning, the values being combined are not circuit nodes or nets but rather subcircuits with a beginning and end.

We can, of course, use a stack notation for this as well; for example, instead of L-(C|L-(C|R)), we could write LCLCR|-|-. A stack-notation expression can represent any number of subcircuits rather than just one, because it can end with any number of items on the stack; similarly, it can represent a transformation of a circuit or set of circuits, rather than just a circuit. (Hmm, this is a digression; perhaps it belongs elsewhere.)

Without the explicit series combination operator, we can get the same amount of power in a stack-language context by having an explicit "push new disconnected node" operator. I think.

Other arities of edge creation

You could also create an edge between the top two nodes on the stack, consuming both and returning neither. In this model, you must explicitly create each node, then explicitly duplicate any node that will be connected to more than one edge; in a sense, the expected degree of a node is 1. (So to get three resistors in series, you would say NODE DUP NODE R NODE OVER R NODE OVER R. I think.) The other approaches I've been describing all have, in some sense, an expected degree of 2 for each node — the operation of adding each new edge to the graph consumes a node and produces a node, which then can be merged with some other node in the (implicitly exceptional) case where that is desired.

You could also imagine leaving all the nodes on the stack, not consuming any of them, so NODE NODE R R R would give you three resistors in parallel.

What's the best default node degree?

Eyeballing a couple of small schematics, I have:

- 8 nets, 8 2-terminal components, 2 3-terminal components, average degree of `(/ (+ (* 8 2) (* 2 3)) 8.0)` = 2.75.
- 12 nets, 8 2-terminal components, 6 3-terminal components, average degree of `(/ (+ (* 8 2) (* 6 3)) 12.0)` = $2.8\bar{3}$.
- 4 nets, 4 2-terminal components, 1 3-terminal component, average degree of `(/ (+ (* 4 2) (* 1 3)) 4.0)` = 2.75.
- 9 nets, 2 2-terminal components, 1 3-terminal component, 1 5-terminal component, average degree of `(/ (+ (* 2 2) (* 1 3) (* 1 5)) 9.0)` = $1.\bar{3}$.
- 12 nets, 12 2-terminal components, 5 3-terminal components, average degree of `(/ (+ (* 12 2) (* 5 3)) 12.0)` = 3.25.

In some sense, this suggests that the "average" degree of a circuit net (i.e. wire) is somewhere around 3.

However, that's including the power rails and ground as nets, which commonly have lots of connections. For example, in the last circuit above (Figure 12.67 from Horowitz & Hill 3ed., a simple laser drive circuit), ground has 7 connections and the positive power rail has 3. If we discount those, the average goes down to 2.416. The first

circuit, figure 5.56, has three ground connections; discounting those gives us 2.375.

Consequently, I think that the most likely optimum is not 1 or ∞ or even somehow 3 but 2.

Binary relations and the relational product

A few years back I spent some time on a database query language based on binary relations called Binate. It has several things in common with the work I'm describing here.

One of the things I never resolved properly in Binate was the status of constants; I treated them as relations from anything whatsoever to the constant value, like the transformation of a register state produced by a load-immediate instruction. This introduces potentially awkward infinities if you take its converse.

Binate uses a really-infix grammar whose expressions all evaluate to binary relations, so it handles non-binary finitary relations with an N-ary relational-product operation, which produces nodes that participate in a bunch of fresh binary relations.

Labels, dynamic scope, and linking

Earlier I mentioned NAND gates as an example of a three-terminal circuit element for which it's natural to consume two input nets from the stack and produce an output net, but in reality a two-input NAND gate has five terminals: power and ground. Usually in a big part of the circuit you use the same power and ground for all the NAND gates, but maybe not in the entire design. Flip-flops additionally have a clock, and it's very common to hook lots of them up to the same clock, but to have more than one clock domain in a design.

An analogous phenomenon pops up in graphics programming: it's common to do a bunch of drawing operations with the same color, the same font, the same line joins, and so on. It's annoying to have to specify this in every drawing operation.

Dynamically-scoped variables — accessible to lots of things, but set during particular dynamic scopes and restored afterwards — are one possible solution for this. In a stack machine, you could use a PostScript-style `load` operation to save the current binding of a label on the stack, then run a bunch of graph-constructing code, then restore it at the end. This requires that invoking the label connect you to its current value, like a DEFERred word in Forth or like an operator in PostScript, not like a regular word in Forth.

Another way of looking at this is that it's like what a linker does with an object file: it resolves the relocations for a given label to point to a given label definition from its environment at link time.

In some cases it may make sense to provide default arguments for things like pullup resistors and bypass capacitors, and those could be provided in a similar way, rather than taken from the stack.

Dropping brackets for parallel construction

In a sense the above uses series combination as the default operation, which can be sort of overridden by providing a different context, like the { } or < > contexts. In algebra, we also have a default operation, which is multiplication, but we don't have to write { $3x^2$ + {

`2x + 5 } }` conventionally; instead we just write $3x^2 + 2x + 5$. You could implement this by beginning with an empty sum (consisting of, say, 0) and an empty term (consisting of 1) "on the stack" and having the `+` separator sum the top two terms and push a new empty term "on the stack"; then all the other things can just implicitly multiply. This does require a final summing action on termination of input, and it can't handle infix division (it needs brackets), but that's not so relevant to graph construction.

You could insist on a single outermost set of parentheses, with these stack effects:

- `(` pushes 0 1
- `)` adds the top two items on the stack
- `+`, as before, adds the top two items on the stack and pushes 1

These rules give the right answer for nested expressions like `( ( 3 x + 2 ) x + 4 )`.

Topics

- Electronics (p. 3430) (138 notes)
- Programming languages (p. 3656) (47 notes)
- Syntax (p. 3738) (28 notes)
- Assembly language (p. 3328) (25 notes)
- Stacks (p. 3730) (21 notes)
- Databases (p. 3400) (20 notes)
- Binary relations (p. 3342) (6 notes)
- Graphs (p. 3486) (5 notes)
- Dataflow (p. 3401) (5 notes)
- State machines (p. 3731) (4 notes)
- Regexps (p. 3680) (2 notes)

Coolants

Kragen Javier Sitaker, 2017-07-04 (updated 2017-07-12) (11 minutes)

This weekend Carolina had a terrible problem in her apartment: the building's radiator sprung a steam leak, and she doesn't have a valve that can cut off the leak, so she's having to depend on the building staff to not turn the boiler on. Whenever they forget and turn the boiler on, her bedroom fills with steam until her frantic phone calls succeed in getting it turned back off.

This led me to think about the problem of fluids for heat transfer in domestic life, and in particular problems of safety in the case of pipe failure. Water has the advantages of being nontoxic, inexpensive, having a large specific heat, and having enormous enthalpies of vaporization and fusion (at accessible equilibrium temperatures). However, it is somewhat corrosive, as a vapor it is somewhat dangerously overeffective (leading to explosions and burn hazards), and its limited temperature range and expansive freezing can cause problems.

Controlling heat flow is one of the major issues in quotidian human life. It gives us hot showers, cold refrigerators, dehumidification, warm houses in winter, cool houses in summer, fired pottery, cooked food, dried fruit, hot tea, warm beds, chilled sprains, and cool foreheads when we have fevers. Lack of air can kill you in minutes, lack of water can kill you in days, lack of food can kill you in weeks, and lack of sanitation can spread your diarrhea to your whole town, but lack of cool can kill you in an arbitrarily short period of time: milliseconds or less.

The major ways we control heat flow are through insulation, thermal mass, glazing, active heating, and convection. Convection is the one I'm focusing on here, because it allows the control of arbitrarily large amounts of power with arbitrarily small ones, given adequate available thermal mass and insulation. Small electric fans are a common way that we control enormous amounts of thermal power using very small amounts of mechanical power.

Candidate convection fluids as alternatives to water include air, glycerin, vegetable oil, propylene glycol, propane, mineral oil, sulfur hexafluoride, dimethyl sulfoxide, eutectic lead-tin mix, methyl ethyl ketone, d-limonene, Fluorinert, acetone, ammonia, turpentine, carbon dioxide, ethanol, difluoromethane, R-410A, tetrafluoromethane, fluoromethane (HFC-41), fluoroform, low-molecular-weight polyethylene glycols, and non-glycerin sugar alcohols (such as sorbitol, mannitol, maltitol, xylitol, erythritol, isomalt).

Air

Air is nonflammable, among the least toxic alternatives, and has the widest working temperature range. It has very low viscosity, allowing it to be pumped easily, and has the lowest cost of any alternative, being free if you aren't too picky about purity. Its major disadvantage is its very low density (1.2 kg/m³ = 1.2 g/ℓ = 1.2 mg/cc), which, combined with its fairly low specific heat (1.01 kJ/kg/K), requires torrential flow rates, large ducts (despite its

low viscosity), and correspondingly high insulation costs.

Air works down to oxygen's condensation point of -183°. It doesn't have a sharply defined upper temperature limit; rather, its upper usable temperature limit is usually set by the corrosive effects of its oxygen on materials in the environment, which themselves do not have a sharply defined transition point but rather an Arrhenius relation. It becomes intolerably corrosive to carbon around 600° (anthracite's glow point) or 700° (coke's glow point) and to most metals in the range from 800° to 1500°, but does not corrode fully oxidized materials such as quicklime, silica, and zirconia, nor fluorinated materials.

Needless to say, air is used constantly as a heat transfer fluid.

Glycerin

Glycerin is very nontoxic — perhaps less toxic even than water and air — and has a fairly wide temperature range, in liquid form from 18° to 290°. It is fairly nonreactive, less corrosive than water. If used alone for domestic climate control, it would be likely to freeze in the pipes in normal use, so you'd probably need either a preheating system to liquefy it or a mixture with some other substance to lower its freezing point. Even at normal temperatures, it is fairly viscous, and a solvent might help with that too.

Glycerin's autoignition temperature is 370°, and because its vapor pressure is very low, its flashpoint is 160°, so a glycerin spill is not a fire hazard under normal circumstances. It's a byproduct of biodiesel production, resulting in a low price for non-food-grade glycerin of 2¢–5¢/kg. Here in Buenos Aires current prices seem to be about AR$76/kg = US$4.75/kg for drug-grade glycerin.

Ethanol

Ethanol is sufficiently nontoxic that people drink it recreationally (7000 mg/kg ORL-RAT LD_{50}), but this also makes it expensive, about US$4 per liter here in Buenos Aires. It's considerably less corrosive than water, although it does dissolve many plastics, including some varnishes. It has an unremarkable heat capacity (0.11 kJ/mol/K, which at 46 g/mol works out to 2.4 kJ/kg/K) and water-like viscosity. It has an anomalously high thermal coefficient of expansion, leading to its use as a less-toxic substitute for mercury in thermometers.

Aside from the cost issue, fire hazards are probably prohibitive for wide domestic coolant use of ethanol by itself. Its autoignition temperature is 365°, but it is considerably more inflammable than this suggests because of its very high vapor pressure — 6 kPa at 20°, which leads to about a 6% concentration in air; its flashpoint is 16°, so a large ethanol spill in an inhabited area is almost certain to explode unless rapidly remediated. Unlike most organic solvents, it's miscible with water, so remediation is possible just by dumping water on it.

Vegetable oil

Vegetable oils vary considerably depending on source, but the commonly-available ones are so nontoxic that they are used as macronutrients for cooking, although they are not quite as nontoxic as glycerin (which, incidentally, is easily prepared from them by

transesterification or saponification). Sunflower oil is currently the cheapest I can find here in Buenos Aires, at AR$190/10ℓ = AR$19/ℓ ≈ US$1.20/ℓ ≈ US$1.40/kg; perhaps this is because Argentina is the world's third biggest producer of it. Historically, soybean oil was usually cheaper (Argentina is also a major world producer of it) but I can't find cheap soy oil here now.

Vegetable oils function over a relatively wide temperature range. Rather than boiling like most of the other chemicals discussed here, they begin to thermally decompose; because of its high saturated-fat content, sunflower oil is typically stable up to 230°, although refined forms can survive to 250°, nearly as high as soybean oil. It remains liquid down to -17°, and soybean oil down to -16°.

They are not inflammable (they will not support flames until well above the smokepoints mentioned above) and they are extremely noncorrosive, protecting metals, woods, and leathers from other kinds of corrosion. Because they are relatively poor solvents, they generally will not dissolve plastics, although they can plasticize some of the softer plastics such as polyethylene and polypropylene. If there is a pipe leak, however, cleaning the oil from the things that it has soaked would often be tricky, requiring detergents or even organic solvents (dry-cleaning).

Vegetable oil fires are definitely not to be extinguished with water — they won't burn until they are hot enough to instantly flash the water into steam, which would aerosolize the burning oil and convert a mere fire into a huge explosion.

Propylene glycol

Propylene glycol is a very nontoxic† (33700 mg/kg LD_{50}) alcohol commonly used as antifreeze, as a solvent for drugs, and as a food additive; in the US it's legal as up to 5% of an alcoholic beverage or 24% of a confection or frosting. Among its medical uses are direct application to human corneas to reduce edema. Cases of toxicity exist in the medical literature but generally result from continuous intravenous use. At about 50 centipoise, it's not as viscous as glycerin, but it's more viscous than water and ethanol. (Chemically, you get either of the propanols by hydroxylating propane, to propylene glycol by hydroxylating either of the propanols, and to glycerin by hydroxylating propylene glycol.)

It is antimicrobial. (Because hey, it's only *so* nontoxic.)

It has an unremarkable specific heat of 2.5 kJ/kg/K and is commonly used as a heat transfer fluid, both alone and mixed with water.

It has a wide temperature range, not boiling until 188° and solidifying into a glass at -60°. It's not much of a fire hazard, with a flashpoint of 220° due to its low vapor pressure (10.6 Pa at 20°) and an auto-ignition temperature of 700°, a higher temperature than anthracite. In fact, I think I've heard that it's used as an antifreeze in industrial fire suppression sprinkler systems.

If exposed to air at high temperatures, it can oxidize over time.

Propylene glycol currently runs about AR$130/kg (US$8/kg) here in Buenos Aires; the most popular use seems to be mixed with glycerin to promote evaporation for vaping in e-cigarettes.

Like glycerin and ethanol, propylene glycol could, if it caught fire, be extinguished with water, because it's miscible with water.

Propylene glycol is a relatively good organic solvent, so a spill of it might cause damage to plastics and varnishes; it's commonly used as a "permanent" plasticizer. It can dissolve about 1% of its own weight in most vegetable oils, but is immiscible with hydrocarbons.

† Except to cats. Propylene glycol is toxic to cats.

Propane

Liquefied propane gas is commonly used as a refrigerant, mixed with butane; unlike most of the other heat transfer fluids surveyed here, we can take advantage of its enthalpy of vaporization, which allows the use of a much smaller amount of heat transfer fluid. However, propane is ridiculously inflammable.

Mineral oil

Mineral oil, or paraffin oil, is a mix of alkanes of a relatively consistent chain length and low vapor pressure; it is relatively nontoxic, but indigestible, and is commonly used as a laxative. It is used as a heat transfer fluid in electric radiators, for high-voltage electric transformers, and in computers. Different grades have different freezing and melting points with different degrees of definiteness, but a typical melting point is -4°; it can survive slightly higher temperatures than vegetable oil, up to

Topics

Extending heckballs

Kragen Javier Sitaker, 2019-11-26 (6 minutes)

In Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) I wrote a bit about Heckballs (Jecvals), although that dates to before I had done a bunch of the work I actually did on them.

Since then I've messed around quite a bit more with sheet cutting design (see for example Cardboard furniture (p. 742)) and come up with some other ideas.

Locking tabs in slits for sprung snap joints

One of the biggest improvements I haven't yet tested solves the problem of Heckballs falling apart once assembled due to the imprecision of the press fit. I saw this solution in a discarded liquor-box partition cardboard on the sidewalk one day. By extending the insertion bevel or chamfer or divot on one side of the mouth of a slit further down the slit, between one third and halfway down, it becomes possible to flex the inserted sheet significantly to one side after it's already inserted in the slit; the non-mouth end of the bevel becomes a fulcrum for this bending. Then, we add a roughly triangular projection to the opposite side of the slit, at the mouth; one side of it forms an insertion bevel at around 45°, while the other side is at right angles to the slit, forming a retaining clip. A corresponding hole is added to the sheet for it to insert into.

I've done this with cardboard and a scalpel, and in that medium it works beautifully, but MDF is quite a bit more rigid; will it work in Jecvals?

To be concrete about the dimensions, the octagon width in Heckballs is 100 mm, so the shallow slits are 25 mm. (There are also deep slits for joining two octagons into a ball.) I was cutting in 3 mm MDF with 2-mm chamfers (i.e., the chamfer forms a 2mm, 2mm, $2^{1½}$ mm right triangle) and 3.03 mm slit width. Extending the chamfer to 12.5 mm down the slit and 3 mm of extra slit width gives 12.5 mm of bendable span. We'd like to use a tab that presses on the full thickness of the inside surface of the hole, which would make it 3 mm tall, 3 mm deep, and a $3\sqrt{2}$ diagonal if it uses 45°.

But how much can we bend without stress-relief cuts? We only have 12.5 mm - 3 mm = 9.5 mm of bend length to work with! According to Heckballs: a laser-cuttable MDF set of building blocks (p. 2782), MDF has elongation at break of 0.45%; if we use 0.3% to be safe, the inner part of a circular bend can be 0.3% shorter and the outer part 0.3% longer, so the bend radius must be 150 times the material thickness or more: 450 mm. $450\text{ mm} - \sqrt{((450\text{ mm})^2 - (9.5\text{ mm})^2)}$ is, unfortunately, only 0.1 mm, which is so small that it might just slip out. If we extend the chamfer depth to 20 mm, leaving only 5 mm at the bottom of the slit to hold things in place, we might have 19 mm of bend; $450\text{ mm} - \sqrt{((450\text{ mm})^2 - (19\text{ mm})^2)} = 0.4\text{ mm}$, and an 0.4 mm tab is enough for some energy barrier to retention, but at a heavy cost to rigidity, and only about 10% of the crush strength of the material.

Also you'd probably want to chamfer the end of the retention tab triangle to keep it from breaking off, fillet the base one way or

another to prevent a stress riser, and angle the retention tab contact surface so that it always makes contact over at least some of its surface despite manufacturing variations.

Three perhaps more viable approaches:

• Have a full 3-mm-thickness tab that engages and disengages the hole without any elastic deformation directly, using a rigid-body relative motion, and some other much smaller normally-zero-load latching tab using elastic deformation to prevent that rigid motion. I'm not sure exactly how this would work yet.
• Instead of relying on sheet flexion to provide the elastic engagement motion, cut an in-plane flexure to allow the *latching tab* to move, perhaps like an injection-molded plastic clip. This can easily achieve much thinner bending sections (200 microns is feasible) and thus much smaller bending radii, and tabs can grab from both sides of the hole, cutting the distance of flexion in half, and avoiding accidental disengagement from structure loading. This might require more complex cuts.
• Use acrylic or cardboard or something instead of inflexible MDF.

Such clipping approaches would provide Jecvals structures with much-needed tensile strength and dimensional precision.

Panels and beam holes

The Jecvals beams form a lattice of the balls 400 mm center to center; this lattice can contain 400-mm squares and 400-mm equilateral triangles. If we add some holes along the centerlines of the beams, we can make panels with tabs that slot into these holes, thus enabling the construction of things like shelves, screens (*biombos*), and tables. To keep adjacent panels from trying to fill up the same hole from opposite sides, there should be at least four different hole positions, none of which are at the center point of the beam, so that flipping a panel over will move it into a different hole in the beams.

In the case of the triangles, there will definitely be non-right angles between the panels and the beams, which means that the holes need to be wider than the panel material thicknes.

In some cases the panels can be made of other materials, such as 1.5-mm MDF, acrylic, or colored paper.

It would be desirable for the holes for the panels to permit the engagement of other parts as well, such as beam ends, but I'm not sure how to make that possible.

Topics

Solar-cell Geiger counters

Kragen Javier Sitaker, 2016-07-30 (1 minute)

Photovoltaic cells are generally made with a PIN structure, with a large undoped intrinsic region between the P and N doped regions. I think that such a PIN diode, when back-biased (the normal case with a photovoltaic cell) will conduct like the avalanche tube of a Geiger counter when exposed to gamma rays or some other kinds of radioactivity.

There are cheaper PIN diodes available than solar cells, but for detecting radioactivity, a large total area is advantageous.

opengeiger.de has an open design for a PIN-diode-based Geiger counter using a low-capacitance PIN diode.

Probably the most practical radioactivity sources for testing such a Geiger counter are radioactive fluorescent light starters (now, thankfully, becoming obsolete) and supermarket potassium chloride, which emits mostly β particles and some gamma rays.

Topics

A one-operand stack machine

Kragen Javier Sitaker, 2016-07-24 (updated 2016-07-25) (12 minutes)

Zero-operand stack machines pack lots of operations per byte, since each instruction is only five or six bits, but typically they need close to twice as many fundamental operations as register machines. This makes register "bytecode" like that of Lua less memory-dense, but it has lower interpretive overhead. The Mill CPU architecture uses a "belt", a fixed-size random-access history of the last few instructions' results, to eliminate the need for register machines to specify destination registers in most instructions, but source registers are still necessary.

Register machines have much better code density than pure memory-to-memory machines like Chifir. They don't need a lot of registers to get much better code density; at eight registers you occasionally spill to memory, and at sixteen (like amd64) you almost never do.

The reason stack machines use more operations is that they have to use extra operations to get the operands into place to operate on them. In the common case, the things you want to operate on are right on top of the stack and are only used once, but often enough, they are not.

Implementations of stack machines often use registers — physical bits of silicon, unless you're doing this in software on top of a hardware substrate that does register renaming — for the top item or two of the stack. You have a "TOS" register, "top of stack", and maybe a "NOS" register, for "next on stack". Then, at RTL, the effect of an operation like + is something like this:

```
TOS ← TOS + NOS
NOS ← pop_overflow_stack()
```

The operation OVER, which pushes a copy of NOS, becomes:

```
push_overflow_stack(NOS)
TOS ← NOS
```

In i386 assembly, if you use %eax as TOS, %edx as NOS, and %esp to point to the overflow stack, these become:

```
plus: add %edx, %eas
      pop %edx
      NEXT
over: push %edx
      xchg %eax, %edx
      NEXT
```

In a hardware implementation, the TOS and NOS registers can be directly wired to the ALU, avoiding propagation delays from the multiplexers that would otherwise be needed.

The basic idea

As an alternative to pure stack or register machines, you could

imagine having several alternate TOS registers, each selected for the duration of a single instruction. If you had four possible TOS registers, you would need two bits in the instruction. Maybe that would eliminate a sufficiently large fraction of the five-bit stack operations that it would be a win.

The circle midpoint algorithm as example code to try designs with

Consider this Python implementation of (some variant of) the midpoint algorithm for drawing circles:

```
def mid(r):
    # `e` is always x² + y² - r²
    x, y, e = r, 0, 0
    while x > y:
        if e > 0:
            e -= x + x - 1
            assert e <= 0
            x -= 1

        yield x, y

        # For efficiency you could fold this into the above update of
        # `e` in the decrementing-`x` case.
        e += y + y + 1
        y += 1
```

On a stack machine

If rendered more or less directly into ANS Forth, I think this comes out as follows:

```
\ `e` is always x² + y² - r²
variable x  variable y  variable e  variable r
: mid
    dup r ! x !  0 y !  0 e !
    begin x @ y @ > while
        e @ 0> if
            x @ dup + 1- negate e +!
            e @ 0 <= assert
            x @ 1- x !
        then

        x @ y @ yield

        \ For efficiency you could fold this into the above update of
        \ `e` in the decrementing-`x` case.
        y @ dup + 1+ e +!
        y @ 1+ y !
    repeat
;
```

Now, aside from `yield` and `assert` not existing normally, and aside from the question of whether you should maybe refactor this into

smaller functions and maybe store the variables in a struct or something, this Forth is a bit more memory-access-heavy than it needs to be. It leaves both stacks empty at the end of each line. This is the easiest way to write Forth, because it completely avoids the temptation to get tricky with what you have on the stacks.

But let's see if we can maybe make it a bit more compact by storing *one* of its four variables on the stack. `e`, say. Also, we can get rid of `r`, because we never use it except to initialize `x`.

```
variable x  variable y
: mid
    x !  0 y !  0
    begin x @ y @ > while
        dup 0> if
            x @ dup + 1- -
            dup 0 <= assert
            x @ 1- x !
        then

        x @ y @ yield

        y @ dup + 1+ +
        y @ 1+ y !
    repeat
;
```

This is a little more compact and less memory-intense. We can go further and store `x` on the stack underneath `e`, compacting further:

```
variable y
: mid4
    0 y !  0
    begin over y @ > while
        dup 0> if
            over dup + 1- -
            dup 0 <= assert
            swap 1- swap
        then

        over y @ yield

        y @ dup + 1+ +
        y @ 1+ y !
    repeat
;
```

This is a little opaque for my taste, but it's not too unrealistic. In Forth itself, we could go further and store `y` on the return stack, but in this case I'm just using Forth as an example stack machine that I can conveniently test code on.

The above subroutine contains 39 instructions:

- 9 immediate constants, 6 of which are `y` and the other 3 of which are `0`;
- 6 memory accesses, which are 2 stores to `y` and 4 fetches from `y`;

• 2 calls to other functions (`assert` and `yield`);
• 9 stack manipulations, which are 3 `over`s, 2 `swap`s, and 4 `dup`s;
• 9 ALU operations, which are 2 comparisons, 4 `+`s and `-`s, and 3 `1+`s and `1-`s.
• 4 control-flow instructions, consisting of two conditional jumps (`if` and `while`), one unconditional jump (`repeat`), and one subroutine return (`;`).

How big is this?

If we figure that the base instruction opcodes cost 5 bits each, the function calls and immediate constants cost another 16 bits each, and the jumps cost another 5 bits for the jump offset, then the total is (+ (* 39 5) (* (+ 9 2) 16) (* 3 5)) = 386 bits, or 49 bytes. (We could probably tweak that a bit, but it's easy for that to amount to overfitting to this function; instead I would argue that this is a reasonable, if imperfect, estimate.)

On a two-operand register machine

Let's consider what it would look like in a two-operand register machine instead. Suppose we have registers A, B, C, D, E, F, G, and H, so that we need three bits to specify a register operand; and let's say that registers A and B hold the first two arguments to a function. Then it might look like this.

```
mid:    B ^= B      ; Y; sets Y to 0 with XOR
        E ^= E
 loop:    C := B
          C -= A    ; X
          JPZ done  ; jump if positive or zero
          C ^= C
          C -= E
          JPZ else  ; skip the following if E > 0
            E -= A
            E -= A
            E++
            C ^= C
            C -= E
            C := ispz(C)  ; an instruction like i386 LAHF
            call assert
            A--
 else:    C := A      ; Let's say our VM autosaves registers; it still needs
          call yield2 ; space for yield to maybe return at least one value!
          A := C      ; So we manually restore A.
          E += B
          E += B
          E++
          B++
        JMP loop
done:   return
```

This is only 25 instructions, which is a lot less interpretive overhead (although of course what matters in that case is not how many instructions are in memory but how many are executed), but each of them is bigger. We're down to only two immediate constants, we still have three 5-bit jump offsets, and let's suppose that our opcodes still need 5 bits even though we don't need stack manipulation

operations any more, and that the 3-bit register-number fields are present even in instructions like ++ and jumps that don't need them. So we're down to (+ (* 25 (+ 5 3 3)) (* 3 5) (* 2 16)) = 322 bits, or 41 bytes.

(This is clashing somewhat with my experience that stack machines in general and Forth in particular usually have very compact code, but...)

On a hybrid one-operand stack machine with four TOS registers

Now let's suppose we have four alternate TOS registers A, B, C, and D, and each of the machine's primitive operations is suffixed with the name of the register it uses for the top of stack. We can assign A to X, B to Y, and C to E, and suppose that our argument is passed in A.

To get a value onto the stack lower down, which is "shared" in the sense that all the levels below the top use the same stack, we can use dupA, dupB, and so on.

I'm supposing here that yield2 takes A and an argument from the stack below it, and consumes them both, leaving in A whatever was below that, and that it is careful to preserve the other registers. To preserve all those registers itself, it explicitly saves them onto the stack at entry.

```
: mid
    dupB dupB -B  dupC dupC -C  dupD
    begin dupA dupB dropD >D while
        dupC 0>C if
            dupA dropD dupD +D 1-D swapC -C
            dupC dupC dupC -C <=C call(assert)
            1-A
        then

        dupA dupB call(yield2)

        dupB dupB +C +C 1+C
        1+B
    repeat
    dropD dropC dropB dropA
;
```

That's 44 instructions, up from 39, and way bigger than 25, which is kind of terrible. This is not what I expected. But there are no more immediate constants, except for the call destinations. Now it's 44 7-bit instructions, plus two (let's say) 16-bit call destinations and three five-bit jump offsets: (+ (* 44 7) (* 2 16) (* 3 5)) = 355 bits. Intermediate in density (though maybe that's only because of not using immediate zeroes this time), but unnecessarily slow.

But the code is actually kind of fucking terrible and awkward. It's easy to inadvertently clobber and super awkward to bring together two values in two different registers.

What if we invert the idea?

On a hybrid one-operand stack machine with 1 TOS

register and 4 stacks

Let's suppose that instead of registers A, B, C, and D, we have *stacks* A, B, C, and D (very vaguely similar to Bernd Paysan's 4stack processor), which share a common TOS register. Then we can store `x` in, say, the TOS register (saving it to A when necessary), `y` in the NOS of B, and `e` in the NOS of C. Is that better?

Now we can use overB or overC to access the value at the top of stacks B or C, or alternatively dropB or dropC if we want to discard the value in TOS at the same time. dupB or dupC pushes a new value onto those stacks.

To make our lives easier, let's store a 0 on D.

Here's an almost certainly buggy but probably roughly correctly sized implementation:

```
: mid
    dupA  dupA -A  dupC  dupD  dropA
    begin  dupA overB >B while
        dupC 0>C if
            dupA dupA +A 1-A -C
            dupC dupD <=C call(assert)
            1-A
        then

        dupA dupB call(yield2)

        dupB dupB +C 1+C
        1+B
    repeat
    dropD dropC dropB dropA
;
```

That's 37 instructions, all of which are 7 bits, except for the two calls and three jump offsets; (+ (* 37 7) (* 3 5) (* 2 16)) = 306 bits, or 39 bytes.

It still feels super bug-prone, since I'm trying to figure out at which moments X is in TOS and when it's on stack A.

This is better than the register machine, but only by 16 bits.

Topics

Implementing flatMap in terms of call/cc, as in Raph Levien's Io

Kragen Javier Sitaker, 2015-09-03 (3 minutes)

I think you can write `flatMap` and thus `map` and `filter` as higher-order functions in the backtracking monad if you have `call/cc`. The iterator just takes a next-continuation and a fail-continuation (or end-continuation) as arguments. `flatMap` then takes a function f and an iterator i and invokes the iterator i with a next-continuation it makes up, and the same end-continuation. The new next-continuation invokes f with the item to get an iterator j and then invokes j with the original next-continuation and a new end-continuation it makes up, which returns from the made-up next-continuation. Then `map` and `filter` are simply invocations of this `flatMap` with slightly modified functions.

This structure allows you to do flatMap (and Python generator expressions) without arbitrary intermediate storage and therefore without a heap. In fact, I suspect you can do it quite easily without a heap in assembly language, piling up the various stack frames of the dynamic nesting structure of iterators by pushing the stack pointer lower and lower, resuming back and forth between the various coroutines by storing and restoring PC and BP and whatever other callee-saved registers your ABI requires.

Normally, I suppose, the fail continuation would just be the regular return path.

This idea turns out to be central to Raph Levien's Io language; although `flatMap` is not in the Io material I've seen (which does not include the original paper) it is very short to define, though to my mind somewhat tricky:

```
flat-map: -> f items k1;
    k1 -> return yield;
    items return -> item next;
    f item -> transformed-items;
    transformed-items next -> transformed-item next-transformed-item;
    yield transformed-item;
    next-transformed-item.
```

Here I have omitted as extraneous (and possibly ambiguous) the parens around action-valued variables as arguments that are used in the original paper; and I am using the convention from the paper that streams take as arguments first the return-continuation and then the yield-continuation. So, for example, the return-continuation for transformed-items is the resumption continuation for the items stream.

You can write `map` and `filter` in terms of `flat-map`, but you can also write them from scratch; we can use Levien's convention that a boolean function takes if-true and if-false continuation parameters.

```
map: -> f items k1;
    k1 -> return yield;
    items return -> item next;
```

```
    f item -> transformed-item;
    yield item;
    next.

filter: -> f items k1;
    k1 -> return yield;
    items return -> item next;
    f item (yield item; next) next.
```

(Oh dude! Raph put the paper online at http://www.levien.com/pubs/io_a_new_programming_notation.pdf ! That clears up a couple of my confusions.)

Topics

- Programming (p. 3658) (286 notes)
- Assembly language (p. 3328) (25 notes)
- Io (p. 3529) (2 notes)

A bag of candidate techniques for sparse filter design

Kragen Javier Sitaker, 2019-09-01 (18 minutes)

Here I want to talk about some things to try for sparse filter design that might extend its range considerably.

What I mean by "sparse filters"

Sparse filters (p. 834) outlines some basic techniques for designing digital filters that require only a few inexpensive operations per sample, Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) demonstrates how to get an almost arbitrary approximate 2-D Gabor wavelet from under 30 additions and subtractions. Real-time bokeh algorithms, and other convolution tricks (p. 2661) describes some ways to do precise flat convolution filters in 2-D sparsely, and The miraculous low-rank SVD approximate convolution algorithm (p. 747) describes some once-well-known ways to do very inexpensive approximate convolution using a singular-value decomposition of the desired convolution kernel — approaches that do require multiplications, but fewer.

The conventional wisdom, as I understand it, is to use direct-form FIR filters, with their non-sparse array of coefficients which gains nothing from cascading (you might as well just convolve the two arrays of coefficients and get a single-stage FIR kernel with one fewer multiplication per sample to do), and IIR filters without any delay elements in them, ideally realized as a cascade of second-order sections, applied bidirectionally if you need linear phase delay. The Hogenauer filter, Karplus-Strong string synthesis, and PLLs hint that it's possible to do dramatically better, especially on computers and FPGAs where delay memory is nearly free but multiplication is expensive; but we still lack a theory of how to design such things.

If you convolve two FIR filters whose impulse responses are m and n samples long, the resulting impulse responses is $m + n - 1$ samples long. But, even without resorting to non-LTI filters, if you convolve two FIR filters whose impulse responses contain m and n nonzero elements, possibly with zero elements between them, the resulting impulse response may have as many as mn nonzero elements. So if you convolve n FIR filters each containing only two nonzero samples, you can get a FIR filter with 2^n nonzero samples, if there is never any overlap. This allows you to get a 2^n-tap FIR filter with only n multiplications — but not *any* 2^n-tap FIR filter, only certain special ones, ones which are as far as I know not very interesting. Still, the possibility is open that we can do exponentially better than we have done so far.

More interestingly, there are certain convolutions of FIR and IIR filters which produce FIR rather than IIR filters; this allows you to get, say, a zero-phase bandpass filter at any frequency with a selectable and arbitrarily high Q — perhaps 1000 or 100'000 — with six to ten additions and subtractions per sample.

Basics

The basic techniques I'm using are CIC or Hogenauer filters; comb filters, especially negative-feedback comb filters; linear filter composition, convolving their time-domain responses; filter addition, adding their time-domain responses; and zero-phase filter inversion.

(The notes in this section probably are not an adequate introduction to the subject, but hopefully are sufficient to tell if I have something basic wrong with my understanding; The Scientist's and Engineer's guide to DSP is a much better introduction.)

The basic Hogenauer filter is a boxcar low-pass filter built from a marginally stable recursive integrator and a feedforward comb filter; its response in the time domain is a pulse, which is of course finite, and in the frequency domain it's a sinc. Most commonly they're used for sample rate downconversion — you cascade two to four of them to get a Gaussian response in both domains, rearrange the order of the stages to put the integrators first, and decimate in between to reduce the memory requirement on the comb filters. Then, typically, you use a direct-form FIR filter at the lower rate to flatten out the passband.

Feedforward comb filters amount to adding the input signal to a delayed and possibly inverted version of itself; by destructive interference, this perfectly cancels frequencies for whom the delay is an odd number of half-cycles, or a whole number of cycles if inverted. Nearby frequencies are attenuated, but the notch is pretty sharp. They also amplify broad ranges of frequencies in between the notches by as much as 6 dB, those for which the interference is constructive. Negative-feedforward comb filters eliminate DC bias, and more generally they have a high-pass effect up to half of their first null frequency; positive-feedforward comb filters double DC bias. Feedforward comb filters' impulse responses consist of pairs of impulses.

Unity-gain *feedback* comb filters are marginally stable: their impulse response is an infinite impulse train, alternating in sign if gain is negative. Instead of sharp nulls, they have sharp resonances, with infinite gain in fact; the positive-feedback ones have resonances at the period of their lag and all its harmonics (including zero), while the negative-feedback ones have resonances at twice the period of their lag and its odd harmonics. So negative-feedback comb filters resonate at the frequencies that a positive-feedforward comb filter with the same lag would cancel, and positive-feedback comb filters resonate at the frequencies that a negative-feedforward comb filter with the same lag would cancel.

If you replace the marginally-stable integrator in a Hogenauer filter with a marginally-stable unity-feedback comb filter — positive or negative — then instead of integrating the DC component of the input, it starts integrating a pulse-train component of the input. (You need to make sure that the following stabilizing feedforward comb filter is canceling in-phase components of that pulse train.) This transforms the Hogenauer filter from a low-pass filter into a bandpass filter, one that responds equally at a fundamental frequency and either all its harmonics, including DC (if the feedback comb uses positive feedback) or just its odd harmonics (if it uses negative feedback). Its window is a boxcar and so its frequency-domain response is just a sinc, or rather a periodic, infinite sequence of sincs, so usually you

need to cascade this one too. A cascade of three or more provides a quite good approximation to a Gabor wavelet.

Cascading or composing filters convolves their time-domain impulse response and multiplies their frequency response; in particular this means that if a filter has zero response at some frequency, then so will any cascade of linear filters including it.

Given two or more filters, you can connect them to the same input and add together their outputs. The time-domain impulse response will be the sum of their individual responses, which is straightforward, and so will the frequency-domain impulse response, which is not straightforward, because the frequency-domain response is in general complex. So if two filters both amplify a certain frequency by 3 dB, their sum might amplify it by 3 dB, or by 9 dB, or they might cancel it out completely, or anything in between, depending on their relative phase.

For this reason it's often convenient to design with linear-phase filters, those whose impulse response is time-reversal-symmetric†, like the boxcar or an equal pair of impulses. By delaying your other signals relative to a linear-phase filter, you can make it a *zero-phase filter*, whose impulse response is time-reversal-symmetric *around zero time*. Linear-phase filters include boxcars, Gaussians, unity-gain positive-feedforward comb filters, and the bandpass variant of the Hogenauer filter described above — as long as its time-domain response has an *odd number* of half-wavelengths in it. They also include arbitrary convolutions of linear-phase filters and arbitrary sums of zero-phase filters.

Inversion is subtracting the null filter — the input signal — from a zero-phase, appropriately-scaled low-pass filter, converting it into a high-pass filter. This is very demanding of the low-pass filter, though: 1 dB of passband droop in the low-pass part is going to limit stopband attenuation to a miserable -10 dB!

† These have *even* time-domain response. Filters with *odd* time-domain response are *also* linear-phase but I don't know how to make them zero-phase other than, I guess, convolving them with another filter with odd time-domain response to get a filter with even time-domain response. There are other linear-phase filters which are weighted sums of odd and even filters with the same frequency response, but I have no idea how to take advantage of that.

Flattening out the passband without a direct-form FIR filter

As I said above, it's common to use a direct-form FIR filter to flatten out the passband of a CIC filter when you use it for rate downconversion. The idea is that you can run just the integrators, which just do an addition per sample, at the high sample rate, to get a kind of floppy low-pass-filtered signal that anyway doesn't have any significant signal left up at the top end to alias when you decimate it; then you can clean up that droopy passband at the lower sample rate where you have time for lots of computation per sample.

But what if you aren't using the Hogenauer filter for downconversion? What if you want a bandpass filter (using the variant I described above) or, God forbid, you want to invert the filter and get a high-pass filter out of it?

Well, it occurs to me that if you want your bandpass filter to have a somewhat flatter top, you have several alternatives. First, instead of cascading several sinc-frequency-response bandpass filters at the *same* frequency, you could *cascade* several at *different* but similar frequencies. The droop in the various frequency responses will compensate somewhat for the amplification that comes from the "integrators". Second, by using zero-phase bandpass filters — windowing their pulse-train time-domain response so that it's symmetric — you can *add* several of them in parallel.

Another alternative is to start with a much wider passband than you need — using a very short window in the time domain — and then *subtract* narrower bandpass filters above and below. Or maybe use feedforward comb filters to notch out frequencies above and below your desired passband.

Similarly, I think you can use a broad zero-phase bandpass filter of the kind described above — maybe with a Q of 2 or so — to shore up the droopy high end of an orthodox Hogenauer filter's passband.

Reducing harmonics of bandpass filters with frequency diversity

Windowing a feedback comb doesn't get you just one passband; it gets you as many passbands as will fit in your sample rate, since either all the harmonics of the baseband or all the odd harmonics will also be passbands. Since you need to cascade a few windowed combs to get acceptable stopband suppression anyway, you might try detuning them from each other a bit not only to flatten the desired passband but also so they mutually weaken one another's harmonics. This works because all the passbands of a feedback comb are equally wide (in cycles per sample) at, say, their half-power points, but harmonics are further apart (in cycles per sample) than the fundamental.

This may not be very important since you probably just want to use a low-pass filter to attenuate those harmonics.

Low-Q bandpass filters

Above I pointed out that you can get a Hogenauer-like bandpass filter with a cascade of a feedback comb like

$$y[t] = x[t] - y[t - 8]$$

and a feedforward comb like

$$y[t] = x[t] + x[t - 24]$$

to get a finite-time impulse response. This is very frugal if you want a lot of oscillations in your window and thus a high Q — you can get an arbitrarily large number of oscillations with just two adds or subtracts. But if you only want a small number of oscillations, for a filter for a broad range of frequencies, you might as well use a direct-form FIR filter:

$$y[t] = x[t] - 2x[t - 8] + x[t - 16]$$

You can see that these are equivalent by plotting the results of the following Numpy code:

```
def feedback_comb(sig, lag, gain):
    rv = sig.copy()
    for i in range(lag, len(rv)):
        rv[i] += gain * rv[i - lag]
```

```
    return rv

def feedforward_comb(sig, lag, gain):
    rv = sig.copy()
    rv[lag:] += gain * sig[:-lag]
    return rv

x = zeros(100)
x[2] = 5
y = feedback_comb(x, 8, -1)
z = feedforward_comb(y, 24, 1)
```

Various pulse sizes

If you have a single integrator connected to your input signal, you can use several negative-feedforward combs on it to get various sizes of boxcars out of it. The same boxcar can be added to itself with multiple different delays to build the overall shape of a filter kernel; indeed, going beyond series-parallel combinations, boxcars of a small number of different sizes can be used, then melted together with a low-pass filter.

Splines

If you convolve two boxcars of the same size you get a triangle function. If you convolve that triangle function with a signal sampled at intervals of the boxcar size — one impulse every boxcar length, with zeroes in between — you get a linear interpolation of the signal. This gives you an easy and fairly sparse way to generate a filter with piecewise-linear time-domain response, requiring one multiply per knot in the kernel, per sample.

If you convolve the triangle with itself, you get a piecewise-cubic approximation of the Gaussian. This is not a basis spline, but it's not far from one; it is a simple linear algebra exercise to express an arbitrary piecewise-cubic signal with knots at one-boxcar intervals in the basis of this kernel, shifted by one-boxcar intervals. (You convolve the signal at the knots with the representation of the basis spline in terms of the approximate Gaussian.) This gives you an only slightly less easy, and very nearly as sparse, way to generate a filter with a piecewise-cubic time-domain response, again, requiring one multiply per knot in the kernel, per sample.

In some cases it might make sense to sum a few different such spline kernels at different scales to different levels of detail at different times during the kernel.

A factitious sinc

A sinc looks like a Gaussian lump in the middle, a sine wave on the left that gets smaller on the left, and a sine wave on the right (180° out of phase with the left one) that gets smaller on the right. What if we add together a Gaussian and a few differently delayed copies of a shortish triangle-windowed oscillation kernel, made out of the bandpass thing described above? Could we get a frequency-domain bandpass response that looks more like sinc's ideal pulse?

Other candidates

• Recognizing a square wave, or in 2-D a zebra pattern, by convolving an impulse train (finite-length, windowed, or otherwise) with a pulse
• Combining filters by multiplication, min, or softmin rather than addition to recognize "templates"
• Measuring the ac energy along lines at different angles to identify the angles which contain most information, before running a more expensive operation (perhaps the humans do this when they tilt their heads)
• Maybe take the FFT of the signal along a line at some angle through an image to identify one or two dominant frequencies; the relative phases of those dominant frequencies in a nearby parallel line (obtainable with Goertzel or with the delay-line techniques discussed above) then give you an estimate of the dominant angle of lines running across the region between them — maybe cheaper than running a cross-correlation on the two lines
• Exponential filtering from a recursive non-unity-gain feedback comb filter (in any spatial direction, e.g. coma from lens aberration) is linear and "sparse" though it requires a multiplication; can you cut it down to a finite impulse response in the same way as its special cases above, the integrator and the unity-gain feedback comb? Even if not perfectly (e.g., due to rounding error), can you do it well enough to make useful IIR approximations of FIR filters?
• More generally, piecewise-complex-exponential approximations of filter responses
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) laments not having a way to rotate the phase of a given frequency 90 degrees without phase-shifting much lower frequencies detectably. Can it be as simple as a unity-negative-feedback comb?
• Spectral inversion is of course not linear, but it can be applied to a signal very cheaply. If you invert the spectrum of a signal on the way in and out of a linear filter, you have inverted the frequency response of that filter, though be careful of the delay so that you don't inadvertently invert the signal amplitude. In addition to doing this around $\frac{1}{2}f_s$ by alternating signs on every sample, you can do it around $\frac{1}{4}f_s$ by alternating signs every other sample, offering the potential of inverting either the signal amplitude or frequency if you get the phase delay wrong

Topics

Passive dehumidifier

Kragen Javier Sitaker, 2017-03-20 (14 minutes)

I was watching Thunderfoot's self-congratulatory video about the failure of the Fontus self-filling water bottle, and it occurred to me to do some calculations.

Looking at a psychrometric chart we can see that under the 21° 32% relative humidity conditions the dude was testing the dehumidifier under, the dewpoint is about 4°; this means you need to cool the air by 17 K before you start getting any water out. To a great extent, you could do this with a heat exchanger such as a recuperator or regenerator which cools the air on the way into the condensing chamber and warms it back up on the way out; the total energy involved is the 1.012 J/g/K specific heat of air times 17 K, or about 17 J/g. That's because the humidity ratio horizontal line on the psychrometric chart shows us that we have about 0.005 g of water per g of dry air under these conditions, which doesn't change as we cool the air down to saturation; this 0.005 ratio means this is about 3.4 kJ per g of water contained in the air, which we hope to conserve in the heat exchanger.

Water's enthalpy of vaporization is about 44 kJ/mol, 2.4 MJ/kg, or 2.4 kJ/g, and that's the energy we have to remove in order to condense the water. You also have to remove more energy to cool the air (and, less significantly, the water) further. It probably isn't practical to cool it below 0°, because then you have to waste more cold on forming ice, but (as you can see on the psychrometric chart) at 0° the saturation concentration is still .0037 grams of water per gram of air, so at that point you've only gotten about a quarter of the water out.

So in summary, you cool a gram of air down from 21° to 0°, spending 21.3 J to cool it down and another 3.1 J to extract 1.3 mg of water from it. Then you use this 0° air to cool down the incoming air through a heat exchanger as much as possible, getting back most of that 21.3 J. To quantify, I think a typical energy recovery heat exchanger can recover 85% of the "lost heat", so we probably only "lose" 3.2 J in this scenario — that is, have to actively remove 3.2 J of heat from each gram of input air in addition to the 3.1 J spent on condensation.

Thermoelectric coolers like those used in the Fontus are unfortunately rather inefficient, with typical coefficients of performance of around 0.5, though perhaps as high as 1.5 for the small 20 K temperature difference we're talking about here, according to Meerstetter's chart. But there are other refrigeration technologies with better coefficients of performance.

So, we have to remove about 3.2 J + 3.1 J = 6.3 J of energy per gram of air, and we may have to spend 12.6 J to do that with a Peltier cooler. And we get 1.3 mg of water. This works out to about 9.7 megajoules per kilogram or liter of water produced. This is higher than Thunderfoot's number of 2.3 MJ/ℓ, which was merely an approximation of the enthalpy of vaporization of water.

(Thunderfoot's original video was working from the "energy factors" of dehumidifiers, which are required to be at least 1.2 ℓ/kWh

to 1.6 ℓ/kWh to qualify for an "ENERGY STAR" label. These work out to 2.25 to 3 MJ/ℓ, but maybe that's measured on a more humid or hotter day.)

To support one person with four liters of water per day, then, we would need about 39 MJ, which is an average of about 450 watts. This is feasible for a person to carry a solar panel for, but not to exert.

Avenues of improvement

But that isn't all we can do.

Greenhouse evaporation probably works

There's supposedly a wilderness survival trick for distilling water using a sheet of clear plastic. You dig a hole in the ground, put a bowl in the center of the hole, cover the hole with plastic, weight it down with rocks around the edge, and put another rock in the center to make a dimple over the bowl. In the sun, the air under the plastic heats up like a greenhouse, evaporating water from the dug surface of the soil, until its dewpoint rises above the temperature of the plastic. Water then condenses on the plastic and runs down the slope to the dimple in the center, where it drips into the bowl.

To keep the air under the plastic saturated with moisture, it is often recommended to pile foliage in the hole, perhaps smashed up a bit in order to allow faster evaporation.

That is:

- Harness solar thermal energy to evaporate water and warm air so it can hold more water.
- Condense water from 100% saturated air instead of 30% saturated air by bringing the air into contact with moist matter.

If you can raise the air temperature to 30° in your greenhouse, which is fairly easy, and get the air to 100% saturation by blowing it through smashed-up leaves or moist dirt or whatever, then you have about 27 mg of water per gram of air. If you can cool that back down to 20° — a purely passive process, and again one that can be facilitated with a recuperative heat exchanger — you can recover about 12 mg per g of it.

Not only is this almost an order of magnitude more humidity per volume of air, reducing the problems associated with device size, the bigger deal is that it's powered almost entirely by low-grade thermal energy, like what you can get from sunlight. Some fans would be useful to circulate air between the condenser and the evaporator, and maybe some water pumps, but the mechanical power needed will be orders of magnitude smaller — less than 10 W, probably less than 1 W.

The thermal power still needs to be adequate to evaporate the water; if the thermal collector is 50% efficient (a common figure for inexpensive glazed solar thermal collectors, much better than the 16% of photovoltaic panels) then you need 4 kg/day · 2.4 MJ/kg / 50% = 220W on average. You probably need 660W during the day to average 220W for all 24 hours, so at 1000 W/m² you need almost a square meter of thermal collector per person.

Purely passive condensation probably isn't practical

You might think that you could avoid this solar evaporation

nonsense, since temperature changes between day and night are often sufficiently large to cross the dewpoint by themselves. And that's true, but even in moist climates, that is not dependable every day, and in dry climates it never happens.

Here in Buenos Aires, it's currently 21°, and the dewpoint is 18°; we should hit 18° for a couple of hours at about 4 AM. This should happen again on Thursday, where the dewpoint will rise to 19° during the day (with a temperature of 22° to 24°) and then the temperature will fall to 19° during the night and morning.

At a dewpoint of 18°, the air holds about 0.013 grams of moisture per gram of air, of which you can condense about 0.001 g/g per kelvin of cooling. So to condense 4 kg of water by cooling the humid Rioplatense air by a single degree below its dewpoint, you need to slightly cool 4 tonnes of air, extracting about 4 megajoules of heat from it, plus the 9.6 megajoules (4 kg · 2.4 MJ/kg) for condensing the water. The most reasonable way to do this is probably to cool down a cold reservoir at night by some 15 megajoules. If your cold reservoir is just water, this is going to require minimally 3.5 tonnes of water and probably 7 or 10 tonnes. On the bright side, 10 tonnes of water can be stored a lot more easily than 4 tonnes of air.

Phase-change materials, like the recently-discussed saturated aqueous solution of sodium hydroxide, probably allow you to do this with only a few hundred kg of mass.

However, all of this is overlooking the fact that it wouldn't work very well this week, because in both cases the low temperature comes *after* the high dewpoint. You'd have to store the cold from now until Thursday afternoon, four days. Doable, but a major undertaking.

Consider, by contrast, Albuquerque. Currently the temperature there is 17° and the dewpoint is -8°. Despite its 15° temperature swing from day to night, you are just not going to have any luck condensing water out of that air with a passive cool reservoir stored on a timescale of less than months; the highest the dewpoint gets is -1°.

Purely passive condensation with weather balloons

If you send a balloon up, it gets cold. The environmental lapse rate of the lower troposphere, up to about 11 km, is about -6.5 K / km, so at 11 km the temperature is usually about 70 K colder than at sea level. In almost all cases, that brings it below the dewpoint at ground level.

If a balloon goes up and then comes back down, it has done no net work, so need not necessarily have dissipated any energy. By bringing heat from the ground up, or bringing cold from the upper atmosphere down, it has not done any work yet either — the thermodynamic free energy there comes from the sun heating the earth's surface, energy which is eventually dissipated by winds forming convection cells. It's just speeding up the convection a bit locally.

In practice, you need *some* energy difference to induce the balloon-atmosphere system into motion in the desired direction, but it could be very small compared to the heat you harvest. A net force of 1 N should be enough to induce a 100 kg phase-change reservoir payload to rise at an acceptable rate. If you let it rise 3200 meters, dissipating that 1 N in drag, it will have dissipated 3.2 kJ. Having cooled the payload to below 0° and dropping a little gas at that altitude so as to reduce lift by 2 N (about 170 liters at 1.2 kg/m³), it

will then return to the ground dissipating another 3.2 kJ. A payload of 100 kg of water ice at 333 kJ/kg will have lost 33.3 MJ of enthalpy of fusion, which was bought with 6.4 kJ, about 0.02% of the total. You could probably even afford 10 N or 100 N.

That said, it is nontrivial to construct and fly a balloon of the 83 m^3, 5.4 m in diameter, needed to lift 100 kg. (I'm disregarding the weight of the hydrogen or helium within as smaller than my margin of error.)

Your 100 kg of ice is enough to condense almost 14 ℓ of water at 2.4 MJ/kg.

Compressor-based heat pumps reach much higher efficiencies

Typical vapor-compression heat pumps like residential air conditioners and normal electric refrigerators have coefficients of performance around 2, which is largely a result of having to pump heat against a large heat gradient, typically 15° to 25° of difference. (Note, however, that this is already several times better than Peltier coolers typically reach.)

(Thunderfoot implicitly claims that it would violate thermodynamic law for a refrigerator to have a CoP over 1, but he is mistaken.)

We can't do much about the ΔT (we have to pump heat from the water-condensing chamber into the surrounding environment) but if we can reach that COP of 2, then instead of spending 12.6 J per gram of air to remove 6.3 J of heat, we can spend 3.2 J, which works out to 2.4 MJ/kg of water, coincidentally exactly the same as the enthalpy of vaporization of water.

If it so happens that your air is already at 100% humidity, a heat pump could in theory condense water with an arbitrarily small ΔT, just by replacing the air very rapidly. With an arbitrarily small ΔT, a heat pump could theoretically have an arbitrarily high coefficient of performance. In practice, water-source heat pumps used in Japan for residential climate control often reach a CoP of 6, which would allow you to condense water at 2.4 MJ/kg ÷ 6 = 0.4 MJ/kg.

However, as Thunderfoot points out, under those circumstances, water is probably already abundant.

Topics

Ideal language syntax

Kragen Javier Sitaker, 2017-07-19 (1 minute)

What would make a language syntax ideal for my tastes?

• Short keywords. `my` or `=` instead of `var`, `to` or underlines, instead of `function`, `for` or a vertical line instead of `while`, maybe even `do` instead of `while` or `for`.
• Little punctuation. Python-style `:` instead of `{}`. `.` instead of `->`. `:` instead of `=>`.
• Punctuation and keywords that follow existing prose use; `--` is far better for comments than `;` or `#`, which mean something totally different in everyday life.
• Not fucking monospace.
• Infix syntax with either traditional precedence or required full parenthesization in cases where it would change the meaning. Because I don't want to puzzle over why `a + b * c` is giving the wrong answer, only to find that it means `(a + b) * c`.
• By the same token, operator overloading; I certainly don't want to have to write `a.plus(b.times(c))` as in JS.
• Some kind of aspect weaving so that a particular piece of code can be focused on only the things that are relevant to a particular task — for example, dataflow, types, proofs, resource limits, escape-analysis annotations.
• Lightweight lambdas.
• Extensibility, because sometimes lightweight lambdas aren't enough.

Of course there are other aspects of programming languages that are more important than the syntax, but the syntax is important.

Topics

• Programming (p. 3658) (286 notes)
• Syntax (p. 3738) (28 notes)
• Typography (p. 3760) (5 notes)

ASCIIbetically homomorphic encodings of general data structures

Kragen Javier Sitaker, 2017-06-15 (2 minutes)

I thought I had written some notes about this previously, but I can't find them right now. I want a serialization for relatively general data structures (say, at least the S-expression or JSON data model) that possesses a useful homomorphism between some kind of natural ordering on the original data structures and the lexicographical ordering of the byte strings they serialize to. That is, if E is the serialization encoding, I want E(X) < E(Y) iff X < Y.

This is for five reasons:

• LevelDB: LevelDB can iterate over the keys that are ASCIIbetically within a certain range. In fact, that's the *only* kind of iteration it supports.
• oMQ: ZeroMQ and Nanomsg can efficiently filter messages from a pub-sub topic that begin with a given substring.
• Compressed indexing: Patricia and related trie structures, as well as FM-indices and related data structures, can efficiently retrieve and even compress data — but they only support retrieval by lexicographical prefixes, not by other arbitrary orderings.
• Suffix arrays: suffix arrays can efficiently find all the occurrences of a substring in a large file, and now there are simple O(N) suffix-array construction algorithms.
• Radix sorting: while comparison sorting is O(N log N), radix sorting is O(N).

To take advantage of these properties, I often end up writing some kind of simple ad-hoc serialization code for the data at hand, which often turns out to have bugs in it, and almost never generalizes to other kinds of data that aren't in the data I'm looking at. (For example, if I separate fields with spaces, I run into ordering errors once I have data containing ASCII control characters or spaces.)

Topics

Nonconductive relays

Kragen Javier Sitaker, 2019-11-12 (3 minutes)

Electromechanical relays have some lovely features: they can provide galvanic isolation, very high crosstalk immunity, enormous gain, very low on-impedance, and, in locking designs, bistability even when the power goes out.

However, they have some big problems: they're power-hungry and their operating speed is limited to the kHz to tens of kHz and their operating life is limited to thousands to millions of operations, which usually limits them to *average* operating speeds in the millihertz or less.

The reason for the short operating life is contact oxidation. (In theory elastic metal fatigue or creep could play a role too, but those are easy enough to avoid.) Mercury-wetted relays are a common design to lengthen this life, and Paper/foil relays (p. 3273) discusses the possibility of using contacts of carbon (like keyboard dome switches), silver, or gold instead.

In other sense-switch applications, a common approach to avoiding the oxidation problem is to use phenomena other than conduction to transfer the energy. The TRS-80 keyboard, for example, was capacitive (though using springs made of polyurethane foam, which degraded rapidly), as are modern touchscreens, touchpads, and some touch panels in embedded devices. And there are numerous inductive sensors for position, orientation, and so on.

It occurs to me that relays can work through these media as well.

For example, if you have two ferrite rods with one winding around each of them, you can make them into a transformer that efficiently transfers AC power from one to the other, from dozens of Hz up to several kHz, by completing the magnetic circuit with more ferrite. By moving this additional ferrite under the control of a solenoid, you have a relay, one that will never suffer contact oxidation, because the contacts are magnetic rather than electrical.

Similarly, although the circuits described in Paper/foil relays (p. 3273) are the usual kind of dc-coupled contact circuits, you could use a similar design to bring one of the plates of a variable capacitor into contact with the dielectric from a distance far enough to drop the capacitance by orders of magnitude. This could easily have enough capacitance to efficiently transmit power at frequencies of 100 MHz and up, again without any electrical contact and thus no oxidation. Such capacitive relays could move smaller amounts of mass, and over shorter distances, than the inductive relays described above, and so they should be able to operate much faster.

Topics

- Electronics (p. 3430) (138 notes)
- Physics (p. 3632) (119 notes)
- Physical computation (p. 3631) (26 notes)
- Relays (p. 3681) (3 notes)

Frustration

Kragen Javier Sitaker, 2018-04-27 (2 minutes)

I am constantly frustrated with the inadequacies of the computer systems we have so far achieved, and I am unconvinced that our current approaches to designing them will get us to a better situation. Yet I am devoting almost none of my time to improving the situation — even on the most personal level of coming up with a reasonable individual email system.

I want to fix this. I want to write a computing system that provides a reasonable self-hosted development environment for the kinds of things I want to work on, allowing me to use the unbelievably huge existing free software ecosystem without being limited by it. I found out two days ago that not one but two of my coworkers at Satellogic have already done this, to one extent or another, with Smalltalk. Juan reports that his system, Cuis, is only about eighty thousand lines of code. Richie's system, Bee, may be bigger or smaller, but it's more to my taste.

It seems clear that the issue is that I'm not treating the self-sufficient computing environment project as a priority, in part because of Satellogic. If I'm going to make progress on it, I need to spend time on it each day, probably instead of Twitter or some shit like that.

A thing that would really help a lot with that would be some kind of ability to use the iPhone to make progress on it while I'm in transit — a little ratchet of progress.

I think a reasonable target might be the "100,000 words" of a 300-page novel, say, 600 kilobytes or so. At five "words" per "line", that would be the same "20,000 lines" of the STEPS project.

Topics

APL with typed indices

Kragen Javier Sitaker, 2013-05-17 (11 minutes)

Parallel arrays are bad because array indices are untyped, and sometimes because of storage management and locality of reference. But what if each array had its own ⎕IO (APL's name for OPTION BASE)? This could put each type of object ID into its own namespace, so that you'd get an index error if you tried to index an array with the wrong kind of object ID. It's like how in a spreadsheet, maybe rows 1–40 are the employees, and rows 101–120 are the products, and so if you have an index between 101 and 120, you know it identifies a product and not an employee. (Too bad that in an actual spreadsheet the column names will be the same in both cases.)

You could actually index arrays using an (object type, zero-based-index) pair rather than combining the two into a single number. This would give you many of the same type-safety and index-safety benefits of structs, aka records, without sacrificing the dubious benefits of aggregate operations. (And if you can verify all the type stuff at compile time, you could drop them at runtime!)

Element-by-element scalar functions should only work on arrays representing, in some sense, attributes of the same object. Operations like "grade up" and "compress" would create new index namespaces.

"Interesting" functions from APL (not element-by-element) include monadic ⍴⍳,⌹⌽⊖⍋⍒⍉, dyadic ∈⍴↑↓⊥⊤,⌿/⍳⌹⌽⊖⍉ and indexing [], plus the operators (scan, reduce, and inner and outer product).

The details I've been able to work out follow.

Since much of this is sort of about types, I'm going to write types in a Haskelly way: `a->b` means that something (a vector) takes as an argument (index) things of type `a`, and returns (has as elements) things of type `b`; and `X: T` is an expression which has the value of `X`, but asserts that that value is of type `T`.

Multidimensional arrays

I'm going to try to work out the cases of zero and one dimensions first, before attempting to handle multiple dimensions. An N-dimensional array for N>0 can be considered as either a homogeneous 1-dimensional array of N-1-dimensional arrays, an array indexed by N-tuples, or an N-1-dimensional array of 1-dimensional arrays; both of the first two interpretations are common in APL. Changing the nature of indexing is likely to interact with them in unpredictable ways.

Indexing []

Indexing `A[B]` (also `.` or simple juxtaposition in K, or `A@B` for another variant) should produce a value with the shape of B by indexing the elements of A. That is,

```
((A: c->d)[B: e->c]): e->d
```

It's like function composition. So the elements of B need to be valid indices for A.

Compress /

Dyadic `A/B`, "compress", gives you one element of output for every nonzero element in A. Its two arguments must have the same set of indices, and the left one must be boolean. If you compress two vectors `B` and `C` with the same vector `A`, the resulting vectors need to be conformable, so you can write `(A/B) + (A/C)`, for example.

In q/k/kdb+, I think this is written `B[where A]`, helpfully decomposing the operation into two steps, the first of which is constructing a vector of the positions where `A` is nonzero. I think this is the right way to analyze this operation, because it constructs a *new index namespace*:

```
(where (A: b->c)): d->b
```

(This is a rather remarkable type, creating a new type out of whole cloth!)

Indeed, this is such a common operation in APL that the Dyalog APL tutorial gives the APL spelling of it, `(V∈D)/⍳⍴V`, as their first example of an idiomatic APL expression.

It's probably more accurate to write `(where (A: b->bool)): d->b`, since the

Expand \

I haven't written enough APL to know what the dyadic expand function is good for. In a sense, it's the inverse of the compress function, but the compress function is lossy — it leaves out chunks of its right input, which the expand function will fill with zeroes. It's guaranteed that `A/A\B` is the same as `B`, but not `A\A/B`.

It seems rather difficult to type-check in a sane fashion: `A` needs to have exactly as many `1`s set as there are valid indices in `B`.

Dyalog APL's expand function extends this to the case where `A` may contain things beyond just booleans, interpreting them as a repeat count (or, if negative, a zero count).

The Dyalog APL tutorial gives the following common uses of expand:

• to add blank or zero rows in between the rows of a matrix (for example, to separate text output into paragraphs);
• to repeat a scalar some number of times.

It also gives as an exercise the problem of replacing a given letter globally with spaces: `'a' Whiten 'Panama is a canal between Atlantic and Pacific'` should return `'P n m is c n l between tl ntic nd P cific'`. The solution, I suppose, is the `A\A/B` above, where `A` is just `~Letter = Phrase`.

http://aplwiki.com/AplIn20Minutes may have some information.

Grade-up and grade-down ⍋⍒

These are used for sorting. `⍋X` (`>X` in K) is a vector such that `X[⍋X]` is sorted ascending; `⍒X` does the reverse. This is more useful than simply giving you the sorted results because, if you have another vector `Y` that can be indexed with the same indices as `X`, you can sort the elements of `Y` according to `X` with `Y[⍋X]`.

It does not generally make sense to index `X[⍋X]` with indices that are valid for `X`, with the exception of things that are sort of coincidental; for example, `1` is probably a valid index for both of them. But it

doesn't make sense to say, for example, `(X > 3)/X[⍋X]`; that's almost certainly a bug.

So grade-up and grade-down, like compress, creates a new index namespace:

```
(⍋(X: a->b)): d->a
(⍒(X: a->b)): d->a
```

This is maybe the point at which array indices might begin to have an advantage as object IDs over raw memory addresses: once you can start taking advantage of the sequence of the array indices.

Reversal and rotation, monadic and dyadic ⌽⊖

`⌽X` is just `X` with the indices of its last axis in reverse order; `⊖X` is the equivalent for the first axis. Analogously, rotation `3⌽X` rotates `X` three items to the right, which can be achieved by reversing `X`, reversing its first three items, and then reversing the remaining items. On the face of it, it seems that these should also create a new index namespace; it doesn't make sense to write `(X > 3)/⌽X`.

However, I think this is not actually true. The valid *range* of values is the same. `1⌽X` gives you the `X` property of the (cyclically) following object in `X`'s index sequence; this is a useful kind of thing to be able to do!

```
⌽(X: a->b): a->b
⊖(X: a->b): a->b
(N: int)⌽(X: a->b): a->b
(N: int)⊖(X: a->b): a->b
```

Shape and index generation: monadic ⍴ and monadic ⍳

`⍳`, aka "iota" or "index generator" (in K, `!` or `til`) counts from 1 (0 in K) to a given number; but its primary use is to generate the valid indices of a vector given its size. The size of a vector is generated with `⍴`, the "shape" operator (in K, `count`), so `⍳⍴V` gives you the indices of `V`.

This use of `⍳` is so important that Dyalog APL's `⍳` applied to a vector generates a sequence of multidimensional indices, such that `V[⍳V]` is equivalent to V for any shape of V (which is impossible in traditional APL), and the "index origin" variable `⎕IO`, equivalent to BASIC's `OPTION BASE` can affect `⍳`'s operation.

Although this form of `⍳` takes only one argument, it's straightforward to use it to generate values starting from any starting point, increasing by every step: `start + step * (⍳N)-1`, although of course the `-1` is a result of the default `⎕IO` being 1 rather than 0.

It seems to me that it's probably worthwhile to preserve the invariant that `⍳⍴V` produces the indices of a vector `V`. If this is to work with `⍴V` producing a scalar or 1-vector in this case, which seems desirable to avoid complication for multidimensional cases, then it needs to be possible to extract both the length and the object type from that scalar; that is, `employees[301]` needs to be a single scalar object from which you can extract `employees` and 301. If we represent `employees`

as `employees[0]`, then it's sufficient to provide a function for extracting either of the two; ordinary subtraction suffices to produce the other, and ordinary addition is sufficient to recombine the two.

So produces a single-element vector:

```
⍴(X: a->b): {0}->a
⍳(X: a): int->a
```

element-of ∈ and index-of, dyadic ⍳

`A∈B` produces a boolean array of the shape of `A` indicating whether each item of `A` occurs in the vector `B`; that is,

```
((A: c->d) ∈ (B: e->d)): c->bool
```

The index type of `B` disappears entirely, because we don't care *where* the elements of `A` occurred in `B`; we're just using it as a set. If we wanted their positions, we'd use the dyadic ⍳ function:

```
((A: c->d) ⍳ (B: e->d)): c->e
```

This is a little bit tricky, because for the elements of `A` that didn't occur at all in `B`, this function produces `1+⍴B` (or `⍴B` in the `⎕IO=0` case), which is not a valid index into `B`. You could say it's not a member of `B`'s index type `e` at all.

In K, the dyadic ⍳ function is written `B?A` instead.

The things I haven't figured out how to handle yet

How do you get literal vectors in your program to have a reasonable type?

What about take, drop, and catenate? Some uses of take and drop can be reasonably handled by rotation, but others can't.

Topics

- Programming (p. 3658) (286 notes)
- Arrays (p. 3326) (17 notes)
- APL (p. 3320) (9 notes)
- Types (p. 3758) (5 notes)
- The K programming language

Git learnings

Kragen Javier Sitaker, 2007 to 2009 (3 minutes)

Here are the top few things I learned about Git, mostly in the first few hours I used it. This is the document I wished I had had, on top of the various introductions floating around. Maybe it will be useful to somebody else.

• Git handles 400MB of HTML crawl data less gracefully than it handles 700K of Python. But it handles that data more gracefully than `cp` and `rsync` do.
• Don't `git push` to a repository that actually has a work area. Always use `git pull` instead. `git push` doesn't update the associated working area, or the index either, so if you try to `git commit` in that repository, you will commit a patch that undoes all the stuff you just did. See http://utsl.gen.nz/talks/git-svn/intro.html section "Push changes and the working copy". You can solve this with `git reset --mixed HEAD`, or eventually `git reset --hard HEAD` to throw away any changes in the working area.

```
23:05 < johnw> $ rsync -av .git/ server:/tmp/foo.git/ ; cd /tmp ; git clone ssh:/o
o/server/tmp/foo.git

23:06 < johnw> that's all you need to setup a remote repository, and to start usi o
ng it right away
```

• `git repack -a -d -f` can achieve some truly astonishing compression ratios. This is how you make git checkouts faster than `cp -a` or `rsync`. In my case, three times faster than `rsync` over a slow network, due to a 7:1 compression ratio.
• You have to `git add` changed files before you can `git commit` them, or use `git commit -a`, because `git commit` commits things from the index, not your work area. In older versions of Git, you used `git update-index` instead of `git add` on changed files.
• `git commit` takes an option `--amend` which lets you amend the previous commit. `git rebase --interactive` lets you amend previous commits in general. Both of these don't really work if you've shared the commits with someone else.
• `git clone -l` makes a hardlinked clone. (This is default in newer versions of Git. It's another factor in making git checkouts fast.)
• git has early-stage support for something called "submodules" in recent versions, similar to `svn:externals`. And there's an in-development `git hunk-commit` command that might end up in git someday that should add most of Darcs's UI niceness to git, although `git gui` or `git add --interactive` get you partway there already. http://raphael.slinckx.net/files/git-darcs-record

02:38 < twb> I found git commit --interactive pretty confusing.
02:39 < andreaja> twb: I prefer to use git add -p

• If you try to `git pull` when you have un-checked-in changes, `git` will complain with an unhelpful error message. Check in the changes or `git stash` them before you pull.

I took the first 125 563 056 bytes of my mailbox and compressed them into 59M with git. However, git (1.4) doesn't seem to work very well with multi-gigabyte quantities.

If you're using the Git 1.4 from Debian Stable, you'll want to know to use `init-db` instead of `init`, `repo-config` instead of `config`, and often `update-index` instead of `add`.

http://cheat.errtheblog.com/s/git
http://www-cs-students.stanford.edu/~blynn/gitmagic/

Topics

Why is there so much anti-plastic sentiment? Visibility, Arcadian primitivism, conspicuous consumption, and profit.

Kragen Javier Sitaker, 2018-06-21 (7 minutes)

Posted to https://news.ycombinator.com/item?id=17369521

The biggest problem with making statements about "plastic" is that it's such a broad category of materials that you almost invariably doom yourself to talking nonsense.

Most plastic (particularly PET, PP, HDPE, and LDPE) is not poisonous, even when decomposed, and does not bioaccumulate in the conventional sense. Perhaps by "poison sponge" you don't mean that it's poisonous; you mean that it absorbs poisons. Well, isn't that what you would want to do with poisons? Clay and activated charcoal absorb and adsorb poisons too. That's why you feed them to poisoning victims. The problem is the poisons, not the plastics.

Using alternatives where they exist may or may not be an improvement. Usually you can use glass bottles instead of plastic bottles, for example. But a PET 2-ℓ bottle might be 27 grams of extremely nontoxic PET. The corresponding glass bottle weighs nearly a kilogram, 30 times as much. This means that most kinds of environmental damage associated with it increase by one and a half orders of magnitude; you need 30 trucks instead of one to transport the bottles to the bottling facility, 30 tons of raw material instead of one to make 30,000 bottles, and so on. The glass also needs higher processing temperatures, using more energy, and produces broken glass when discarded, so the disparity is actually somewhat larger.

Similarly, a traditional plastic shopping bag is entirely nontoxic, weighs 100 milligrams, and can be reused three or four times, but is sterile the first time you use it. If you replace it with a canvas bag that weighs 65 grams and can be reused hundreds of times, you're using 650 times more material in exchange for only 100 times more uses, and you dramatically increase your risk of food poisoning from raw vegetables. Washing the canvas bag once will typically use both more energy than the plastic bag used during its entire lifecycle and also more material — it's hard to get it clean with only 100 milligrams of soap!

A better tradeoff is to use slightly thicker plastic bags which can survive dozens of uses, use new plastic bags for your raw vegetables, and bury the plastic safely when you're done with it. (A common pathological effect of regulations against plastic shopping bags is that people have to replace them with plastic garbage bags.)

Being able to use 30 or 100 times less material, and common, nontoxic materials like carbon and hydrogen instead of rare, toxic materials like boron and chromium, are major environmental advantages of *many* plastics in *many uses*. There are *some uses* of plastics which are environmentally harmful, and there are *some plastics* which do generate toxic products if they are allowed to break

down — most notably PVC.

Car fuel consumption is, generally speaking, closely proportional to weight; and, for a given emissions control system, harmful emissions are closely proportional to fuel consumption. Individual car weight has diminished enormously during the last 50 years primarily due to greatly increased use of plastics, though improved metals have also played a role. You can build cars almost entirely from metals, with only a few crucial components such as gaskets and hoses made from plastics, but doing so is environmentally harmful.

So why is there so much anti-plastic sentiment? I think there are four main reasons, aside from the actual environmental damage from some uses of plastics.

• Plastic is very visible, and renouncing plastic is a low-cost, high-visibility way to advertise your virtue as an Environmentally Conscious Person. Doing things that would actually have a significant benefit to the environment, such as not having children, not eating meat, not financially supporting the US military through taxes, carefully weighing the costs and benefits of possible actions, and using passive solar climate control in your house, is costly and therefore unpopular.

• I'm a hippie, and the horror at what industrial civilization is doing to our beautiful planet leads many hippies to reject industrial civilization entirely. This kind of anarcho-primitivism, which I do not agree with, considers products of industry and especially petroleum and chemistry to be undesirable, at times even ritually impure, like cannibalism. Thus canvas bags are preferable to plastic bags, wool (such as actual fleece) is preferable to microfiber polyester ("polar fleece"), and brass is preferable to plastic, entirely independent of their actual environmental impact. This creates a sort of coincidental association between environmentalism and the rejection of plastic which provides fertile ground for anti-plastics arguments and barren ground for pro-plastics arguments.

• At least in the US, the upper class considers wool preferable to microfiber polyester and brass preferable to plastic for an entirely different reason from anarcho-primitivists: they consider innovation and cheap goods to be déclassé, deriving much of their social value from a traditionalist, Romanticist value system. Cynics might also point out that using expensive goods where cheap ones would do serves as a form of conspicuous consumption, reliably signaling the wealth of the consumer to observers. Either way, the upper class — which still controls much of the press in the US, and thus has a powerful role as tastemakers, despite the rise of lower-class celebrities like the Kardashians and Trump — is also fertile ground for anti-plastics arguments and barren ground for pro-plastics arguments.

• Selling alternatives to disposable plastics is very profitable. As a simple example, a supermarket can sell cloth bags instead of giving away plastic bags for free. Many times, its customers will forget to bring cloth bags with them, and will buy more bags than they need, so in practice a single cloth bag will replace 10 plastic bags instead of 100. This works as a form of price discrimination, since customers on tighter budgets will be more careful to bring bags to avoid the artificially imposed cost. Finally, this makes the supermarket appear

upscale, both because it's advertising its hip environmental consciousness, and because it is less associated with déclassé things like plastic bags. This enables it to charge higher profit margins on its other merchandise.

https://en.wikipedia.org/wiki/Microplastics

Topics

Paper/foil relays

Kragen Javier Sitaker, 2019-04-02 (updated 2019-10-23) (13 minutes)

Preliminary calculations suggest it's feasible to build electrostatic relays out of paper and graphite that operate reliably for millions of cycles at frequencies from DC up to medium-wave RF, at voltages of dozens to hundreds of volts, with individual devices that can be clearly seen with the naked eye. Actually this is so absurdly good on paper that surely someone has tried it and the scheme has some hidden fatal flaw I'm not seeing. Details follow.

MEMS electrostatic relays

I was reading about electrostatic relays the other day. MEMS relays are made using nanophotolithography techniques, just like CMOS, but have significantly different performance characteristics; they turn on and off more slowly than MOSFETs do, but once they're active, they pass signals with lower resistance and thus higher speed; and they don't have a linear region the way MOSFETs do, having near-infinite gain at their transition point. Careful circuit design can get circuits with comparable performance.

The idea of an electrostatic relay might seem paradoxical: wouldn't you need a higher "gate voltage" to turn it on and off than what it can switch? The solution taken by the MEMS designs is very simple: the contact area is small, and the signal conductor is narrow, while the gate area is large. (They are insulated from one another with a layer of amorphous silicon dioxide.) Because the contact area is small, the electrostatic force generated by the signal being switched is proportionally small, perhaps two orders of magnitude smaller than the force generated by the larger-area gate.

Other solutions are possible, involving things like leverage to allow a smaller electrostatic force to outwrestle a larger one, but just using larger and smaller plate areas is simple enough.

Macroscopic paper foil electrostatic relays

It occurred to me that you could build such devices macroscopically, or mesoscopically (with, say, a characteristic dimension of 100μm to 1mm rather than 10mm or 1μm) out of paper and foil. The most obvious conductors to use, aluminum and copper, are not very suitable for relay contacts for operation in air — sparks at the contacts will produce hard, nonconductive oxide layers which will make the relays unreliable in short order. Reasonable alternative contact materials include gold (reduces its oxide), silver (has a conductive oxide), graphite (has gaseous oxide), tin (has a semiconducting oxide), and lead (has conductive metallic dioxide).

Typical electromagnetic relays are good to only about 10k to 100k operations, although high-reliability mercury-wetted reed relays sometimes advertise a million. These electrostatic relays should be able to last many times longer through the use of more appropriate materials, much lower circuit inductance, and much lower currents. But this is speculative.

Spring materials might be trickier. Paper might work okay at least for prototypes, but ideally you'd like something that doesn't creep or

fatigue over time, such as glass foil or foils of other metal oxides. Alternatively, instead of relying on the insulator to provide a spring force, we could rely on the conductor — metallic conductors such as copper, silver, gold, or even iron should be immune to creep and fatigue at these temperatures, deformations, and thicknesses, and a thin film of some insulator could be deposited onto the surface of the gate conductor to separate the gate from a later-deposited channel.

A 3mm×3mm paper foil relay might consist of a 100μm-thick square of aluminum foil on the gate side (this is conservative; common household foil is 22μm, while ribbon-microphone aluminum-leaf is 0.6–2μm), attracted toward a 100-μm-thick square on a substrate paper, which has a 100-μm-thick insulating paper layer (again, conservative; ordinary 80 g/m² office paper is this thick, but 80-μm 60 g/m² paper is easily available) separating it from the traces being bridged, which might each have a 1mm×1mm contact area. Actually you probably only need one such area, but let's keep it simple. The moving gate might move by 100μm to bring the contacts into contact.

Our 3mm × 3mm sheet works out to be more like 5mm × 5mm including quiet zones around it:

```
EE EE EE EE EE
EE AA AA AA EE
DD BB CC BB DD
EE AA AA AA EE
EE EE EE EE EE
```

Here the different pixels have stacked-up contents as follows when the contacts are open, with "G" being the gate electrode, "." being air, "I" being insulating paper, "C" being the contact material, "N" being the channel conductor (for example, copper), and "S" being the substrate electrode, which can be aluminum or copper or whatever.

```
AA:   BB:   CC:   DD:   EE:

GGGG  GGGG  GGGG  ....  ....
IIII  IIII  IIII  ....  ....
....  CCCC  NNNN  ....  ....
....  ....  ....  ....  ....
....  CCCC  ....  NNNN  ....
IIII  IIII  IIII  IIII  IIII
SSSS  SSSS  SSSS  ....  ....
```

If each of these layers is 100μm thick, which seems plausible, we have 500μm between the gate and the substrate at the point where we want to activate the thing. The total moving mass is about 0.9 mm³ of aluminum gate (about 2.4 mg) plus a similar volume of paper (about 0.9 mg) plus 0.3 mm³ of channel (say, another 1 mg, depending on what you make it from) for a total of 4.3 mg. You want to somehow hinge or spring it so that the paper spring restoring force is large comparable to the weight of the 4.3 mg (so it won't fail from being upside down) but not too enormous. How much electrostatic force can we expect?

Say we run the thing on 200V, since it's an electrostatic device and those have always required a fair amount of EMF to do anything.

Coulomb's law $F = k(q_1q_2)/r^2$ tells us that two 1-nanocoulomb point charges 500 μm apart will generate a 36-millinewton force. But how much charge do we have at 200 volts? If our capacitance $C = \varepsilon A/d$ and our ε is ε_0 — probably a good approximation for paper, and an excellent one for air — our capacitance is 0.160 pF, so we have about 32 picocoulombs on each foil, giving about 1.12 mN, under whose influence the relay will snap shut at initially 259 m/s², about 26 times the acceleration of gravity, which is in the right ballpark. It might even be possible to use lower voltages like 48V or 24V.

https://www.hindawi.com/journals/jchem/2017/4909327/ may be relevant; on a single sheet of notebook paper (probably 80 μm?) the dude got 53 pF/cm², so 0.53 pF/mm² (?), which is in pretty close agreement with what I calculated above for the larger plate separation encountered in an electrostatic relay.

(I think we can neglect the electrostatic force of the channel since, as I said above, we can make it almost arbitrarily narrow.)

Neglecting the spring force, which I think can be substantially smaller than the electrical force, we have in theory an operational speed of around 2 μs, which in theory requires about a 16 μA charging current — though the capacitance will increase by 20% by the time the contacts come into contact, generating additional charging current and electrostatic force.

At currents and voltages like these, resistances below a few hundred kilohms will have no effect, so you might as well use graphite for all the conductors, rather than trickier and heavier metal foils. (Silver's resistivity ρ is 1.59×10^{-8} Ωm, copper's 1.68×10^{-8}, gold's 2.44×10^{-8}, aluminum's 2.82×10^{-8}, lead's 2.2×10^{-7}, amorphous carbon's $5–8\times10^{-4}$, and graphite's $2.5–5.0\times10^{-6}$ perpendicular to the basal plane. So a 100μm×1mm×3mm trace of randomly oriented graphite particles in good contact might contribute as much as 10^{-5}Ωm×3mm/1mm/.1mm = 0.3Ω, six orders of magnitude too small to matter. This suggests in some sense that you could narrow the channel by six orders of magnitude, down to 1nm, before its resistance became important, but that is of course impractical.) Also, you probably want to avoid sharp corners to avoid ionizing the air or the paper.

Actually reaching such high frequencies might require you to extensively perforate the paper (without bridging the contacts on the two sides) or operate the whole device in a vacuum to avoid air resistance.

Dynamic deformation

Andrea Shepard points out that at high frequencies the paper will not behave as a rigid or quasi-rigid object; applying a force to one part of the paper will cause displacement to ripple out from that place as shear waves at a speed of sound in the paper, on the order of 1 km/s, which is to say 1 mm/μs. This would cause great slowness if the gate electrode did not overlap the "channel" and "contact" material, as it does in the above design sketch. Even in the above sketch, there is a distance of some 100 μm between them, and the compressive deformation will take on the order of 100 ns to propagate through it and 1 μs to approximate the behavior of a rigid object again. I hazard a guess that this will not be the limiting factor in the performance of these hypothetical devices.

The perfectly overlapping electrodes are somewhat questionable, though, for a different reason: electric fields can't normally penetrate

conductive masses such as the channel and contact material, because statically speaking the field induces a surface charge sufficient to cancel it. (This also creates a MOSFET-like "charge injection" problem.) So there might be no net force from the overlapping part of the gate, and the shear waves described above might carry almost all of the opening or closing force.

Scaling laws

If you scale the device down by a factor N, its area diminishes by N^2 while its plate separation diminishes by N, so the capacitance, the charge per volt, decreases by N. At the same time, though, the Coulomb force per nanocoulomb increases by N^2, so the force per volt *increases* by N. The mass that must be moved decreases by N^3, so the acceleration per volt increases by N^4, so the time to cover a given distance at a given voltage decreases by N^2. And the contact separation distance also decreases by N, which, with the same acceleration, would give you a $\sqrt{N}$ speedup. So you actually get an $N^2\sqrt{N}$ speedup, in a vacuum, at a constant voltage. Maybe more in the real world. Or less.

This suggests that, with crepe paper or Mylar or glass foil or something, you should be able to reach well into the megahertz with devices that are still individually visible to the naked eye, although perhaps a bit tricky to construct by hand.

In the limit you might have to diminish the voltage as the plate separation and radii of corners decrease to keep corona discharge and avalanche breakdown under control. If you decrease the voltage, you get proportionally less force, and need proportionally less charge. This means that the characteristic resistances stay the same, but in some sense the amplification factor drops. Like CMOS, these devices are capacitive loads, though very small ones; they have gain limited only by the leakage current, in this case further potentiated by the additional extremely high factor of gain at the threshold voltage between barely contacting and barely not contacting.

Multipole relays

In the above diagram, the vast majority of the area is occupied by the gate electrode; the channel occupies only a small part of the device. It might be worthwhile to run two or three channels across it to amortize the expense of the large gate over more channels. As calculated above, the channel itself doesn't reach a significant resistance at these voltages and currents until it's only 1nm wide (and 100µm tall, making it more like a graphite wall than a graphite trace, so maybe reducing it to 30µm × 30µm would be more reasonable). So you could run many channels across, controlling them all with a single gate voltage, and thus get many bits of switching out of a single device.

Topics

Shaped hammer face giant pressure

Kragen Javier Sitaker, 2019-11-10 (21 minutes)

As explained in Electric hammer (p. 1865) and Hammering toolhead (p. 3297), a hammer is a simple machine, although Archimedes and Galileo were unaware of this because they lacked the modern concept of kinetic energy; its mechanical advantage is limited only by the fact that its impact is not instantaneous.

It turns out that it is possible to substantially improve this mechanical advantage.

Basic hammering physics

As impact time approaches zero, the force and pressure of the impact approach infinity. Moreover, even during the impact, the deceleration of the hammer varies; as the deceleration approaches infinity, so does the force and pressure.

Thus you can beat an ice cube with a massive broomstick without damaging it, but a light whack with the bowl of a metal spoon can shatter it; and the tempered glass of a car window can withstand a baseball bat impact, but not a crackhead throwing a piece of sintered-aluminum-oxide spark plug at it.

Once the impact begins, the resulting deceleration does not affect the entire hammer at once, nor does the force affect the entire workpiece at once; instead, they travel through these bodies as sound waves. These waves reflect from other surfaces of the bodies in the same way that electrical waves reflect from the ends of transmission lines or sound waves reflect from the open ends of pipes, and may be focused or scattered in different parts of the solids. As a rule of thumb, the transmission line or hammer behaves like a rigid object or lumped-element node only over timescales about ten times the round-trip time of waves through it.

These sound waves are somewhat more complex than everyday sound waves. First, they involve both shear waves and compression waves, which travel at different speeds, each of which can produce the other under some circumstances. Second, their magnitude is often large enough to cause the usual linear approximations to break down; in particular, if there is a region within which the material stiffens with increased deformation, this can cause higher-amplitude waves to travel faster than, and overtake, lower-amplitude waves, and this can result in the propagation of self-sustaining discontinuous wavefronts through the material — in sound waves in air we call this a “shock wave” or a “detonation”, and a similar phenomenon in water channels is a “flash flood” or, on the beach, a “breaker”. Except when playing a percussion instrument, normally the point of using a hammer is to provoke some kind of nonlinear response from the thing you are hammering on, such as breaking or deforming plastically, so these phenomena are actually quite common in hammering.

Additionally, unless the impact is deep inside a hole in the workpiece, surface acoustic waves represent additional modes of vibration that may or may not carry significant energy dissipation.

Paddles

One obvious approach to shortening the impact time and thus increasing the force is to shorten the distance from the impact side of the hammer head to the back side, shortening the timescale over which it behaves like a rigid object. Taken to the logical extreme, this suggests using a hammer that is just a thin, flat sheet of material, which is brought into contact with the workpiece at the same time over its whole surface.

One use of this approach is the "slapper foil" detonator, which does in fact work, but other examples include flat paddles, slappers, or leather spanking straps used to punish children or excite masochists. These enable the development of sufficiently high impact forces to cause localized tissue damage despite low energies (typically on the order of a joule) distributed over large areas (typically on the order of 100 J/m^2). Even more prosaically, the same strategy is used for open-hand slapping, which the humans commonly use as a form of symbolic aggression to induce pain without any danger of more serious damage; and it is a factor in the effectiveness of spoon shattering of ice as well.

One difficulty of slapping things with paddles is that, in air, the large surface area of the paddle tends to accumulate an area of high-pressure air in front of it, soaking up most of the input energy. This can be desirable if the objective is merely to make a loud noise, but if maximal impact force is the objective, it is a drawback. Strategies to reduce this problem include putting air holes in the paddle, dividing the paddle into many strands (a "flogger"), or using a whip rather than a paddle.

Workpiece waveform propagation

During the impact, wavefronts of deformation are traveling through the workpiece as well as the hammer, and if the materials have reasonably similar acoustic impedances, they can even pass back and forth across the hammer-workpiece interface rather than being entirely reflected. Their particular behavior in the workpiece material depends on the properties of that material; for example, when a human is being slapped, I think much of the slap energy is dissipated by viscous and plastic behavior in the immediate neighborhood of the impact, so there is no tissue damage even millimeters away, while when using a bent wire to tap glass that has been scored on the other side, the stresses induced by the propagating wave are high enough to provoke the formation of a crack from the score.

(I don't really know if that's the reason slapping humans causes no damage except at submillimeter distances from the impact. Possibly dispersion also plays a role.)

Sensitivity to impact conditions

The direction in which each wavefront travels through the workpiece depends on the speed of each sound in the workpiece material and on the relative phase of the impact at each part of the workpiece surface; since the speeds of sound in solids are typically around 1km/s and the speeds of movement of hammers are typically around 1m/s, even fairly small rotations between mated hammer and workpiece surfaces can result in fairly large differences in propagation directions. Consider a flat hammer face that is 10mm across striking a

flat workpiece at 1 m/s with a compression-wave speed of sound of 1km/s with one milliradian of deviation from parallelism. So, when one side of the hammer face strikes, the other is still 10 microns away from striking; the contact area spreads across the hammer face gradually, as the hammer and workpiece deform, until reaching the opposite edge of the face 10 microseconds later. In those 10 microseconds, the waves from the initial impact have propagated 10 mm into the material, resulting in a planar wavefront propagating at a 60-degree angle from the surface normal, rather than in a nearly normal direction, as you might expect. The shear (or transverse or S) waves will propagate in a more nearly normal direction, since they propagate typically about 40% slower.

Since the "propagation time" of the contact area across the material is on the order of the speed of sound in either material, we should not expect the initial impact to alter the conditions much for later parts of the impact.

Shaped hammers

By shaping the hammer face to time different parts of the impact relative to one another, we can change the propagation characteristics of the wave in the material. For example, a convex hammer face, like the bowl of a spoon, or the traditional spherical blacksmithing hammers of Dablo, Burkina Faso, shown in Christopher Roy's documentary "From Iron Ore to Iron Hoe", will tend to produce a spherical wave that propagates outwards in all directions in the workpiece from a common center, and inwards toward a focus inside the hammer, like the convergent ultrasound waves used in sonoluminescence. A concave hammer face will tend to produce a convergent wave *within the workpiece*, but as noted above, small misalignments in the hammer angle will produce large deviations in the direction in which the wave propagates.

You might think that a potentially more predictable way to focus waves within the workpiece might be to shape the workpiece surface to be concave instead, but that just makes matters worse; the same small angles of hammer misalignments will still produce the same large angles of deviation, and additionally now lateral translation of the hammer blow by similar amounts will also produce those same deviation angles.

Outriggers

A better approach is to use three or more small "outriggers" sticking out of the sides of the hammer face to rotate it into submicron alignment with the workpiece surface by elastically deforming over hundreds of microseconds before the main impact. This approach can also compensate for some degree of imperfection in the workpiece surface itself, but not for micron-scale concavity or convexity over the impact area.

Phased arrays

Given a precise depth map of the workpiece surface, you could use a phased array of hammers in, for example, a hexagonal grid, each launched toward the surface to arrive at a time scheduled with submicrosecond jitter; this should enable the formation of a precisely customized wavefront within the workpiece material.

I had thought that perhaps you could use independent outriggers on a swarm of independent hammers (perhaps swung together using some kind of compliant coupling) to do this depth-mapping and timing mechanically rather than in software; the idea was that the outriggers would come into contact with the surface first, with the microsecond-scale jitter imposed by micron-scale surface roughness and milliradian-scale alignment imprecision, and would act to retard the subhammers that would otherwise make contact first and/or advance the subhammers that would otherwise make contact late, all by a few microseconds. But I don't see how to make that work mechanically.

Focusing hammers

Given a known hammer geometry, it should be possible to arrange for focusing of deformation waves at or near the hammer surface as well. For example, a convex hammer surface can produce an expanding spherical pressure wave inside the hammer upon the initial impact, with minimal dependence on the precise angle and position of the impact; if the back surface of the hammer is an ellipsoid with one focus at the virtual source of this wave, it can then reflect this spherical wave into a convergent spherical wave focused on a particular point or points. (See Caustics (p. 1619) for notes on designing light-reflective surfaces to form chosen focused patterns of intense light; this approach is also applicable to these sound waves.) The material at or near this focus point will experience stresses enormously greater than the stresses it would normally experience from an ordinary hammer blow, perhaps by two orders of magnitude or more. This point can be chosen to be within the hammer, at the interface with the workpiece, and probably most usefully, some distance inside the workpiece, thus limiting the damage to the hammer.

To be concrete, suppose we are using a steel hammer with a Young's modulus of 200 GPa (which Plastic cutters (p. 1074) and Wikipedia claim is typical of steels), weighing 1 kg, swung at 10 m/s, striking a flat piece of steel, and the hammer's circular striking surface is shaped to be precisely 20 microns higher in the center than at its edges, which are 40 mm apart. (At 7.9 g/cc, the hammer head then must average 101 mm long to weigh 10 kg.) If the hammer strikes straight on, the center of its face will strike 2 microseconds earlier than its edges do. Supposing the speed of sound in steel is 6 km/s, that means the center of the wave will have a phase advance of 2 microseconds and 12 mm relative to the edges. The 12 mm of steel compressed by 20 microns amounts to 1.7 millistrains and thus is under some 300 MPa of pressure. (XXX this is wrong but I don't know how to correct it yet; see below.)

A naive, untutored hammer of these dimensions used in this way will convert its 50 J of kinetic energy in a relatively straightforward fashion into 50 J of elastic deformation, with the pressure wave and other waves bouncing back and forth through the hammer and workpiece, augmenting the pressure by some 300 MPa on each bounce, over a timescale of a millisecond or two, during which time the hammer has come to a stop and begun to rebound violently — an average deceleration of about 5 km/s/s with a peak of about 10 km/s/s, producing about 10 kN of peak force. However, this only

amounts to about 8 MPa of pressure, so clearly I have miscalculated in a very significant way above; ASTM A36 steel actually has a yield stress of only 250 MPa, so 8 MPa is a lot more plausible than 250 MPa.

(This 10 kN amounts to a mechanical advantage of about 1000, depending on how much space you have to swing the hammer in: a constant 10 N accelerates the 1 kg hammer up to 10 m/s in one second, requiring 5 meters of swing.)

Suppose our super-hammer, with its ellipsoidal back face, can focus the initial impact wave to two orders of magnitude higher strains within the workpiece, or some 800 MPa. This is enough to provoke cold steel into plasticity, which will prevent the 800 MPa figure from actually being reached. A significant fraction of the impact energy will then be converted into heat within about a gram of steel inside the workpiece, heating it by some tens of degrees.

Applications

If it is possible to provoke fracture in the workpiece by this approach, which should be easily feasible for brittle materials, the available concentration factors should increase dramatically: the bubble of vacuum thus formed inside the workpiece would experience very large forces and temperatures as its walls crashed together on the following oscillation, again similar to sonoluminescence or to the collapse of the void when a stone falls into water or fluidized sand.

A particularly interesting situation is when the focus is at or near the opposite face of a flat workpiece, because it seems plausible that this mechanism could be used to pit it, eject material from it, heat-treat it, or cold-weld it to something else. This is a particularly appealing prospect for heat-treating inaccessible places like the interior surfaces of structural tubing after welding, although it might turn out to be impractical to achieve high enough concentration factors in ductile metals to reach the necessary temperatures.

Hammer materials

Better hammer materials might include things like copper and tungsten, with their lower speeds of sound, or even some kind of lead-filled steel foam. Graded-acoustic-impedance metamaterials should make it possible to reflectionlessly couple sonic energy into the workpiece even from a hammer (or other transducer) with a quite different acoustic impedance.

Multilayer slappers

Above I mentioned that one use of paddles is for exciting masochists. A device popular in this connection is a thing known as a "slapper": a double-layered paddle similar to the slapstick used in vaudeville or by Arlecchino. Shortly after the initial impact, the second layer of the paddle slaps into the first, producing an additional painful impact (as well as a much louder noise).

A potential advantage of this construction is that energy is not lost during the hammer swing to the air resistance of the second layer; the air between the layers moves along with the hammer, so it only absorbs any energy during the impact, when it is compressed between the layers to some degree, which is what gives the slapstick its loud

slap.

This could be extended to many layers, but without further elaboration, the layers will produce many separate impacts as they pile up on a pile on the workpiece, starting with the ones closest to the workpiece. This may be useful for producing some sort of vibration or recorded-sound reproduction, but by itself it will not produce any kind of unusual impact; dead-blow hammers filled with shot already do something similar.

What is needed for giant pressure is for the layers to collide very nearly simultaneously, but in the *reverse* order, starting with the layers furthest from the workpiece, with the compression wave propagating through each layer just in time for the layer to hit the next layer. For 1-mm-thick layers of steel, the ideal timing should be about 170 ns apart; with 100 such layers the overall sequence would total about 17 microseconds.

However, even if the impacts are simultaneous, this would only broaden the peak of the shock wave traveling through the stack of layers by those 17 microseconds, which is already about a factor of 50 to 100 better than a standard hammer, exceeding the physical limits of steel, though perhaps not those of some technical ceramics.

A variety of well-known pantograph-like linkages could be pressed into service for getting the stack of layers to uniformly expand and contract; moreover, they could be activated by the contact of "outriggers" with the workpiece surface, thus completing the stackup as the frontmost layer slams into the surface. The difficulty will be to get them to perform with enough precision to significantly exceed the impact force of an ordinary hammer.

By slightly thinning the sheets in their centers, the impact shock wave can be produced as a convergent spherical shock wave, focused as before on a chosen position within the workpiece. In this case, however, the precision of the hammer's positioning and angle are no longer so important, because the focusing of the shock wave does not depend on them, but only on the relative positioning of the layers in the stackup.

How much would you need to thin them? With the 40-mm-diameter, 101-mm-long geometry suggested above, and 6 km/s propagation velocity, the rearmost sheet would need a phase advance of about 2 mm and thus 333 ns at its outermost edge, if the focus is near the front of the hammer. If a 10 m/s impact velocity were split up evenly among 100 inter-sheet gaps, it would strike the sheet in front of it with a relative velocity of 0.1 m/s, so we're talking about etching it 33 nanometers deep in the middle to get a 330-ns phase difference. Other sheets would be etched progressively deeper, but we're still talking about a very difficult level of precision to reach any meaningful kind of focus.

By perforating the sheets it should be possible to provide low-resistance air paths from the center of the stack to outside of it, thus reducing the loss of potential impact energy to air compression. Simple aligned round perforations would provide lengthwise channels, but elongated and partly-aligned perforations could provide diagonal and radial air channels, which would be shorter and thus lower-resistance for many hammer geometries.

Topics

- Physics (p. 3632) (119 notes)
- Materials (p. 3560) (112 notes)
- Manufacturing (p. 3558) (50 notes)
- Mechanical things (p. 3569) (45 notes)
- Hammers (p. 3491) (3 notes)
- Acoustics (p. 3304) (2 notes)
- Steel

Lithium fission energy

Kragen Javier Sitaker, 2016-09-06 (updated 2019-09-16) (6 minutes)

From the WP article about Castle Bravo:

> a theoretical error made by designers... They considered only the lithium-6 isotope in the lithium deuteride secondary to be reactive; the lithium-7 isotope, accounting for 60% of the lithium content, was assumed to be inert. ... It was assumed that the lithium-7 would absorb one neutron, producing lithium-8 which decays (via beryllium-8) to a pair of alpha particles on a timescale of seconds—vastly longer than the timescale of nuclear detonation. However, when lithium-7 is bombarded with energetic neutrons, rather than simply absorbing a neutron, it captures the neutron and decays almost instantly into an alpha particle, a tritium nucleus, and another neutron.

The result was an extra 10 megatons of yield (4.2e16 joules, i.e. 42 petajoules) from the 10.7 tonnes of the device, most of which was presumably the lithium; this is about 6 or 7 petajoules per kg.

This reaction is remarkably similar to the "energy amplification" reaction used in thorium reactors: a neutron entering a mass of lithium-7 will stimulate a decay, which emits another neutron, which is either lost or stimulates another decay, and so on. We can infer that this does not produce a self-sustaining chain reaction because natural lithium still contains 92.5% lithium-7, and there is apparently no critical mass of lithium which results in a uranium-like or plutonium-like chain reaction.

(Presumably this is because in fact the decay to two helium nuclei via ^{8}Be mentioned above does happen even at high energies and consumes some or most of the neutrons. This was in fact the first artificial "splitting of the atom", carried out by Cockcroft and Walton in 1932 with a 700kV tube to accelerate protons — the cyclotron, necessary to reach energies per charge higher than your voltage, was only invented that same year in Berkeley.)

Lithium, however, is more appealing than thorium for a variety of reasons, occurs at 17 ppm in the crust, compared to thorium's 6 ppm, and there is an existing industry extracting over half a million tonnes of it from the crust per year, while thorium only has a few uses and is somewhat dangerous to refine because of its natural radioactivity. Thorium is imported into the US at a few tens of tonnes per year, and costs around US$100 per kilogram, while lithium is imported at a rate of some 3000 tonnes per year and costs around US$6000 per tonne, or US$6 per kilogram.

Lithium-6 also produces tritium when irradiated with a neutron, yielding 4.8 MeV (about 40 times less than the energy of an actinide fission), which works out to 77 TJ/kg, about 1% of the energy density inferred above for the Castle Bravo excess energy.

(The World Nuclear Association says that lithium-7 constitutes 92.5% of natural lithium and has a very low neutron cross section of 0.045 barns, and that on proton bombardment it fissions to 2He-4 yielding 17 MeV.)

US$6/kg and 6 PJ/kg conveniently give a fuel cost of US$1/PJ. By comparison, a common wholesale price for electrical energy is US$60/MWh (though this fluctuates minute by minute and goes negative most nights). This electrical price works out to US$16.67 per gigajoule, and thus US$16'666'667 per petajoule, some three

million times the price of the lithium fast fission energy. Even the lower energy of the ^{6}Li decay would be some thirty thousand times cheaper than grid power.

The tritium produced also decays energetically with a half-life of some 12 years; while this is too slow to be useful for a nuclear weapon, it could be useful for a nuclear reactor.

There's still an engineering question about how difficult it is to generate the neutrons to initiate the reaction; fusing lithium deuteride to generate the neutrons is clearly not an option in most cases, and you need to take into account the energy consumed by the particle accelerator. I suspect that the slow fission reaction by way of ^{8}Be mentioned above would also work, in which case the neutrons need not be energetic. Quite likely the usual approach of neutron spallation from a mercury target impacted by a proton beam would be adequate.

Lithium is particularly troublesome here because of its small cross-section — while ^{238}U has a thermal neutron capture cross-section of about 2 barns, I'm guessing both ^{6}Li and ^{7}Li have capture and fission cross-sections in the neighborhood of the capture cross section of hydrogen (0.2 barns), deuterium (0.0003 barns), or carbon (0.002 barns), which are another two to four orders of magnitude lower for fast neutrons. This means you might need 10 or 100 or 1000 times as much thickness of lithium absorbing the neutrons as you would for an actinide, or, alternatively, 10 or 100 or 1000 times as many neutrons.

The fission produces helium, a gas, and tritium, which can be gaseous if it reacts with other tritium rather than something else in the area; this suggests that the fuel should be kept liquid, perhaps as a molten salt, to allow the gas to bubble out instead of building up inside a solid.

Given the uniquely prominent position of ^{7}Li/^{8}Be fission in the history of nuclear physics research, it is inconceivable that this idea is original, so there is probably an obvious, well-known reason why it doesn't work; Szilárd surely tried it in the early 1930s, ten years before he and Fermi got a successful chain reaction going in uranium in 1942, but of course Szilárd didn't have access to modern fast electronics or even a cyclotron at the time.

Topics

- Materials (p. 3560) (112 notes)
- Energy (p. 3438) (63 notes)
- Nuclear (p. 3599) (3 notes)
- Lithium (p. 3553) (2 notes)
- Fission

Hot oil cutter

Kragen Javier Sitaker, 2016-08-16 (updated 2016-08-17) (8 minutes)

Some plastics, such as 6/6 nylon, are so resistant to abrasion that it is very difficult to machine them, but have very low viscosity once melted. You can cut nylon with an abrasive wheel, for example, but it destroys the wheel. Acetal (that is, polyoxymethylene or POM, aka Delrin) is popular not primarily because of its material properties — although they are quite good — but because it can be cut much more easily than most other thermoplastics.

Plastic foams such as Styrofoam can be cut using a hot wire, typically stainless steel, but this approach doesn't work for solid plastics; the molten plastic closes up the kerf behind the wire. Hot-wire cutting also typically suffers from poor temperature control, vaporizing and burning some of the plastic, which increases the risk of hazardous fumes.

Suppose that instead you had a narrow steel or aluminum pipe, for example 8 mm diameter with 0.5-mm-thick walls, heated to a precisely controlled temperature by pumping heated oil through it. The oil could, for example, be heated to 290° and pumped around a closed loop past an electrically heated aluminum heatsink, with the heat applied to the heatsink controlled by a thermostat measuring the temperature of the output oil. With this approach, most of the molten nylon can be made to run out of the kerf as the pipe advances through it, particularly if the nylon material is in a sheet of only a few millimeters thickness.

If we take 45–45 cal/g (188 kJ/kg) to be the heat of fusion — a bit less than that of paraffin — 271° to be the final melting point, and 1.21 g/cc to be the density of 6/6 nylon, from Starkweather, Noller, and Jones 1984 then melting a 9 mm kerf through a 20 mm thickness of nylon would require 41 J/mm to do just the melting. If we believe Engineering Toolbox's specific heat table, the heat capacity is 1.7 kJ/kg/K, so heating by 250 K is another 425 kJ/kg, for a total of 613 kJ/kg. This means that same 9 mm kerf at 20 mm thickness requires 135 J/mm. So cutting at a reasonable speed of 10 mm/s, neglecting conduction, requires 1300 W. Cutting at higher powers is more efficient, because you'll get a smaller error from neglecting conduction.

Polyvinyl chloride is another example of a plastic that can be easily cut with a hot object, but is hazardous with poor temperature control. If we believe the Polymer Science Learning Center's decomposition temperature table, PVC decomposes in the 200°–300° range but doesn't melt until 265°, and nylon 6,6 decomposes in the 310°–380° range, while PET melts at 268° and decomposes in the 283°–306° range.

Transferring 1500 W to oil through a heatsink probably requires a heatsink of some 100 mℓ capacity, so the total amount of oil needed for this tool is probably around 150 mℓ.

Avocado oil has a smoke point of 270°, which is probably high enough to melt nylon, but more normal cooking oils like soybean oil smoke at much lower temperatures like 238°. This suggests that if a nontoxic oil is to be used in this tool, it has to be medical-grade

paraffin or polydimethylsiloxane rather than any actually edible oil. I'm not certain that either of these, but especially medical-grade paraffin, will withstand such high temperatures; Engineering Toolbox suggests a limit of 149° for mineral oil and 260° for PDMS or other silicones, but I suspect that may just be the point where the oil ceases to lubricate. Some companies do sell high-temperature lubricants capable of lubricant use up to 270° and more; the fluorinated Krytox XHT supposedly doesn't degrade until 350° and doesn't corrode metals until 288°.

If the oil is heated to 290° and must not cool below 271°, we only have 19 K of sensible heat in which to store all the heat to be delivered to the nylon. If the oil has a heat capacity of 1.67 kJ/kg/K (according to Engineering Toolbox's table), that's only 32 kJ/kg, so we need 47 g/s of flow. At 0.8 g/cc, that's 59 mℓ/s. If a single-acting piston pump driven at 1500 rpm is driving this, the pump's displacement needs to be 2.35 cc.

In a 7-mm-diameter pipe, that's a rather shocking 1.5 m/s average linear speed. The Reynolds number is almost 11000. But according to an anonymous online calculator, the pressure drop from 200 mm of such a flow with .01 mm pipe roughness, if the fluid were water, would be only 11 mbar (1.1 kPa). This implies that only 65 mW of pumping power is needed, which seems surprisingly small to me, suggesting that maybe my pipe resistance calculation is wrong. Another random online calculator suggests that the head loss will be 294 mm, which would be 2.9 kPa, which is a little higher but still in the ballpark.

These high powers suggest that it might be desirable to power the tool directly with fire rather than electrically.

Alternative heat transfer fluids might include perfluorocarbons (like the Krytox XHT mentioned earlier), molten salts, and molten metals.

In particular, ordinary tin-lead 63/37 solder melts at 183°, doesn't boil until 1500°, and doesn't suffer chemical breakdown.

It does tend to dissolve metals — tin quite rapidly, of course, but also gold, silver, and copper at significant speeds, and even nickel some 25 times slower than copper. In the case of copper, the dissolution diminishes rapidly once the solder is saturated with dissolved copper (at a fairly low level), as is done in SAVBIT solder. Presumably this also applies to other metals it can dissolve, too. People add nickel to solder to keep it from dissolving iron. Phosphorus counteracts this effect and increases stainless steel erosion — in lead-free tin-copper solder.

There's debate about whether tin-lead solder is capable of even *joining* steel, which I thought might imply that it has a very hard time *dissolving* steel as well. Other alloys (lead-silver, cadmium- silver, tin-silver, and maybe tin-bismuth) supposedly work well for joining steel. But apparently tin-lead solder does work with steel if you use acid flux.

Molten solder in wave-soldering equipment is normally contained in stainless-steel equipment, which suffers erosion over a period of months, but this is at lower temperatures than what I'm discussing here. One study I found, though, found about 0.25 mm erosion depth on stainless steel 304 (and a bit less on 316) after 384 hours in a 350° lead-free solder, which seems slow enough that the tool could

still be useful.

Type metal is a variant that has the desirable property that it doesn't have a sudden change in volume when it melts. It's a tin-lead solder that also includes antimony; the traditional composition is 18% tin, 28% antimony, 54% lead, while the eutectic is 4% tin, 12% antimony, 84% lead, which melts at 240°. (Elemental antimony is fairly nontoxic, although its compounds are deadly, and its fumes are bad for you too.) Legend has it that type metal is very poor at dissolving iron.

Scrap type metal from Linotypes and the like is available on MercadoLibre at AR$55/kg (US$3.60/kg).

If using a coolant that melts above room temperature, then to keep the machine from freezing solid permanently the first time you turn it off, you could thread a resistance heating element all the way through the pipe, probably with insulation around it. That way, when you turn the element on, it will melt a path around it through the tube, allowing the coolant to begin to flow.

Topics

Mail reader

Kragen Javier Sitaker, 2018-04-27 (updated 2018-06-18) (7 minutes)

Several times I've started writing a mail reader. More than once I've gotten something I used for a while, then stopped.

Shape of the problem

I currently have 4.8 gigabytes of incoming email on adjuvant, which goes back to 2014, and a 1-gibibyte sample on my laptop. My laptop's SSD is capable of 77 MB/s, or 13 seconds per gigabyte.

Compression for faster access

lz4 can fail to compress 300 MB/s on its CPU, or compress a gibibyte of this mail by about 40% in about 20.0 seconds, of which only 3.1 are user time (suggesting the other 16.9 were waiting on the SSD). The 818-megabyte compressed file decompressed in 2.0 user seconds and 12.2 wallclock seconds. This suggests using LZ4 is capable of getting a modest storage speedup and saving about a gigabyte.

gzip -1 gets it down to 638 megabytes, and can decompress it in 18.3 seconds (16 seconds user, 2.5 seconds kernel). This suggests that gzip is actually nearly as fast as the SSD! pigz actually gets the time down to 11.6 seconds, suggesting that gzip -1 is actually *faster* than lz4 -1 here, because it gets substantially better compression and so can more effectively use the disk bandwidth.

lz4 -9 takes 78" to compress the file down to 779 megabytes; the resulting file can be decompressed in 3.1 seconds from warm cache, but reading it from the disk still takes 14 seconds.

pigz -9 compresses the file to 610 megabytes in 35", and the resulting file decompresses (with pigz) in 13.4" (10.6" user, 1.4" system).

In sum, compression might be some kind of a win, but probably not much of one, and it's easy for it to be a loss. lz4 -1 is probably pretty reasonable, but it's still slower than memcpy.

Linear access speeds

I'd like to be able to do full-text search on the mailbox in a reasonable amount of time. `grep ajgwio` takes 1.0 user seconds and 1.0 system seconds (and 14.4 wallclock seconds) to read through the gibibyte (already in memory!) and find no matches.

`grep '^From '` takes 110 milliseconds and finds 5838 messages, giving a mean message size of a bit under 200K. However, this is wrong; `grep -a '^From '` gives 41630 messages and takes 870 ms wallclock, 500 ms user and 480 ms kernel, providing a more reasonable message size of 26K.

`grep -a ajgw` yields some 25 random hits in 1300 ms wallclock, 820 ms user, 430 ms system. These are entirely in base64 data. `grep -a 'zines de villa urquiza'`, a literal string that occurs in four subject lines, takes 730 ms wallclock, 260 ms user, 450 ms system. It seems to be reading with read(), initially in 32768-byte chunks and eventually in half-mebibyte chunks, with some 17000 system calls, which seems like an inefficient approach in this case, but it does seem to be taking

substantially less time with the longer search string. And it avoids the problems of mmap(). Even `LANG=C fgrep -Ua 'We still need more volunteers to watch the Tor community and report'` still takes 620 ms wallclock, 230 ms user, 400 ms system.

Previous experiments on this laptop suggest 300 ns per system call plus 171 ps per byte, or 171 ns per kilobyte. This suggests copying the data takes 184 milliseconds, and doing the system call entries and exits should take about another 5 milliseconds. This is about half of the actual kernel time alone observed, so perhaps part of the problem is mere cache misses.

Previous experiments on this laptop have found a memcpy speed of about 3 gigabytes/second for files of a few kilobytes and 1.9 gigabytes/second for larger files, which seems like it could easily account for the extra runtime.

Probably this indicates that we could scan through data already in RAM at 6 gigabytes per second, but the laptop only has 4 gigabytes of RAM, so the mailbox won't fit in RAM. And reading it into RAM would take nearly a minute! Even with pigz, it would take 45 seconds. That's not an acceptable keystroke response time.

Storing an index in LevelDB

It's clear that I can't get by with just linear scans for searching, not with almost five gigabytes of email.

LevelDB on my laptop can insert 62 million small records into a LevelDB in 3'24". The result is 516 megabytes, or 8.3 bytes per record. I haven't measured how many queries per second it can handle but I assume it's pretty adequate.

I think this is probably inadequate performance for insertion, though, even assuming it remains constant for larger datasets. A somewhat typical message contains 250 words of body text (more or less 250 unique, too) in 5 kilobytes of message; perhaps a mean message contains 1000 unique words. Then LevelDB will be able to handle the insertion of only about 300 messages per second. The 41630 messages in my sample gibibyte would take a bit over two minutes; the whole mailbox would take about ten minutes.

Actually, you know what? Ten minutes to index my last three years of mail doesn't sound unusably slow. I should give it a try.

Syncing

I need to be able to get the mail from the server, which means running some kind of code on the server. I could upload a separate Python program, or I could send a bunch of shell commands, including over a pipe to an ssh process or something. Or I could do something really weird like upload bytecode for some kind of immutable virtual machine.

I also need some way to limit the amount of disk space I use locally.

Languages

For writing workflow stuff I probably want Python, or maybe JS or something. But for handling large volumes of data quickly, or for that matter for guaranteed UI responsiveness, I probably don't. Certainly not Python, not even PyPy I think.

I don't know how much of a hassle it would be to call LevelDB via ctypes or cffi. It's pretty easy in C++, and writing a little C-callable

C++ API that does what I need should be pretty easy, too. And then calling it from ctypes should be pretty easy.

...there's already a LevelDB binding for Python!

Topics

Simplifying code with concurrent iteration

Kragen Javier Sitaker, 2014-04-24 (2 minutes)

An observation from Unix is that it's often more convenient to structure a computation as a set of concurrent processes, each applying some simple transformation to one or more data streams, than to force everything into a single thread.

I'm observing this right now in a $work project that is scraping data from a server, respecting API rate limits, where the explicit state machine that handles API errors is gradually getting more and more complicated. In a well-structured program, this state machine would be written as a composition of primitive actions using sequences, conditionals, and iteration, with the state of the state machine encoded in the program counter; but since each API response from the server is handled by a callback function, this is not straightforward.

A simple way to handle this is to spawn off a separate thread that has a blocking proxy object for the API. When it calls a method on the blocking proxy object, this sends a message to the server (via the callback thread if necessary) and waits until the callback thread gets a response, either success, error, or if necessary, a timeout. Then I can write my retry and looping logic in this other thread using normal structured control flow and block-structured local variables, instead of a passel of boolean variables and suchlike.

This would work well on this project, where I have just one thread — although I'm probably not going to do it, because I don't expect to modify that code enough more to justify such a rewrite — but there are some practical memory difficulties with large numbers of threads.

OUTLINE

- fixed-size stacks (ref to previous thingy)
- Io language
- ref to previous thingy about low-level iteration
- threading to avoid buffering (connection to SAX)
- variable-size thread contexts
- merging with CPS sequence push

Topics

- Programming (p. 3658) (286 notes)
- Program design (p. 3654) (11 notes)
- Concurrency (p. 3386) (9 notes)

Hash feature detection

Kragen Javier Sitaker, 2015-09-17 (5 minutes)

Thinking about image registration and feature extraction, I thought: what about hashing?

In particular, think about a record player playing a scratched-up record, full of random crackles and pops, each due to a pit in the record surface left by the needle being shoved into it by long-ago dust. If you're looking for a particular location on the record, you can cross-correlate the audio stream with the specific sequence of crackles and pops at that location, and you are nearly sure to have a sharp peak at exactly the right point. This, in theory, allows you to determine the position of the turntable disc at that moment to within a fraction of a micron.

This won't work in its simple form if the playback speed is a little bit off — the distance between the pops in the pattern you're looking for will be off by more than the width of a pop, and so the correlation won't have a clear peak. But this can be remedied inexpensively, and save you the expense of running the full cross-correlation to boot.

Suppose our sample pattern is one second of audio. Take the strongest, say, five pops in that second. Typically, they'll be a decibel or so louder than the sixth-loudest pop, which is a decibel or so louder than the seventh-loudest, and so on. Now, take the last 1.5 seconds of audio and find the loudest eight pops in it, and try matching all 56 three-pop subsequences of those eight against all 10 three-pop subsequences of those five you're looking for, 560 comparisons in all. The candidate pops might not be in exactly the same order of loudness, because some of them might be very similar to the same level, and there is noise in the playback, but because the pops are so much louder than the rest of the recording, it's almost certain that there will be a match. And because you're matching a three-pop sequence, you have a ratio of intervals that have to match pretty closely between the pattern and the signal, according to some kind of time stretch. And that gives you a probably-unique candidate alignment on which to try the whole correlation.

You can adjust the values of 8, 5, 3, and 1.5, of course. Increasing 8 and 5 will increase the number of candidate alignments to try, thus increasing the chances of finding the correct alignment; increasing the value of 3 will increase the number at first and then decrease it. Increasing 1.5 will increase the amount of scale variation that can be tolerated, but at the risk of missing a match unless you increase 8, 5, 3, or all three.

(Of course, you can do much better if you have a pretty good idea where you are in the record groove to start with, because then you just need to adjust your estimate of the velocity up or down a bit as each pop or crackle goes by.)

In a sense, image feature detection algorithms are an attempt to superimpose crackles and pops onto images in such a way that their position, relative to the underlying image, remains precise in the face of a panoply of insults: from noise to 2-D rotation to scaling to 3-D rotation obscuring part of an object.

But in some sense, at least in the one-dimensional world where we

don't need to worry about rotations in two or three dimensions, it seems like it should be trivial to develop a robust feature-detection algorithm. Nearly any function should work as long as it's local, not totally disrupted by noise, and positions itself relatively precisely. For example, if we apply a couple of noise-reduction filters (maybe a median filter, if speckle noise is not actually signal as in the above example, and a Gaussian convolution) the last few peaks or zero-crossings should often be pretty stable; perhaps we can hash the distances between them, or the ratios of distances between them, perhaps quantized a bit. Then we can apply a filter to this stream of hash values: for example, ignore the hash unless its last 10 bits are zeroes. This should make our "pops" sufficiently sparse to be useful.

If we're trying to apply such an approach to images, it might be useful to do it by tracing paths in the image, using some kind of convergent approach, like maybe ascending the gradient of the absolute value of the gradient, thus seeking to walk along edges.

Topics

- Programming (p. 3658) (286 notes)
- Graphics (p. 3483) (91 notes)
- Digital signal processing (DSP) (p. 3419) (60 notes)
- Audio (p. 3331) (40 notes)
- Coregistration

Reconstructing a 3-D Lambertian surface from video with a moving light source

Kragen Javier Sitaker, 2016-09-16 (1 minute)

Lambertian surface reconstruction

To 3-d scan a Lambertian surface with a plain video camera and a light:

Move a point source of light near a Lambertian surface as you capture video from a camera stationary relative to the surface. Pixelwise differences between frames show the effect of removing the source from one frame and adding it to another. Hypothetically, ambient diffusion from other reflectors contributes negligibly to this difference, since the source is close to the surface being scanned. Given an estimated depth for each pixel and per-frame estimated light locations in 3-space, you can compute a simulated frame difference, which you can compare to the real frame difference to get an error. Each pixel of this error is a nonlinear (?) function of ten unknowns: the two x, y, z of the light locations, the estimated depth to that pixel, the surface gradient depth at that object point, and the estimated reflectance at that point. Given just one such error image, we can optimize the two light locations to minimize the error using gradient descent, holding the estimated depths fixed, or we can optimize the depths

Topics

- Graphics (p. 3483) (91 notes)
- Mathematical optimization (p. 3611) (29 notes)
- 3-D modeling (p. 3300) (9 notes)
- Cameras (p. 3364) (8 notes)
- Video (p. 3768) (7 notes)
- Structure from shading (p. 3709) (2 notes)

Hammering toolhead

Kragen Javier Sitaker, 2017-08-18 (6 minutes)

Using a hammer-and-chisel approach should enable new kinds of lightweight, portable, but slow CNC machine tools, by way of eliminating toolhead side forces during positioning.

One of the greatest challenges in precision machining is toolhead side forces. A 3-D printer can get by with a low-rigidity frame, open-loop control with stepper motors, and triangular-thread (V-thread) leadscrews in large part because the forces at its toolhead are so small. But cutting metal with edged tools or abrasives, even very sharp edged tools, produces substantial side forces, and these forces change rapidly and even unpredictably as contact is established or contact angles change. Consequently, precision machining invariably uses closed-loop control and massive structural members to achieve the desired stiffness, and usually also uses ballscrews or acme-thread leadscrews.

I've speculated previously that it with sufficiently rapid control systems, it will be possible to provide such stiffness "virtually", by countering a detected displacement with a larger opposing displacement in some actuator, effectively multiplying the rigidity of the frame by the ratio of displacements. But here I want to discuss a different approach, that of hammering.

A hammer or karate chop is a kind of simple machine: you apply a small force to the hammer over a large distance, and the hammer applies a large force to the work over a proportionally small distance. (Stevin and Galileo did not include the hammer in their list of simple machines, as they lacked, I think, the concept of kinetic energy.) Sufficiently rapid and precise feedback about the hammer's path allows you to control the point of impact precisely with low energy, but an alternative mechanism is the hammer-and-chisel mechanism used by an automatic center punch; this allows you to position the chisel as slowly and precisely as you like, then transmitting a hammer strike anywhere on the chisel head precisely to the chisel tip.

The hammer-and-chisel mechanism has been used for this purpose since the Paleolithic, when it enabled soft, squishy human hands to precisely shape rocks harder than steel.

At one point I was pretty skeptical of hammering on things to precisely shape or fix them, since there's inevitably a shock produced at the time of impact, and that shock can damage things as it propagates through the rest of the structure. But eventually I learned that hammering is, in many cases, safer and more precise than applying a large force more slowly through an elastic frame; applying a force F through a mechanism with stiffness k requires building up an energy $\frac{1}{2}F^2/k$. If the resistance diminishes once initial resistance is overcome, as in solid friction and many kinds of material fracture, this energy is released all at once in an uncontrolled fashion. By contrast, a hammer blow can have an arbitrarily small energy, and the energy *at a given force* is limited only by the stiffness of the chisel and hammer head themselves, which can be made orders of magnitude smaller than the stiffness of an entire frame.

This very feature is one of the reasons for the use of automatic

center punches: their hammer energy is the same on every stroke, so every divot has fairly precisely the same depth.

You could imagine a CNC machine that precisely repositioned a chisel several times a second under very low toolhead forces, striking the chisel with a hammer separately moved with higher, but still fairly low, forces, in order to cut or form a piece of material by applying a precisely controlled amount of energy with a very large force at a precisely controlled location. Wood chisels have been used with this technique by carpenters to make precise cuts in wood since the paleolithic.

As the hammer approaches the chisel, it must of course be held in some kind of mechanism to apply force to it, which means that it needs some non-negligible stiffness to the chisel and workpiece. However, at the point that it actually strikes the chisel, it is not necessary for it to be held with any stiffness in the axis in which it strikes the chisel. It could, for example, be sliding down a tube under its own momentum, or held with a negative-stiffness cancelation mechanism like that used to isolate the LIGO from vibration. Such an expedient could prevent the shock of impact from being transmitted back into the struture holding the hammer, which would introduce a vibrational impulse into the machine frame.

For metal cutting, it probably is not possible to reach the optimal cutting speed with this approach; recall, as cutting speeds go up, up to a point, tool life goes up too, because the machine is cutting hot metal which applies less force to the cutting edge, wearing it less. (I think it also is more efective at producing a built-up edge, which worsens cutting precision but decreases wear.) So we should expect that cutting tools used in this way will not last as long as cutting tools used in the traditional cutting mode on a lathe with continuous movement. Milling-machine cutting points (whether inserts or just flute edges) are typically used in a similar discontinuous-cutting mode, but the metal of the workpiece does not have time to cool completely between strokes.

Topics

Notes concerning “omq”

• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)

Notes concerning “3-D modeling”

• An algebraic approach to 3D geometry (p. 669) 2014-06-03 (updated 2014-06-29) (22 minutes)
• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Kinect modeling (p. 164) 2016-09-16 (1 minute)
• Reconstructing a 3-D Lambertian surface from video with a moving light source (p. 3296) 2016-09-16 (1 minute)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• Cloth structure from shading (p. 84) 2019-09-01 (2 minutes)

Notes concerning "3-D printing"

• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Notes on 3-D printing a mechanical LUT (p. 1326) 2014-04-24 (3 minutes)
• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• Slotted tape with skewed involute roulette bristles as an alternative to hose clamps and possibly screws (p. 395) 2014-07-02 (6 minutes)
• Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• A one-motor robot (p. 118) 2015-09-03 (13 minutes)
• Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) 2015-09-11 (updated 2019-12-20) (49 minutes)
• Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)
• Filling hollow FDM things with other materials (p. 2119) 2016-09-07 (5 minutes)
• 3-D printing glass with continuously varying refractive indices for optics without internal surfaces (p. 1156) 2016-10-06 (3 minutes)
• Jello printing (p. 2426) 2016-12-14 (8 minutes)
• Wang tile addition (p. 3201) 2017-02-16 (3 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• Vibratory powder delivery (p. 1747) 2017-02-25 (2 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Needle binder injection printing (p. 1492) 2019-08-05 (12 minutes)
• Sulfuric acid dehydration printing (p. 174) 2019-12-18 (updated 2019-12-19) (3 minutes)

Notes concerning “The Intel 8080 CPU”

• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)

Notes concerning “Aardappel”

• IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
• Programming paradigms for tiny microcontrollers (p. 2104) 2007 to 2009 (6 minutes)

Notes concerning “Acoustics”

• Audio tablet (p. 2178) 2019-09-28 (7 minutes)
• Shaped hammer face giant pressure (p. 3278) 2019-11-10 (21 minutes)

Notes concerning “Actors”

• Programming paradigms for tiny microcontrollers (p. 2104) 2007 to 2009 (6 minutes)
• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)

Notes concerning “Agriculture”

• Food miles imply insignificant energy costs (p. 2187) 2007 to 2009 (4 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Calculations about desalination in Israel (p. 2827) 2016-08-11 (3 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Low-carbohydrate diets are ecologically sustainable (p. 540) 2018-04-27 (2 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• Can artificially-lit vertical farming compete with greenhouses? (p. 2064) 2019-09-08 (12 minutes)

Notes concerning “Artificial intelligence”

• Some notes from playing 20q.net (p. 1246) 2007 to 2009 (22 minutes)
• Additive smoothing for Markov models (p. 2429) 2007 to 2009 (updated 2019-05-19) (11 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Interval filters (p. 1282) 2015-09-17 (2 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• Texture synthesis with spatial-domain particle filters (p. 857) 2016-10-06 (2 minutes)
• Gradient descent beyond machine learning (p. 2310) 2018-05-18 (2 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)

Notes concerning “Air quality”

• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Spark particulate sieve (p. 2047) 2016-10-06 (updated 2016-10-11) (7 minutes)
• Notes on a possible household air filter (p. 1961) 2018-05-05 (updated 2018-05-15) (10 minutes)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Algebra”

• An algebraic approach to 3D geometry (p. 669) 2014-06-03 (updated 2014-06-29) (22 minutes)
• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Time series data type (p. 304) 2016-08-26 (3 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• Changing the basis to a more expressive one with better affordances (p. 1389) 2016-09-29 (5 minutes)
• What is the type of lerp? (p. 1985) 2017-01-08 (5 minutes)
• Finite function circuits (p. 2050) 2017-02-16 (updated 2019-05-17) (29 minutes)
• Affine arithmetic optimization (p. 2801) 2017-07-19 (updated 2019-09-15) (3 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Fermat primes (p. 1451) 2019-07-07 (4 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)

Notes concerning “Algorithms”

• A cute algorithm for card-image templates (p. 1946) 2007 to 2009 (2 minutes)
• Additive smoothing for Markov models (p. 2429) 2007 to 2009 (updated 2019-05-19) (11 minutes)
• Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• Alastair thesis review (p. 1784) 2013-05-17 (1 minute)
• Use crit-bit trees as the fundamental string-set data structure (p. 1498) 2013-05-17 (3 minutes)
• Cycle sort (p. 2344) 2013-05-17 (1 minute)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• Constant-space grep (p. 296) 2014-02-24 (3 minutes)
• Simple persistent in-memory dictionaries with $\log^2$ lookups and logarithmic insertion (p. 3110) 2014-02-24 (6 minutes)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• Compression with second-order diffs (p. 2152) 2014-04-24 (3 minutes)
• Precisely how is 3 “optimal” for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• Polynomial-spline FIR kernels by integrating sparse kernels (p. 1819) 2014-04-24 (12 minutes)
• Randomizing delta-sigma conversion to eliminate aliasing (p. 2834) 2014-04-24 (7 minutes)
• Some speculative thoughts on DSP algorithms (p. 2590) 2014-04-24 (20 minutes)
• Rendering iterated function systems (IFSes) with interval arithmetic (p. 2433) 2014-09-02 (6 minutes)
• You can’t sort a file whose size is cubic in your RAM size in two passes, only quadratic (p. 2311) 2015-05-28 (5 minutes)
• Editor buffers (p. 1328) 2015-07-15 (updated 2015-09-03) (16 minutes)
• Cobstrings (p. 1312) 2015-08-21 (updated 2015-08-31) (5 minutes)
• Parsing a conservative approximation of a CFG with a FSM (p. 159) 2015-09-03 (7 minutes)
• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Ternary mergesort (p. 2161) 2015-09-03 (2 minutes)
• Very fast FIR filtering with time-domain zero stuffing and splines (p. 1146) 2015-09-03 (updated 2015-09-07) (13 minutes)
• Convolution surface plotting (p. 2264) 2015-09-03 (updated 2015-09-13) (2 minutes)

• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Interval filters (p. 1282) 2015-09-17 (2 minutes)
• Hash gossip exchange (p. 1470) 2015-11-19 (4 minutes)
• Improving LZ77 compression with a RET bytecode (p. 964) 2016-04-05 (updated 2016-04-06) (3 minutes)
• Trees as code (p. 2488) 2016-05-10 (4 minutes)
• Gaussian spline reconstruction (p. 656) 2016-06-05 (updated 2016-06-06) (5 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• Append only unique string pool (p. 1797) 2016-07-27 (2 minutes)
• Algorithm time capsule (p. 2263) 2016-08-11 (1 minute)
• Internal determinism (p. 2803) 2016-08-17 (2 minutes)
• Robust hashsplitting with sliding Range Minimum Query (p. 733) 2016-09-05 (7 minutes)
• Intro to algorithms (p. 2625) 2016-09-06 (4 minutes)
• An almost-in-place mergesort (p. 740) 2016-09-07 (5 minutes)
• Gradient rendering (p. 583) 2016-09-24 (11 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• What's the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• Chintzy depth of field (p. 629) 2016-10-27 (1 minute)
• Bitsliced operations with a hypercube of shuffle operations (p. 2363) 2016-11-30 (2 minutes)
• The paradoxical complexity of computing the top N (p. 1890) 2017-01-04 (7 minutes)
• Using Aryabhata's pulverizer algorithm to calculate multiplicative inverses in prime Galois fields and other multiplicative groups (p. 2255) 2017-01-06 (updated 2019-07-05) (4 minutes)
• Constant time sets for pixel painting (p. 1484) 2017-02-07 (2 minutes)
• Wang tile addition (p. 3201) 2017-02-16 (3 minutes)
• Set hashing (p. 2485) 2017-03-09 (9 minutes)
• Amnesic hash tables for stochastically LRU memoization (p. 502) 2017-04-03 (1 minute)
• Incremental persistent binary array sets (p. 1008) 2017-04-10 (4 minutes)
• String tuple encoding (p. 2419) 2017-04-28 (2 minutes)
• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Golomb-coding operands as belt offsets likely won't increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

• CIC-filter fonts (p. 1229) 2017-06-28 (1 minute)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Binary translation register maps (p. 2080) 2017-07-19 (1 minute)
• Double heap sequence (p. 2521) 2017-07-19 (2 minutes)
• Rasterizing polies (p. 2023) 2017-07-19 (3 minutes)
• A tournament to decide which notes to devote attention to polishing (p. 1195) 2017-07-19 (2 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)

• Multiplication with squares (p. 1983) 2017-07-19 (updated 2019-07-09) (5 minutes)
• Another candidate lightweight frequency tracking algorithm (p. 2069) 2017-08-18 (4 minutes)
• Cassette tape capacity (p. 2079) 2018-04-27 (1 minute)
• Incremental recomputation (p. 1184) 2018-04-27 (12 minutes)
• Gradient descent beyond machine learning (p. 2310) 2018-05-18 (2 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Top algorithms (p. 913) 2018-07-29 (4 minutes)
• Bit difference array (p. 1748) 2018-10-28 (10 minutes)
• Quintic upsampling of time-series with 1½ multiplies per sample (p. 2844) 2018-10-28 (2 minutes)
• Digital noise generators (p. 1137) 2018-10-28 (2 minutes)
• Cheap textures (p. 736) 2018-10-28 (updated 2019-05-05) (5 minutes)
• Dilating letterforms (p. 651) 2018-11-04 (15 minutes)
• Gauzy shit (p. 2985) 2018-11-04 (4 minutes)
• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)
• Improving Lua #L with incremental prefix sum in the ∧ monoid (p. 2008) 2018-12-18 (7 minutes)
• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Sample reversal (p. 1353) 2018-12-18 (updated 2019-01-17) (5 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216) 2019-01-04 (updated 2019-11-12) (26 minutes)
• Median filtering (p. 3155) 2019-01-17 (11 minutes)
• Hardware multiplication with square tables (p. 1886) 2019-02-08 (updated 2019-07-09) (4 minutes)
• Tabulating your top event of the month efficiently using RMQ algorithms (p. 619) 2019-03-19 (8 minutes)
• Accelerating Euler's Method on linear time-invariant systems by exponentiating matrices (p. 348) 2019-03-24 (updated 2019-04-02) (7 minutes)
• Karatsuba (p. 2090) 2019-04-20 (2 minutes)
• Granite texture (p. 1991) 2019-05-08 (updated 2019-05-09) (5 minutes)
• A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)

• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Midpoint method texture mapping (p. 1837) 2019-06-01 (3 minutes)
• Smooth hysteresis (p. 422) 2019-06-11 (13 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)
• Fermat primes (p. 1451) 2019-07-07 (4 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Complex linear regression (in the field ℂ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Query evaluation with interval-annotated trees over sequences (p. 1423) 2019-08-30 (updated 2019-09-03) (30 minutes)
• Differentiable neighborhood regression (p. 2944) 2019-08-31 (15 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• Cloth structure from shading (p. 84) 2019-09-01 (2 minutes)
• Processing halftoning (p. 915) 2019-09-01 (15 minutes)
• Debokehfication (p. 473) 2019-09-01 (updated 2019-09-12) (4 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)
• Nonlinear bounded leaky integrator (p. 3150) 2019-09-11 (8 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)
• Sparse sinc (p. 1880) 2019-09-15 (10 minutes)
• B-Tree ropes (p. 2762) 2019-09-24 (updated 2019-09-25) (19 minutes)
• Is there an incremental union find algorithm? (p. 1602) 2019-10-01 (8 minutes)
• Negative weight undirected graphs (p. 1590) 2019-11-01 (8 minutes)

• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)
• Interval raymarching (p. 1342) 2019-11-02 (updated 2019-11-10) (6 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)
• Approximate optimization (p. 517) 2019-11-13 (3 minutes)

- Magic sinewave filter (p. 200) 2019-12-17 (6 minutes)
- Sorting in logic (p. 498) 2019-12-28 (2 minutes)

Notes concerning “Aliasing”

• Randomizing delta-sigma conversion to eliminate aliasing (p. 2834) 2014-04-24 (7 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)
• Time domain analog chaos (p. 2198) 2018-10-28 (4 minutes)
• Antialiased line drawing (p. 1803) 2018-11-13 (updated 2019-09-01) (4 minutes)

Notes concerning “Alternate history”

• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Saturation detector (p. 1588) 2013-05-17 (3 minutes)
• An extremely simple electromechanical state machine (p. 50) 2014-04-24 (16 minutes)
• Alien game challenge (p. 2313) 2015-09-03 (6 minutes)
• Vitruvius could have taken photographs (p. 992) 2016-07-30 (1 minute)
• A plotter language of 9-bit bytes (p. 2154) 2017-05-29 (updated 2017-06-01) (11 minutes)
• Wang tile font (p. 1463) 2018-08-16 (5 minutes)
• Gradient pixels (p. 2202) 2018-08-16 (updated 2018-10-28) (9 minutes)
• Hall-effect Wheatstone bridges for impractical steampunk electronic logic gates (p. 2351) 2019-04-24 (2 minutes)

Notes concerning “Anatomy”

- Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
- Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)

Notes concerning “Android”

- Fast geographical maps on Android (p. 455) 2015-10-16 (9 minutes)
- Notes on local file browsing (p. 484) 2019-09-15 (updated 2019-09-28) (4 minutes)

Notes concerning “Anytime algorithms”

• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Affine arithmetic optimization (p. 2801) 2017-07-19 (updated 2019-09-15) (3 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)

Notes concerning “APL”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• A stack of coordinate contexts (p. 2987) 2007 to 2009 (9 minutes)
• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• APL with typed indices (p. 3264) 2013-05-17 (11 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)
• Observable transaction possibilities (p. 2086) 2019-06-15 (10 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)
• A formal language for defining implicitly parameterized functions (p. 144) 2019-09-05 (updated 2019-09-30) (29 minutes)

Notes concerning "Approximation"

• Differentiable neighborhood regression (p. 2944) 2019-08-31 (15 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)

Notes concerning "Archival"

• Copyright status of the Oxford English Dictionary: relevant data (p. 82) 2007 to 2009 (3 minutes)
• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Holographic archival (p. 766) 2014-04-24 (10 minutes)
• Offline datasets (p. 599) 2014-04-24 (15 minutes)
• Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
• The Dontmove archival virtual machine (p. 2113) 2014-06-29 (5 minutes)
• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• XCHG: An Archival Swap Machine (p. 2997) 2014-06-29 (7 minutes)
• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
• Quadratic opacity holograms (p. 3073) 2015-09-03 (7 minutes)
• Can you read the lunar lander's plaque from Earth? Or write a new one? (p. 125) 2015-09-03 (9 minutes)
• Viral wiki (p. 1024) 2015-10-15 (3 minutes)
• Designing an archival virtual machine (p. 3203) 2016-05-12 (6 minutes)
• Wikipedia people (p. 948) 2016-06-01 (6 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• Algorithm time capsule (p. 2263) 2016-08-11 (1 minute)
• Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
• Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
• Distributed computing environment (p. 776) 2017-07-19 (8 minutes)
• Piezoelectric engraving (p. 1070) 2017-07-19 (4 minutes)
• Notes on scraping the Codex Arundel to preserve it (p. 330) 2017-08-22 (1 minute)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)
• Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) 2018-11-09 (3 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)

• Raid zim (p. 253) 2019-01-17 (updated 2019-02-08) (1 minute)
• Progressive revealment crypto (p. 2068) 2019-04-10 (2 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Some extensions of William Beaty's scratch holograms (p. 2536) 2019-07-11 (9 minutes)

Notes concerning “Arduino”

• Randomizing delta-sigma conversion to eliminate aliasing (p. 2834) 2014-04-24 (7 minutes)
• Arduino radio (p. 169) 2016-07-30 (4 minutes)
• Could you do DDS of comprehensible radio signals with an Arduino? (p. 1829) 2017-03-31 (4 minutes)
• Urban autarkic network (p. 1026) 2018-04-27 (1 minute)
• Arduino curve tracer (p. 591) 2018-06-17 (10 minutes)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)

Notes concerning “Argentina”

• Smoky day (p. 3010) 2008-04-19 (4 minutes)
• Personal notes from 2013-06-06 (p. 2673) 2013-06-06 (updated 2014-04-24) (11 minutes)
• Some personal notes from February 2014 (p. 2134) 2014-02-13 (8 minutes)
• A Sunday in 2014 (p. 1089) 2014-02-24 (3 minutes)
• Jim Weirich’s death and my daily life (p. 829) 2014-04-24 (5 minutes)
• Notes from a Buenos Aires blackout, summer 2013-2014 (p. 267) 2014-04-24 (15 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• Ostinatto (p. 2780) 2014-04-24 (4 minutes)
• José, the Galician mover (p. 3076) 2015-11-09 (2 minutes)
• Phase-change heat reservoirs for household climate control (p. 2257) 2016-06-14 (updated 2016-06-17) (13 minutes)
• A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206) 2017-06-15 (4 minutes)
• Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)

Notes concerning “Arrays”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• A stack of coordinate contexts (p. 2987) 2007 to 2009 (9 minutes)
• Designing a Scheme for APL-like array computations, like Lush (p. 661) 2007 to 2009 (4 minutes)
• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• APL with typed indices (p. 3264) 2013-05-17 (11 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• Ndarray java (p. 2261) 2015-05-28 (1 minute)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Raggedcolumns (p. 2703) 2015-08-28 (3 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• JIT-compiling array computation graphs in JS (p. 155) 2017-07-19 (1 minute)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Composing code gobbets with implicit dependencies (p. 2437) 2018-04-27 (updated 2019-05-21) (3 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)
• Observable transaction possibilities (p. 2086) 2019-06-15 (10 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)

Notes concerning “Asciibetical homomorphism”

- String tuple encoding (p. 2419) 2017-04-28 (2 minutes)
- ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)

Notes concerning "Assembly language"

• Notes on reading eForth 1.0 for the 8086 (p. 541) 2007 to 2009 (5 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• Interesting features of the GNU assembler Gas (p. 3000) 2007 to 2009 (2 minutes)
• Maybe Counting Characters in UTF-8 Strings Isn't Fast After All! (p. 2992) 2007 to 2009 (15 minutes)
• Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation (p. 1508) 2013-05-17 (updated 2013-05-20) (21 minutes)
• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• The Dontmove archival virtual machine (p. 2113) 2014-06-29 (5 minutes)
• Implementing flatMap in terms of call/cc, as in Raph Levien's Io (p. 3248) 2015-09-03 (3 minutes)
• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• What's the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Golomb-coding operands as belt offsets likely won't increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

• Bit difference array (p. 1748) 2018-10-28 (10 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Assembler bootstrapping (p. 2922) 2019-07-18 (updated 2019-12-08) (16 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)
• Notes on Óscar Toledo G.'s bootOS (p. 277) 2019-10-07 (updated 2019-10-08) (28 minutes)
• Forth assembling (p. 940) 2019-12-08 (updated 2019-12-11) (18 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)
• My very first toddling steps in ARM assembly language (p. 1684)

2019-12-10 (updated 2019-12-13) (46 minutes)
• Can you eliminate backpatching? (p. 1769) 2019-12-17 (8 minutes)

Notes concerning “Astronomy”

• Solar system scale model (p. 2678) 2017-04-18 (1 minute)
• Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias (p. 309) 2019-04-10 (updated 2019-04-16) (2 minutes)

Notes concerning “Audio”

• Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
• Alastair thesis review (p. 1784) 2013-05-17 (1 minute)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• A unicast phased-array ultrasonic “radio” (p. 3077) 2013-05-17 (4 minutes)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• Randomizing delta-sigma conversion to eliminate aliasing (p. 2834) 2014-04-24 (7 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Would Synthgramelodia make a good base for livecoding music? (p. 2540) 2015-09-03 (8 minutes)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Hash feature detection (p. 3294) 2015-09-17 (5 minutes)
• Piano synthesis (p. 1655) 2015-09-17 (updated 2017-07-19) (6 minutes)
• Virtual instruments (p. 658) 2015-11-09 (3 minutes)
• How can we build an efficient microcontroller-based amplifier? (p. 2821) 2016-07-13 (5 minutes)
• What’s the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• The Magic Kazoo: a synthesizer you stick in your mouth (p. 1873) 2017-04-04 (updated 2019-05-12) (6 minutes)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• FM chirp sonar (p. 351) 2017-07-04 (1 minute)
• Cassette tape capacity (p. 2079) 2018-04-27 (1 minute)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Whistle detection (p. 357) 2018-06-06 (updated 2018-12-02) (18 minutes)
• Multitouch livecoding (p. 122) 2018-06-17 (1 minute)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Hacking a buck converter into a class-D amplifier? (p. 2109) 2018-06-30 (4 minutes)
• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)
• Sample reversal (p. 1353) 2018-12-18 (updated 2019-01-17) (5 minutes)
• Honk development (p. 1188) 2019-03-21 (2 minutes)
• Groping toward a high-efficiency speaker driver (p. 1212)

2019-04-02 (15 minutes)
• Audio video boustrophedon sync (p. 858) 2019-04-03 (2 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Measuring submicron displacements by pitch bending a slide guitar (p. 905) 2019-05-05 (18 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Audio tablet (p. 2178) 2019-09-28 (7 minutes)
• Examination of a shitty USB car charger (p. 286) 2019-10-24 (13 minutes)
• Hadamard rhythms (p. 831) 2019-11-01 (6 minutes)
• Audio logic analyzer (p. 1801) 2019-11-12 (3 minutes)
• Applying FM synthesis to natural sounds such as voices (p. 596) 2019-11-12 (2 minutes)
• English diphones (p. 2061) 2019-12-03 (5 minutes)
• Magic sinewave filter (p. 200) 2019-12-17 (6 minutes)

Notes concerning “Augmentation”

• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• Ice pants (p. 298) 2017-04-04 (updated 2019-01-22) (17 minutes)
• Commentaries on reading Engelbart’s “Augmenting Human Intellect” (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• Bokeh pointcasting (p. 92) 2019-09-08 (updated 2019-09-09) (16 minutes)
• Hearing aids for disability compensation, protection, and augmentation (p. 764) 2019-09-08 (updated 2019-09-09) (4 minutes)

Notes concerning “Autism”

- Autism is overfitting (p. 2692) 2019-08-31 (1 minute)
- Hearing aids for disability compensation, protection, and augmentation (p. 764) 2019-09-08 (updated 2019-09-09) (4 minutes)

Notes concerning “Automata theory”

• On hanging out with cranks (p. 1715) 2008-04 (4 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Parsing a conservative approximation of a CFG with a FSM (p. 159) 2015-09-03 (7 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• DReX and “regular string transformations”: would an RPN DSL work well? (p. 453) 2016-09-19 (3 minutes)
• One-line thoughts that don’t merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• Wang tile addition (p. 3201) 2017-02-16 (3 minutes)
• Tagging parsers (p. 208) 2018-11-23 (updated 2018-12-10) (9 minutes)
• What can you build out of 256-byte ROMs? (p. 1468) 2018-12-02 (1 minute)
• Turning a delay line into a counter with a FSM (p. 1680) 2018-12-10 (1 minute)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)

Notes concerning “Automatic differentiation”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Affine arithmetic has quadratic convergence when interval arithmetic has linear convergence (p. 1029) 2016-08-24 (updated 2017-01-18) (10 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)

Notes concerning “AVR microcontrollers”

• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• You’re pretty much fucked if you want to build an oscilloscope on the AVR’s ADC (p. 1269) 2013-05-17 (3 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• The Tinkerer’s Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Arduino radio (p. 169) 2016-07-30 (4 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• Augmenting a slow precise ADC with a sloppy fast high-pass filtered parallel ADC (p. 1469) 2017-03-20 (2 minutes)
• Loading new firmware on an AVR (p. 88) 2017-03-31 (3 minutes)
• Could you do DDS of comprehensible radio signals with an Arduino? (p. 1829) 2017-03-31 (4 minutes)
• Can you bitbang USB with an ATMega’s RC oscillator? (p. 1990) 2017-04-04 (1 minute)
• Minimum hardware and software to get a flexible notetaking device running (p. 535) 2017-04-28 (4 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Bench trash power supply (p. 1457) 2018-04-27 (9 minutes)
• Arduino curve tracer (p. 591) 2018-06-17 (10 minutes)
• Turning off the power supply for every sample to reduce noise (p. 239) 2018-06-18 (2 minutes)
• The TWI and I²C buses and better alternatives like CAN and RS-485 (p. 1638) 2018-06-28 (updated 2018-07-05) (24 minutes)
• The Adafruit Feather (p. 2966) 2018-06-30 (1 minute)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)
• Really simple lab power supply (p. 240) 2019-12-10 (7 minutes)

Notes concerning “Backtracking”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)

Notes concerning “Barcode”

- Barcode receipts (p. 2359) 2007 to 2009 (6 minutes)
- A 2007 overview of matrix barcodes (p. 1094) 2007 to 2009 (2 minutes)

Notes concerning “Batteries”

• Bike charger (p. 2099) 2014-04-24 (2 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Lithium battery welder (p. 2846) 2018-06-21 (updated 2019-01-22) (2 minutes)
• Balcony battery (p. 2377) 2019-02-11 (updated 2019-12-06) (6 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)
• Energy storage efficiency (p. 1300) 2019-07-30 (4 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Bicicleta”

• Bicicleta maps (p. 582) 2007 to 2009 (2 minutes)
• Nested inheritance (p. 340) 2007 to 2009 (2 minutes)
• OMeta contains Wadler's "Views" (p. 842) 2007 to 2009 (updated 2019-05-20) (13 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)

Notes concerning “Binary relations”

• Twingler (p. 2547) 2014-02-24 (7 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Time series data type (p. 304) 2016-08-26 (3 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)

Notes concerning “Binate”

- Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
- Term rewriting (p. 3221) 2017-07-19 (3 minutes)
- Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)

Notes concerning “Bitcoin”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• State of the world 2016 (p. 2973) 2016-09-05 (10 minutes)
• What are Bitcoin’s uses other than sidestepping the law? (p. 2159) 2019-03-11 (updated 2019-07-05) (6 minutes)
• Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) 2019-07-03 (6 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)

Notes concerning “BitTorrent”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Hash gossip exchange (p. 1470) 2015-11-19 (4 minutes)

Notes concerning “Bokeh”

• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Debokehfication (p. 473) 2019-09-01 (updated 2019-09-12) (4 minutes)
• Bokeh pointcasting (p. 92) 2019-09-08 (updated 2019-09-09) (16 minutes)

Notes concerning “Book reviews”

• Alastair thesis review (p. 1784) 2013-05-17 (1 minute)
• Review notes for Chris Anderson’s “Makers” (p. 1072) 2013-05-17 (5 minutes)
• The Stretch book is truly alien (p. 1888) 2018-11-27 (6 minutes)
• Commentaries on reading Engelbart’s “Augmenting Human Intellect” (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• A review of Wirth’s Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)

Notes concerning “Bootstrapping”

• IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
• The book written in itself (p. 2400) 2016-06-12 (updated 2016-06-14) (18 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Some notes on FullPliant and Pliant (p. 866) 2018-04-27 (9 minutes)
• Hand drawn font compositing (p. 1810) 2018-10-28 (2 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Assembler bootstrapping (p. 2922) 2019-07-18 (updated 2019-12-08) (16 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)

Notes concerning “Bottles”

• Storing dry bulk foods in used Coke bottles (p. 2145) 2012-10-15 (updated 2012-10-21) (5 minutes)
• Food storage (p. 2706) 2013-05-11 (updated 2013-05-17) (54 minutes)
• Bottle washing (p. 921) 2014-04-24 (7 minutes)
• Comparison of the PCO-1810 and PCO-1881 plastic bottlecap standards (p. 3223) 2014-05-25 (updated 2016-07-27) (2 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Hot water bottles (p. 3066) 2018-07-14 (4 minutes)
• Mayonnaise (p. 1320) 2019-03-19 (updated 2019-06-10) (10 minutes)

Notes concerning “The Brainfuck esolang”

• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• XCHG: An Archival Swap Machine (p. 2997) 2014-06-29 (7 minutes)
• Designing an archival virtual machine (p. 3203) 2016-05-12 (6 minutes)
• Options for bootstrapping a compiler from a tiny compiler using Brainfuck (p. 2958) 2017-07-19 (2 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)

Notes concerning “Browsers”

• What’s wrong with ../../? (p. 2304) 2007 to 2009 (2 minutes)
• Web prefetch (p. 3046) 2017-06-15 (1 minute)
• Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
• Dercuano search (p. 1532) 2019-05-16 (2 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)
• Notes on local file browsing (p. 484) 2019-09-15 (updated 2019-09-28) (4 minutes)

Notes concerning “BubbleOS”

• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Minimal GUI libraries (p. 663) 2015-11-14 (updated 2015-11-15) (5 minutes)
• The book written in itself (p. 2400) 2016-06-12 (updated 2016-06-14) (18 minutes)
• MiniOS (p. 1091) 2016-12-28 (updated 2017-01-03) (6 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Hand drawn font compositing (p. 1810) 2018-10-28 (2 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Dilating letterforms (p. 651) 2018-11-04 (15 minutes)
• Leconscrip: a family of JS subsets for BubbleOS (p. 2126) 2018-11-23 (2 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)
• Yeso notes (p. 2585) 2018-12-25 (updated 2019-01-01) (11 minutes)
• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• A review of Wirth’s Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)

Notes concerning “Buddhism”

Notes concerning “Building blocks”

• Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) 2016-08-17 (updated 2016-08-30) (24 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Extending Heckballs (p. 3239) 2019-11-26 (6 minutes)

Notes concerning “Business cards”

Notes concerning “Bytecode”

• Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)

Notes concerning “Bytestrings”

• String tuple encoding (p. 2419) 2017-04-28 (2 minutes)
• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)

Notes concerning “C ”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• Linear trees (p. 1811) 2016-05-19 (updated 2016-05-20) (6 minutes)

Notes concerning “C”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• I think I understand how to use libart's antialiased rendering API now (p. 409) 2007 to 2009 (10 minutes)
• C bad (p. 2832) 2007 to 2009 (4 minutes)
• Error Reporting is Part of the Programmer's User Interface (p. 2323) 2007 to 2009 (18 minutes)
• Developing Win32 programs on Debian (p. 609) 2007 to 2009 (12 minutes)
• Win32 startup (p. 2439) 2007 to 2009 (2 minutes)
• Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation (p. 1508) 2013-05-17 (updated 2013-05-20) (21 minutes)
• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• How to generate unique IDs for IMGUI object persistence? (p. 2035) 2014-09-02 (3 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• Cobstrings (p. 1312) 2015-08-21 (updated 2015-08-31) (5 minutes)
• An IMGUI-style drawing API isn't necessarily just immediate-mode graphics (p. 2671) 2015-09-03 (3 minutes)
• Linear trees (p. 1811) 2016-05-19 (updated 2016-05-20) (6 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• The history of NoSQL and dbm (p. 45) 2017-04-10 (16 minutes)
• Byte-stream pipe and antipipe façade objects for editor buffers (p. 950) 2017-04-10 (3 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Whistle detection (p. 357) 2018-06-06 (updated 2018-12-02) (18 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Yeso notes (p. 2585) 2018-12-25 (updated 2019-01-01) (11 minutes)
• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• Honk development (p. 1188) 2019-03-21 (2 minutes)
• Reducing the cost of self-verifying arithmetic with array operations

(p. 2205) 2019-06-23 (15 minutes)

Notes concerning “Caching”

• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• How can we usefully cache screen images for incrementalization? (p. 1077) 2013-05-17 (18 minutes)
• Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
• Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870) 2015-05-28 (updated 2016-06-22) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Memoize the stack (p. 2021) 2015-09-03 (5 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Amnesic hash tables for stochastically LRU memoization (p. 502) 2017-04-03 (1 minute)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• A minimal dependency processing system (p. 911) 2017-09-21 (3 minutes)
• Incremental recomputation (p. 1184) 2018-04-27 (12 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) 2018-11-09 (3 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)
• Dercuano backlinks (p. 475) 2019-05-22 (7 minutes)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)

Notes concerning “Calculators”

• drag-and-drop calculator for touch devices (p. 1045) 2015-09-03 (5 minutes)
• Interactive calculator 0 (p. 1453) 2015-09-17 (2 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• Usability of scientific calculators (p. 2379) 2016-09-29 (19 minutes)
• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Relational modeling (p. 1102) 2017-05-17 (updated 2017-06-01) (6 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Interactive calculator (p. 2771) 2018-04-26 (16 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• First impressions on using the μMath+ calculator program for Android (p. 195) 2019-05-21 (13 minutes)

Notes concerning “Calculus vaporis”

• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• Clanking replicators (p. 171) 2016-11-30 (3 minutes)

Notes concerning “Cameras”

• Vitruvius could have taken photographs (p. 992) 2016-07-30 (1 minute)
• Reconstructing a 3-D Lambertian surface from video with a moving light source (p. 3296) 2016-09-16 (1 minute)
• Servoing a V-plotter with a webcam? (p. 62) 2017-02-16 (3 minutes)
• Flying spot reilluminatable stage (p. 2358) 2017-05-15 (1 minute)
• Paper editing (p. 1761) 2017-06-15 (3 minutes)
• Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias (p. 309) 2019-04-10 (updated 2019-04-16) (2 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)
• Camera flash extrapolation (p. 2792) 2019-11-12 (6 minutes)

Notes concerning “Carbon capture”

- House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
- Scrubber mask (p. 90) 2019-05-08 (5 minutes)

Notes concerning “Cardboard”

• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• Cardboard furniture (p. 742) 2019-08-01 (updated 2019-08-11) (15 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29) (31 minutes)

Notes concerning “Cellular automata”

• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• Transmission line computer (p. 509) 2016-07-11 (updated 2019-07-23) (7 minutes)

Notes concerning “Caustics”

• Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
• You can’t construct optical systems with arbitrary light transfers, but you can do some awesome shit (p. 981) 2018-09-10 (11 minutes)
• Caustic simulation (p. 1454) 2018-09-10 (updated 2018-11-04) (2 minutes)
• Gauzy shit (p. 2985) 2018-11-04 (4 minutes)
• Caustic business card (p. 255) 2019-04-08 (3 minutes)
• Analemma sundial (p. 1955) 2019-07-05 (11 minutes)

Notes concerning “Cement”

• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Pythagorean cement pipes for your shower singing (p. 156) 2019-09-08 (updated 2019-09-09) (7 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)
• Berlinite gel (p. 848) 2019-12-14 (updated 2019-12-15) (10 minutes)

Notes concerning “Censorship”

• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Solving the incentive problem for censorship-resistant DHTs (p. 923) 2016-09-07 (updated 2019-05-21) (3 minutes)

Notes concerning “Ceramic”

• Improvising high-temperature refractory materials for pottery kilns (p. 2701) 2013-05-17 (4 minutes)
• Waterproofing (p. 429) 2015-09-03 (4 minutes)
• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)
• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• Clay fabrication objectives (p. 2111) 2017-01-16 (updated 2017-01-17) (3 minutes)
• Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• Vibratory powder delivery (p. 1747) 2017-02-25 (2 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• A brief note on autonomous cyclic fabrication systems from inorganic raw materials (p. 2965) 2018-04-27 (1 minute)
• Plasma glazing (p. 71) 2019-04-24 (1 minute)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Induction kiln (p. 2352) 2019-06-02 (19 minutes)
• Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033) 2019-07-16 (updated 2019-07-17) (28 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)

Notes concerning “Chat”

- Gaim group chat (p. 2677) 2007 to 2009 (3 minutes)
- Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
- Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)

Notes concerning “Chemistry”

• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Bottle washing (p. 921) 2014-04-24 (7 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) 2015-09-11 (updated 2019-12-20) (49 minutes)
• Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
• Vitruvius could have taken photographs (p. 992) 2016-07-30 (1 minute)
• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Immersion plating of copper on iron with blue vitriol (p. 1099) 2016-09-24 (8 minutes)
• Freeze distillation at 1 Hz (p. 2796) 2016-10-06 (5 minutes)
• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)
• Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
• Home dehumidifier (p. 3131) 2018-05-20 (updated 2019-04-02) (12 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• Needle binder injection printing (p. 1492) 2019-08-05 (12 minutes)
• Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) 2019-11-24 (2 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Berlinite gel (p. 848) 2019-12-14 (updated 2019-12-15) (10 minutes)
• Sulfuric acid dehydration printing (p. 174) 2019-12-18 (updated 2019-12-19) (3 minutes)

Notes concerning “Chifir”

• Designing an archival virtual machine (p. 3203) 2016-05-12 (6 minutes)
• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)

Notes concerning “China”

• What are Bitcoin’s uses other than sidestepping the law? (p. 2159) 2019-03-11 (updated 2019-07-05) (6 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)

Notes concerning “CIC or Hogenauer filters”

• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Sparse filters (p. 834) 2018-12-02 (4 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)

Notes concerning “Circle midpoint algorithm”

• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Midpoint method texture mapping (p. 1837) 2019-06-01 (3 minutes)

Notes concerning “Clay”

- Waterproofing (p. 429) 2015-09-03 (4 minutes)
- A brief note on autonomous cyclic fabrication systems from inorganic raw materials (p. 2965) 2018-04-27 (1 minute)
- Plasma glazing (p. 71) 2019-04-24 (1 minute)
- Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)

Notes concerning “Computer-mediated communication systems”

• Desbarrerarme: a UI for speaking to people (p. 186) 2015-09-03 (5 minutes)
• DHT bulletin board (p. 3001) 2016-09-07 (7 minutes)

Notes concerning “CoAP”

• What’s wrong with CoAP (p. 560) 2017-06-15 (3 minutes)
• Micro pubsub (p. 1504) 2017-06-15 (8 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Compressing REST transactions with per-connection state (p. 1117) 2018-04-27 (11 minutes)

Notes concerning “Code generation”

• Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation (p. 1508) 2013-05-17 (updated 2013-05-20) (21 minutes)
• Kernel code generation (p. 2302) 2019-07-02 (6 minutes)

Notes concerning “Communication”

• Cheap shit ultrawideband (p. 2776) 2013-05-17 (10 minutes)
• Time domain lightning triggering (p. 2534) 2013-05-17 (4 minutes)
• A unicast phased-array ultrasonic “radio” (p. 3077) 2013-05-17 (4 minutes)
• Constructing error-correcting codes using Hadamard transforms (p. 1474) 2013-05-17 (updated 2013-05-20) (22 minutes)
• Ultraslow radio for resilient global communication (p. 2071) 2013-05-17 (updated 2013-05-20) (26 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Arduino radio (p. 169) 2016-07-30 (4 minutes)
• License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162) 2016-09-19 (5 minutes)
• Could you do DDS of comprehensible radio signals with an Arduino? (p. 1829) 2017-03-31 (4 minutes)
• Dumb vocoder (p. 1391) 2017-05-10 (2 minutes)
• Interactive bandwidth (p. 779) 2017-08-03 (2 minutes)
• Cassette tape capacity (p. 2079) 2018-04-27 (1 minute)
• Urban autarkic network (p. 1026) 2018-04-27 (1 minute)
• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)
• Broadcast ECC with graceful degradation, or avoiding the cliff effect (p. 2045) 2018-12-18 (5 minutes)
• Free space optical coding gain (p. 1244) 2019-05-08 (updated 2019-05-09) (4 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated 2019-08-28) (37 minutes)
• Transmitting low-power TV signals around your house via RF modulation with an SDR (p. 1950) 2019-12-01 (6 minutes)

Notes concerning “Compilers”

• Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• Eur-Scheme: a simplified Ur-Scheme (p. 876) 2007 to 2009 (13 minutes)
• Simple system language (p. 1659) 2013-05-17 (7 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• Prototyping stuff (p. 176) 2016-08-11 (1 minute)
• What’s the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• Binary translation register maps (p. 2080) 2017-07-19 (1 minute)
• Options for bootstrapping a compiler from a tiny compiler using Brainfuck (p. 2958) 2017-07-19 (2 minutes)
• JIT-compiling array computation graphs in JS (p. 155) 2017-07-19 (1 minute)
• A review of Wirth’s Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)
• Dercuano grinding (p. 2475) 2019-10-01 (12 minutes)

Notes concerning “Compression”

• Git data (p. 2823) 2007 to 2009 (5 minutes)
• Git learnings (p. 3268) 2007 to 2009 (3 minutes)
• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Compression with second-order diffs (p. 2152) 2014-04-24 (3 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Improving LZ77 compression with a RET bytecode (p. 964) 2016-04-05 (updated 2016-04-06) (3 minutes)
• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Secure, self-describing, self-delimiting serialization for Python (p. 243) 2017-04-11 (8 minutes)
• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Golomb-coding operands as belt offsets likely won’t increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

• Compressing a screen update with a tree of dirty bits (p. 303) 2017-06-21 (1 minute)
• CIC-filter fonts (p. 1229) 2017-06-28 (1 minute)
• Compact code cpu (p. 397) 2017-07-19 (3 minutes)
• Cassette tape capacity (p. 2079) 2018-04-27 (1 minute)
• Compressing REST transactions with per-connection state (p. 1117) 2018-04-27 (11 minutes)
• Mail reader (p. 3290) 2018-04-27 (updated 2018-06-18) (7 minutes)

• Wang tile font (p. 1463) 2018-08-16 (5 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• Some musings on applying Fitts’s Law to user interface design and data compression (p. 1164) 2019-05-06 (updated 2019-05-09) (27 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)

• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)

Notes concerning “Concurrency”

• Enumerating binary trees and their elements (p. 1445) 2007 to 2009 (4 minutes)
• Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
• Simplifying code with concurrent iteration (p. 3293) 2014-04-24 (2 minutes)
• Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
• Low-cost green thread locks (p. 252) 2016-09-06 (2 minutes)
• Minimal transaction system (p. 2460) 2017-09-21 (5 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)
• Transactional event handlers (p. 139) 2019-01-24 (14 minutes)

Notes concerning “Constraint satisfaction”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Generalizing my RPN calculator to support refactoring (p. 969) 2016-10-17 (12 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Relational modeling (p. 1102) 2017-05-17 (updated 2017-06-01) (6 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• Designing a drawing editor for well-factored drawings (p. 2115) 2019-05-07 (9 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)

Notes concerning “Construction”

• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Improvising high-temperature refractory materials for pottery kilns (p. 2701) 2013-05-17 (4 minutes)
• Waterproofing (p. 429) 2015-09-03 (4 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Pythagorean cement pipes for your shower singing (p. 156) 2019-09-08 (updated 2019-09-09) (7 minutes)

Notes concerning “Content addressable”

• Git data (p. 2823) 2007 to 2009 (5 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• A simple content-addressable storage-server protocol (p. 774) 2015-09-03 (3 minutes)
• ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)
• Distributed computing environment (p. 776) 2017-07-19 (8 minutes)
• Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) 2018-11-09 (3 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)

Notes concerning “Control”

• Charge transfer servo (p. 3017) 2013-05-17 (2 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)
• Starfield servo (p. 1709) 2016-08-30 (updated 2018-11-07) (13 minutes)
• Jello printing (p. 2426) 2016-12-14 (8 minutes)
• Servoing a V-plotter with a webcam? (p. 62) 2017-02-16 (3 minutes)
• High-precision control of low-stiffness sytems with bounded-Q resonances (p. 1002) 2017-05-29 (updated 2017-06-01) (4 minutes)
• Sous vide (p. 2921) 2019-04-02 (2 minutes)
• Derivative based control (p. 2829) 2019-11-12 (6 minutes)

Notes concerning “Convolution”

• Some speculative thoughts on DSP algorithms (p. 2590) 2014-04-24 (20 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Very fast FIR filtering with time-domain zero stuffing and splines (p. 1146) 2015-09-03 (updated 2015-09-07) (13 minutes)
• Convolution surface plotting (p. 2264) 2015-09-03 (updated 2015-09-13) (2 minutes)
• Convolution with intervals (p. 1044) 2015-09-07 (1 minute)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Recurrent comb cascade (p. 483) 2018-11-09 (updated 2018-11-10) (2 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)
• Sparse sinc (p. 1880) 2019-09-15 (10 minutes)
• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)

Notes concerning “Cooking”

• Storing dry bulk foods in used Coke bottles (p. 2145) 2012-10-15 (updated 2012-10-21) (5 minutes)
• A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206) 2017-06-15 (4 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Deep freeze (p. 1465) 2017-08-22 (updated 2019-01-22) (7 minutes)

• Low-carbohydrate diets are ecologically sustainable (p. 540) 2018-04-27 (2 minutes)
• Home dehumidifier (p. 3131) 2018-05-20 (updated 2019-04-02) (12 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Mayonnaise (p. 1320) 2019-03-19 (updated 2019-06-10) (10 minutes)
• Sous vide (p. 2921) 2019-04-02 (2 minutes)
• Waterfryer (p. 1462) 2019-04-20 (1 minute)

Notes concerning “Cooling”

• Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Passivhaus seasonal thermal store (p. 1723) 2017-03-02 (updated 2017-03-07) (2 minutes)
• Passive dehumidifier (p. 3256) 2017-03-20 (14 minutes)
• Ice pants (p. 298) 2017-04-04 (updated 2019-01-22) (17 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Deep freeze (p. 1465) 2017-08-22 (updated 2019-01-22) (7 minutes)

• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Drone cutting (p. 1106) 2019-06-11 (12 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)
• Can artificially-lit vertical farming compete with greenhouses? (p. 2064) 2019-09-08 (12 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Copper plating”

- Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
- Immersion plating of copper on iron with blue vitriol (p. 1099) 2016-09-24 (8 minutes)
- Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
- Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)

Notes concerning “Copper”

- Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
- Immersion plating of copper on iron with blue vitriol (p. 1099) 2016-09-24 (8 minutes)
- Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
- Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)

Notes concerning “Cross compiling”

• Developing Win32 programs on Debian (p. 609) 2007 to 2009 (12 minutes)
• Win32 startup (p. 2439) 2007 to 2009 (2 minutes)

Notes concerning “Cryptography”

• How to use “correct horse battery staple” as an encryption key, including a recommended 4096-word list (p. 2522) 2014-04-24 (44 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• What’s wrong with CoAP (p. 560) 2017-06-15 (3 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)
• Progressive revealment crypto (p. 2068) 2019-04-10 (2 minutes)
• Human memorable secret sharing (p. 426) 2019-08-10 (2 minutes)
• Resurrecting duckling hashing (p. 3128) 2019-10-26 (updated 2019-11-10) (8 minutes)

Notes concerning “CSS”

• Dercuano stylesheet notes (p. 374) 2019-04-28 (updated 2019-05-09) (72 minutes)
• Dercuano formula display (p. 495) 2019-04-30 (5 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)

Notes concerning “Comma-separated values (CSV)”

- Storing CSV records in minimal memory in Java (p. 524) 2015-09-03 (6 minutes)
- Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)

Notes concerning “Databases”

• The Problem: Writing With One Access Pattern, Reading With Another (p. 3004) 2007 to 2009 (19 minutes)
• lattices, powersets, bitstrings, and efficient OLAP (p. 2345) 2014-04-24 (17 minutes)
• Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870) 2015-05-28 (updated 2016-06-22) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)
• Prototyping stuff (p. 176) 2016-08-11 (1 minute)
• ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• The history of NoSQL and dbm (p. 45) 2017-04-10 (16 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Database explorer (p. 225) 2017-06-20 (2 minutes)
• Compressing REST transactions with per-connection state (p. 1117) 2018-04-27 (11 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• Prolog table outlining (p. 2837) 2019-07-05 (11 minutes)
• Text relational query (p. 1223) 2019-08-28 (10 minutes)
• Query evaluation with interval-annotated trees over sequences (p. 1423) 2019-08-30 (updated 2019-09-03) (30 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)
• Bytecode pubsub (p. 205) 2019-12-04 (6 minutes)

Notes concerning “Dataflow”

- Tagged dataflow (p. 405) 2007 to 2009 (2 minutes)
- Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
- Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
- Incremental recomputation (p. 1184) 2018-04-27 (12 minutes)
- A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)

Notes concerning “Datasets”

• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• Handling Landsat 8 images in limited RAM with netpbm (p. 1884) 2014-04-24 (4 minutes)
• Offline datasets (p. 599) 2014-04-24 (15 minutes)
• Fast geographical maps on Android (p. 455) 2015-10-16 (9 minutes)
• Servoing a V-plotter with a webcam? (p. 62) 2017-02-16 (3 minutes)

Notes concerning “Death”

- High-risk behavior in context (p. 1485) 2007 to 2009 (5 minutes)
- Jim Weirich’s death and my daily life (p. 829) 2014-04-24 (5 minutes)

Notes concerning "Decentralization"

• Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Stuff I've posted to kragen-tol over the years about post-HTTP (p. 1815) 2014-02-24 (12 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• DHT bulletin board (p. 3001) 2016-09-07 (7 minutes)
• ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
• Solving the incentive problem for censorship-resistant DHTs (p. 923) 2016-09-07 (updated 2019-05-21) (3 minutes)
• Distributed computing environment (p. 776) 2017-07-19 (8 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• How inefficient is SNAT hole-punching via random port trials? (p. 1155) 2018-04-27 (2 minutes)
• What are Bitcoin's uses other than sidestepping the law? (p. 2159) 2019-03-11 (updated 2019-07-05) (6 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)

Notes concerning “Dependencies”

• Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
• Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)
• A minimal dependency processing system (p. 911) 2017-09-21 (3 minutes)
• Dercuano backlinks (p. 475) 2019-05-22 (7 minutes)

Notes concerning “Dercuano”

Notes concerning “Desalination”

• Calculations about desalination in Israel (p. 2827) 2016-08-11 (3 minutes)
• Fast sea salt evaporator (p. 1087) 2017-06-01 (3 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)

Notes concerning “Deterministic builds”

• Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)

Notes concerning “Deterministic computation”

• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• Prototyping stuff (p. 176) 2016-08-11 (1 minute)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)
• Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) 2018-11-09 (3 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)

Notes concerning “Distributed hash tables”

- DHT bulletin board (p. 3001) 2016-09-07 (7 minutes)
- Solving the incentive problem for censorship-resistant DHTs (p. 923) 2016-09-07 (updated 2019-05-21) (3 minutes)

Notes concerning "Digital fabrication"

• Polycaprolactone (p. 1813) 2007 to 2009 (3 minutes)
• Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)
• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Notes on 3-D printing a mechanical LUT (p. 1326) 2014-04-24 (3 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) 2015-09-11 (updated 2019-12-20) (49 minutes)
• Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
• Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)
• Sun cutter (p. 56) 2016-09-06 (9 minutes)
• Filling hollow FDM things with other materials (p. 2119) 2016-09-07 (5 minutes)
• Laser ablation of zinc or pewter for printed circuit boards (p. 2799) 2016-09-19 (4 minutes)
• 3-D printing glass with continuously varying refractive indices for optics without internal surfaces (p. 1156) 2016-10-06 (3 minutes)
• Hot air ice shaping (p. 864) 2016-10-06 (4 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Servoing a V-plotter with a webcam? (p. 62) 2017-02-16 (3 minutes)
• Wang tile addition (p. 3201) 2017-02-16 (3 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• Vibratory powder delivery (p. 1747) 2017-02-25 (2 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Hammering toolhead (p. 3297) 2017-08-18 (6 minutes)
• Laser cut next step (p. 824) 2018-04-27 (updated 2018-04-30)

(7 minutes)
• Friction-cutting plastic (p. 2412) 2019-02-25 (8 minutes)
• Single-point incremental forming of aluminum foil (p. 769) 2019-03-11 (updated 2019-06-10) (14 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Plastic cutters (p. 1074) 2019-04-20 (5 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Some extensions of William Beaty's scratch holograms (p. 2536) 2019-07-11 (9 minutes)
• Needle binder injection printing (p. 1492) 2019-08-05 (12 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)
• Hot lye granite cutting (p. 581) 2019-11-01 (2 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)
• Incremental roller comb forming (p. 3146) 2019-11-27 (4 minutes)
• Berlinite gel (p. 848) 2019-12-14 (updated 2019-12-15) (10 minutes)
• Nomadic furniture optimization (p. 1658) 2019-12-15 (2 minutes)
• Sulfuric acid dehydration printing (p. 174) 2019-12-18 (updated 2019-12-19) (3 minutes)

Notes concerning “Dijkstra”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• The Gelfand Principle, or how to choose educational examples (p. 1967) 2007 to 2009 (8 minutes)

Notes concerning “Displays”

• How can we take advantage of 16:9 screens for programming? (p. 1982) 2012-12-17 (2 minutes)
• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Electroluminescent matrix (p. 974) 2016-07-27 (2 minutes)
• Phosphorescent laser display (p. 1987) 2016-08-16 (8 minutes)
• Bubble display (p. 1542) 2017-01-24 (updated 2017-08-03) (1 minute)
• Sparkle wheel display (p. 1738) 2017-05-10 (6 minutes)
• A plotter language of 9-bit bytes (p. 2154) 2017-05-29 (updated 2017-06-01) (11 minutes)
• Oscilloscope screens (p. 578) 2018-06-05 (2 minutes)
• Microlens vibrating lightfield (p. 1219) 2018-07-14 (updated 2018-07-15) (11 minutes)
• Antialiased line drawing (p. 1803) 2018-11-13 (updated 2019-09-01) (4 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Transmitting low-power TV signals around your house via RF modulation with an SDR (p. 1950) 2019-12-01 (6 minutes)

Notes concerning “Dontmove”

Notes concerning “Drawing”

Notes concerning “Drying”

• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Passive dehumidifier (p. 3256) 2017-03-20 (14 minutes)
• Home dehumidifier (p. 3131) 2018-05-20 (updated 2019-04-02) (12 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033) 2019-07-16 (updated 2019-07-17) (28 minutes)

Notes concerning “Domain-specific languages”

- Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
- Would Synthgramelodia make a good base for livecoding music? (p. 2540) 2015-09-03 (8 minutes)
- Flexible text query (p. 2900) 2018-07-14 (4 minutes)
- Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)

Notes concerning “Digital signal processing (DSP)”

• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• Compression with second-order diffs (p. 2152) 2014-04-24 (3 minutes)
• Precisely how is 3 “optimal” for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• Polynomial-spline FIR kernels by integrating sparse kernels (p. 1819) 2014-04-24 (12 minutes)
• Randomizing delta-sigma conversion to eliminate aliasing (p. 2834) 2014-04-24 (7 minutes)
• Some speculative thoughts on DSP algorithms (p. 2590) 2014-04-24 (20 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Very fast FIR filtering with time-domain zero stuffing and splines (p. 1146) 2015-09-03 (updated 2015-09-07) (13 minutes)
• Bitstream dsp (p. 3153) 2015-09-03 (updated 2019-06-23) (3 minutes)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Hash feature detection (p. 3294) 2015-09-17 (5 minutes)
• Piano synthesis (p. 1655) 2015-09-17 (updated 2017-07-19) (6 minutes)
• Gaussian spline reconstruction (p. 656) 2016-06-05 (updated 2016-06-06) (5 minutes)
• Chintzy depth of field (p. 629) 2016-10-27 (1 minute)
• One-line thoughts that don’t merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• The Magic Kazoo: a synthesizer you stick in your mouth (p. 1873) 2017-04-04 (updated 2019-05-12) (6 minutes)
• Karplus–Strong PLLs (p. 2100) 2017-06-09 (1 minute)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• Another candidate lightweight frequency tracking algorithm (p. 2069) 2017-08-18 (4 minutes)
• Cassette tape capacity (p. 2079) 2018-04-27 (1 minute)
• Framed-belt DSP (p. 3095) 2018-04-27 (3 minutes)
• How can we do online pitch detection? (p. 1869) 2018-04-27 (updated 2018-04-30) (6 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05

(4 minutes)
• Whistle detection (p. 357) 2018-06-06 (updated 2018-12-02) (18 minutes)
• Word stream architecture (p. 2215) 2018-06-17 (13 minutes)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Quintic upsampling of time-series with 1½ multiplies per sample (p. 2844) 2018-10-28 (2 minutes)
• Recurrent comb cascade (p. 483) 2018-11-09 (updated 2018-11-10) (2 minutes)
• Sparse filters (p. 834) 2018-12-02 (4 minutes)
• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Sample reversal (p. 1353) 2018-12-18 (updated 2019-01-17) (5 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Median filtering (p. 3155) 2019-01-17 (11 minutes)
• Honk development (p. 1188) 2019-03-21 (2 minutes)
• Free space optical coding gain (p. 1244) 2019-05-08 (updated 2019-05-09) (4 minutes)
• Smooth hysteresis (p. 422) 2019-06-11 (13 minutes)
• Observable transaction possibilities (p. 2086) 2019-06-15 (10 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Phase relations (p. 2200) 2019-07-23 (updated 2019-07-24) (4 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• Processing halftoning (p. 915) 2019-09-01 (15 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)
• Debokehfication (p. 473) 2019-09-01 (updated 2019-09-12) (4 minutes)
• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)
• Nonlinear bounded leaky integrator (p. 3150) 2019-09-11 (8 minutes)
• Sparse sinc (p. 1880) 2019-09-15 (10 minutes)
• Audio tablet (p. 2178) 2019-09-28 (7 minutes)
• Comb filtering PWM (p. 40) 2019-10-28 (4 minutes)
• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)
• Camera flash extrapolation (p. 2792) 2019-11-12 (6 minutes)
• Applying FM synthesis to natural sounds such as voices (p. 596)

2019-11-12 (2 minutes)
• Transmitting low-power TV signals around your house via RF modulation with an SDR (p. 1950) 2019-12-01 (6 minutes)
• Magic sinewave filter (p. 200) 2019-12-17 (6 minutes)

Notes concerning “E-ink”

• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• Solar computer 2 (p. 414) 2017-07-19 (3 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)

Notes concerning “Error-correcting codes”

• Constructing error-correcting codes using Hadamard transforms (p. 1474) 2013-05-17 (updated 2013-05-20) (22 minutes)
• Practically decodable random error correction codes with popcount (p. 606) 2015-07-01 (updated 2015-09-03) (6 minutes)
• Broadcast ECC with graceful degradation, or avoiding the cliff effect (p. 2045) 2018-12-18 (5 minutes)
• Raid zim (p. 253) 2019-01-17 (updated 2019-02-08) (1 minute)

Notes concerning "Economics"

• Barcode receipts (p. 2359) 2007 to 2009 (6 minutes)
• Microfinance (p. 2875) 2007 to 2009 (6 minutes)
• Rich programmers (p. 805) 2007 to 2009 (4 minutes)
• Food miles imply insignificant energy costs (p. 2187) 2007 to 2009 (4 minutes)
• Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)
• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Dollar auctions and tournaments in human society (p. 884) 2013-05-17 (7 minutes)
• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Some personal notes from February 2014 (p. 2134) 2014-02-13 (8 minutes)
• What would a basic income guarantee for Argentina cost? (p. 2409) 2014-04-24 (7 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
• The internet is probably not going to collapse for economic reasons (p. 3194) 2016-09-06 (9 minutes)
• Solving the incentive problem for censorship-resistant DHTs (p. 923) 2016-09-07 (updated 2019-05-21) (3 minutes)
• Hybrid RAM (p. 2877) 2016-09-24 (5 minutes)
• Clanking replicators (p. 171) 2016-11-30 (3 minutes)
• Where did the Rubius comic book come from? (p. 1564) 2017-01-10 (4 minutes)
• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• The continuous-web press and the continuous press of the World-Wide Web (p. 1134) 2017-03-20 (6 minutes)
• A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206) 2017-06-15 (4 minutes)
• Japan can achieve energy autarky via solar energy, but not much before 2027 (p. 2819) 2017-07-12 (4 minutes)
• Low-carbohydrate diets are ecologically sustainable (p. 540) 2018-04-27 (2 minutes)
• 2017 [Provisional English translation of intercepted transmission] (p. 2192) 2018-04-27 (updated 2018-07-14) (13 minutes)
• Dutch auction raffle (p. 2474) 2018-06-05 (3 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)
• Ultralight tunnel personal rapid transit (p. 706) 2019-03-11 (15 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24)

(32 minutes)
• Better be weird (p. 1831) 2019-06-17 (updated 2019-06-24)
(9 minutes)
• Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) 2019-07-03 (6 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)
• Energy storage efficiency (p. 1300) 2019-07-30 (4 minutes)
• The fable of the specialized fox (p. 1076) 2019-08-17 (1 minute)

Notes concerning “Editors”

• The delta from QEmacs,with only 88 commands, to a usable Emacs, is small (p. 1543) 2013-05-17 (2 minutes)
• Editor buffers (p. 1328) 2015-07-15 (updated 2015-09-03) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Writing hypertext is still a pain (p. 715) 2016-02-18 (6 minutes)
• DReX and “regular string transformations”: would an RPN DSL work well? (p. 453) 2016-09-19 (3 minutes)
• One-line thoughts that don’t merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• Byte-stream pipe and antipipe façade objects for editor buffers (p. 950) 2017-04-10 (3 minutes)
• A sentence-granularity hypertext editor (p. 2290) 2018-04-27 (4 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Fencepost cognitive interface errors in text editing (p. 993) 2019-04-24 (24 minutes)

Notes concerning “Education”

• The Gelfand Principle, or how to choose educational examples (p. 1967) 2007 to 2009 (8 minutes)
• Intro to algorithms (p. 2625) 2016-09-06 (4 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• The ultimate capacity of human memory if spaced-practice memorization works as advertised (p. 2442) 2017-01-04 (updated 2017-01-08) (14 minutes)
• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Replicating education (p. 557) 2017-07-19 (7 minutes)
• An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)

Notes concerning “Electrochemical machining”

- A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
- Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
- Caustic business card (p. 255) 2019-04-08 (3 minutes)

Notes concerning “Electrolysis”

• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
• Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
• Caustic business card (p. 255) 2019-04-08 (3 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Electronics”

• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder (p. 527) 2013-05-17 (5 minutes)
• Charge transfer servo (p. 3017) 2013-05-17 (2 minutes)
• Cheap shit ultrawideband (p. 2776) 2013-05-17 (10 minutes)
• Harvesting energy with a clamp-on transformer (p. 1952) 2013-05-17 (7 minutes)
• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• You’re pretty much fucked if you want to build an oscilloscope on the AVR’s ADC (p. 1269) 2013-05-17 (3 minutes)
• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Saturation detector (p. 1588) 2013-05-17 (3 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• The Tinkerer’s Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• lattices, powersets, bitstrings, and efficient OLAP (p. 2345) 2014-04-24 (17 minutes)
• Precisely how is 3 “optimal” for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• An extremely simple electromechanical state machine (p. 50) 2014-04-24 (16 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Bitstream dsp (p. 3153) 2015-09-03 (updated 2019-06-23) (3 minutes)
• Virtual instruments (p. 658) 2015-11-09 (3 minutes)
• Making a logic gate of a single MOSFET (p. 167) 2016-06-28 (5 minutes)
• Transmission line computer (p. 509) 2016-07-11 (updated

2019-07-23) (7 minutes)
• How can we build an efficient microcontroller-based amplifier? (p. 2821) 2016-07-13 (5 minutes)
• Jellybean ICs 2016 (p. 817) 2016-07-14 (updated 2019-05-05) (17 minutes)
• How would you maximize the energy density of a capacitor? (p. 42) 2016-07-27 (5 minutes)
• Electroluminescent matrix (p. 974) 2016-07-27 (2 minutes)
• Matrix memory (p. 503) 2016-07-27 (1 minute)
• Arduino radio (p. 169) 2016-07-30 (4 minutes)
• Coinductive keyboard (p. 1893) 2016-07-30 (4 minutes)
• Solar-cell Geiger counters (p. 3241) 2016-07-30 (1 minute)
• Transmission line diode computation (p. 3090) 2016-07-30 (3 minutes)
• Argentine oscilloscope pricing 2016 (p. 1965) 2016-08-16 (4 minutes)
• Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times? (p. 1027) 2016-09-06 (3 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• Circuit notation (p. 1161) 2016-09-08 (updated 2017-04-18) (7 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162) 2016-09-19 (5 minutes)
• Laser ablation of zinc or pewter for printed circuit boards (p. 2799) 2016-09-19 (4 minutes)
• Hybrid RAM (p. 2877) 2016-09-24 (5 minutes)
• Marking metal surfaces with arcs (p. 1993) 2016-10-06 (4 minutes)
• Spark particulate sieve (p. 2047) 2016-10-06 (updated 2016-10-11) (7 minutes)
• Current hardware trends tend toward raytracing (p. 1351) 2016-10-07 (4 minutes)
• Nonlinear differential amplification (p. 2949) 2016-12-14 (2 minutes)
• My attempt to learn about jellybean electronic components (p. 1974) 2017-02-08 (updated 2019-09-29) (22 minutes)
• Finite function circuits (p. 2050) 2017-02-16 (updated 2019-05-17) (29 minutes)
• Non-inverting logic (p. 861) 2017-02-18 (updated 2019-07-20) (8 minutes)
• A 7-segment-display font with 68 glyphs (p. 1798) 2017-02-21 (4 minutes)
• Lab power supply (p. 2421) 2017-02-21 (updated 2018-06-18) (17 minutes)
• Vibratory powder delivery (p. 1747) 2017-02-25 (2 minutes)
• Augmenting a slow precise ADC with a sloppy fast high-pass filtered parallel ADC (p. 1469) 2017-03-20 (2 minutes)

• Loading new firmware on an AVR (p. 88) 2017-03-31 (3 minutes)
• Could you do DDS of comprehensible radio signals with an Arduino? (p. 1829) 2017-03-31 (4 minutes)
• Can you bitbang USB with an ATMega's RC oscillator? (p. 1990) 2017-04-04 (1 minute)
• The Magic Kazoo: a synthesizer you stick in your mouth (p. 1873) 2017-04-04 (updated 2019-05-12) (6 minutes)
• Disk oscilloscope (p. 713) 2017-04-10 (updated 2017-06-20) (3 minutes)
• TV oscilloscope (p. 1253) 2017-04-10 (updated 2017-06-20) (4 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Laser printer oscilloscope (p. 449) 2017-04-18 (updated 2017-06-20) (2 minutes)
• Minimum hardware and software to get a flexible notetaking device running (p. 535) 2017-04-28 (4 minutes)
• Can a simple nonlinear VCO enable super cheap oscilloscopes? (p. 1357) 2017-05-04 (updated 2017-05-10) (5 minutes)
• Dumb vocoder (p. 1391) 2017-05-10 (2 minutes)
• Adding GPIO lines over USB with a Saleae logic analyzer (p. 628) 2017-05-10 (1 minute)
• VCR oscilloscope (p. 213) 2017-05-10 (updated 2017-06-20) (2 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• CCD oscilloscope (p. 1861) 2017-06-20 (updated 2017-07-04) (7 minutes)
• A stack of stacks for simple modular electronics (p. 1779) 2017-06-27 (5 minutes)
• Constant current switching capacitor charging (p. 605) 2017-07-19 (1 minute)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Bench trash power supply (p. 1457) 2018-04-27 (9 minutes)
• Cassette tape capacity (p. 2079) 2018-04-27 (1 minute)
• Earring computer (p. 2505) 2018-04-27 (1 minute)
• Urban autarkic network (p. 1026) 2018-04-27 (1 minute)
• Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642) 2018-04-30 (updated 2018-07-05) (29 minutes)
• Arduino curve tracer (p. 591) 2018-06-17 (10 minutes)
• Diode logic (p. 272) 2018-06-17 (16 minutes)
• Resistor assortment (p. 310) 2018-06-17 (4 minutes)
• Snap logic (p. 2580) 2018-06-17 (3 minutes)
• Word stream architecture (p. 2215) 2018-06-17 (13 minutes)
• Transistors vs. Microcontrollers (p. 2918) 2018-06-17 (updated 2018-07-05) (8 minutes)
• Turning off the power supply for every sample to reduce noise (p. 239) 2018-06-18 (2 minutes)
• Lithium battery welder (p. 2846) 2018-06-21 (updated 2019-01-22) (2 minutes)
• The TWI and I²C buses and better alternatives like CAN and RS-485 (p. 1638) 2018-06-28 (updated 2018-07-05) (24 minutes)

• Hacking a buck converter into a class-D amplifier? (p. 2109) 2018-06-30 (4 minutes)
• The Adafruit Feather (p. 2966) 2018-06-30 (1 minute)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Electric hammer (p. 1865) 2018-07-02 (updated 2018-07-05) (14 minutes)
• Capacitors: some notes on tradeoffs (p. 134) 2018-07-05 (5 minutes)
• Can you turbocharge the STM32 ADC to build an oscilloscope? (p. 137) 2018-07-14 (5 minutes)
• Microlens vibrating lightfield (p. 1219) 2018-07-14 (updated 2018-07-15) (11 minutes)
• Comparable counters (p. 2441) 2018-08-16 (1 minute)
• Notes on circuitry for the Nutra seed activator (p. 3099) 2018-08-16 (20 minutes)
• Gradient pixels (p. 2202) 2018-08-16 (updated 2018-10-28) (9 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Time domain analog chaos (p. 2198) 2018-10-28 (4 minutes)
• The details of the GPU in this laptop (p. 2970) 2018-10-29 (2 minutes)
• Performance properties of sets of bitwise operations (p. 636) 2018-11-06 (updated 2018-11-07) (16 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Parallel register file (p. 2952) 2018-11-27 (2 minutes)
• The Stretch book is truly alien (p. 1888) 2018-11-27 (6 minutes)
• What can you build out of 256-byte ROMs? (p. 1468) 2018-12-02 (1 minute)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• Hardware multiplication with square tables (p. 1886) 2019-02-08 (updated 2019-07-09) (4 minutes)
• Groping toward a high-efficiency speaker driver (p. 1212) 2019-04-02 (15 minutes)
• Paper/foil relays (p. 3273) 2019-04-02 (updated 2019-10-23) (13 minutes)
• Macroscopic capacitive DLP (p. 1834) 2019-04-08 (1 minute)
• Hall-effect Wheatstone bridges for impractical steampunk electronic logic gates (p. 2351) 2019-04-24 (2 minutes)
• Measuring submicron displacements by pitch bending a slide guitar (p. 905) 2019-05-05 (18 minutes)
• Free space optical coding gain (p. 1244) 2019-05-08 (updated 2019-05-09) (4 minutes)
• A phase-change soldering iron (p. 2270) 2019-05-08 (updated 2019-05-09) (14 minutes)
• Inductor thermocouple sensing (p. 2037) 2019-06-01 (21 minutes)
• Induction kiln (p. 2352) 2019-06-02 (19 minutes)
• How to get 6 volts out of a 7805, and why you shouldn't (p. 537)

2019-06-08 (updated 2019-06-10) (8 minutes)
• Smooth hysteresis (p. 422) 2019-06-11 (13 minutes)
• Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033) 2019-07-16 (updated 2019-07-17) (28 minutes)
• Printed circuits on fired-clay ceramic (p. 960) 2019-08-13 (11 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated 2019-08-28) (37 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)
• Capacitive droppers and transformerless power supplies (p. 887) 2019-09-18 (11 minutes)
• Audio tablet (p. 2178) 2019-09-28 (7 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Examination of a shitty USB car charger (p. 286) 2019-10-24 (13 minutes)
• Comb filtering PWM (p. 40) 2019-10-28 (4 minutes)
• Audio logic analyzer (p. 1801) 2019-11-12 (3 minutes)
• Nonconductive relays (p. 3262) 2019-11-12 (3 minutes)
• Backwards cockcroft walton (p. 2282) 2019-12-01 (2 minutes)
• High temperature semiconductors (p. 2436) 2019-12-01 (2 minutes)

• Transmitting low-power TV signals around your house via RF modulation with an SDR (p. 1950) 2019-12-01 (6 minutes)
• Really simple lab power supply (p. 240) 2019-12-10 (7 minutes)

Notes concerning “Emacs”

- Emacs22 annoyances (p. 3197) 2007 to 2009 (4 minutes)
- Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
- Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
- Fencepost cognitive interface errors in text editing (p. 993) 2019-04-24 (24 minutes)

Notes concerning “Email”

• Why Thunderbird is inadequate for opening a 7-gigabyte mbox (p. 980) 2007 to 2009 (2 minutes)
• Giving Golang a second look for writing a mailreader (in 2012) (p. 290) 2012-12-17 (updated 2013-05-17) (2 minutes)
• Desbarrerarme: a UI for speaking to people (p. 186) 2015-09-03 (5 minutes)
• Service-oriented email (p. 1302) 2017-06-20 (updated 2017-06-21) (15 minutes)
• Mail reader (p. 3290) 2018-04-27 (updated 2018-06-18) (7 minutes)

Notes concerning “Energy harvesting”

• Harvesting energy with a clamp-on transformer (p. 1952) 2013-05-17 (7 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Mic energy harvesting (p. 2583) 2016-09-07 (updated 2016-09-08) (5 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• Constant current switching capacitor charging (p. 605) 2017-07-19 (1 minute)
• Solar computer 2 (p. 414) 2017-07-19 (3 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated 2019-08-28) (37 minutes)
• Capacitive droppers and transformerless power supplies (p. 887) 2019-09-18 (11 minutes)

Notes concerning “Energy”

• Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
• A comparison of prices for different forms of energy (p. 3097) 2007 to 2009 (2 minutes)
• Food miles imply insignificant energy costs (p. 2187) 2007 to 2009 (4 minutes)
• The economics of solar energy (p. 1175) 2008 (27 minutes)
• Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)
• Pensamientos acerca de diseñar un calefón solar (p. 117) 2012-10-15 (2 minutes)
• Más pensamientos acerca de diseñar un calefón solar (p. 1713) 2012-10-15 (5 minutes)
• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder (p. 527) 2013-05-17 (5 minutes)
• Bike charger (p. 2099) 2014-04-24 (2 minutes)
• Notes from a Buenos Aires blackout, summer 2013-2014 (p. 267) 2014-04-24 (15 minutes)
• The future of the human energy market (2014) (p. 1846) 2014-04-24 (19 minutes)
• Fukushima leak (p. 2905) 2014-04-24 (6 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Jellybean ICs 2016 (p. 817) 2016-07-14 (updated 2019-05-05) (17 minutes)
• How would you maximize the energy density of a capacitor? (p. 42) 2016-07-27 (5 minutes)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)
• Pulley generator (p. 3148) 2016-09-05 (2 minutes)
• Spring energy density (p. 1010) 2016-09-05 (updated 2019-04-20) (3 minutes)
• The internet is probably not going to collapse for economic reasons (p. 3194) 2016-09-06 (9 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• Lithium fission energy (p. 3285) 2016-09-06 (updated 2019-09-16) (6 minutes)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)
• Mic energy harvesting (p. 2583) 2016-09-07 (updated 2016-09-08) (5 minutes)

• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162) 2016-09-19 (5 minutes)
• Recuperator heat storage (p. 594) 2016-11-01 (updated 2019-08-21) (4 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Illumination cost (p. 1242) 2017-05-31 (3 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• Japan can achieve energy autarky via solar energy, but not much before 2027 (p. 2819) 2017-07-12 (4 minutes)
• Solar computer 2 (p. 414) 2017-07-19 (3 minutes)
• Energy storage in a personal water tower: pretty impractical (p. 2044) 2017-07-19 (2 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642) 2018-04-30 (updated 2018-07-05) (29 minutes)
• Turning off the power supply for every sample to reduce noise (p. 239) 2018-06-18 (2 minutes)
• Lithium battery welder (p. 2846) 2018-06-21 (updated 2019-01-22) (2 minutes)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Three phase oscillating belt (p. 214) 2018-10-28 (4 minutes)
• Cheap textures (p. 736) 2018-10-28 (updated 2019-05-05) (5 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Balcony battery (p. 2377) 2019-02-11 (updated 2019-12-06) (6 minutes)
• Groping toward a high-efficiency speaker driver (p. 1212) 2019-04-02 (15 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)
• Energy storage efficiency (p. 1300) 2019-07-30 (4 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated

2019-08-28) (37 minutes)
• Can artificially-lit vertical farming compete with greenhouses? (p. 2064) 2019-09-08 (12 minutes)
• Heliogen (p. 597) 2019-11-19 (6 minutes)
• Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) 2019-11-24 (2 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Underwater energy autonomy (p. 1662) 2019-11-25 (9 minutes)
• Phase change unplugged oven (p. 1433) 2019-12-15 (0 minutes)
• Argentine electric bill (p. 2898) 2019-12-18 (3 minutes)

Notes concerning “Environment”

- Fukushima leak (p. 2905) 2014-04-24 (6 minutes)
- Self replication changes (p. 2842) 2017-01-16 (5 minutes)
- A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
- Low-carbohydrate diets are ecologically sustainable (p. 540) 2018-04-27 (2 minutes)

Notes concerning “Egg of the Phoenix”

• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)

Notes concerning “Epistemology”

- On hanging out with cranks (p. 1715) 2008-04 (4 minutes)
- Studies support authority (p. 1541) 2017-04-10 (2 minutes)

Notes concerning “Erlang”

- Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
- Erlang musings (p. 2789) 2007 to 2009 (3 minutes)

Notes concerning “Español”

• Pensamientos acerca de diseñar un calefón solar (p. 117) 2012-10-15 (2 minutes)
• Más pensamientos acerca de diseñar un calefón solar (p. 1713) 2012-10-15 (5 minutes)
• Cristina Fernández de Kirchner tweets about the attempt to kidnap Assange (p. 985) 2014-04-24 (3 minutes)
• ¿Qué necesito para relación de pareja? (p. 1298) 2016-03-09 (6 minutes)
• La vibración del hierro, ¿es de baja frecuencia o qué? (p. 2231) 2016-10-07 (3 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)

Notes concerning “Espeak”

- Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
- English diphones (p. 2061) 2019-12-03 (5 minutes)

Notes concerning “Etymology”

- The etymology of “tradeoff” (p. 165) 2016-08-11 (5 minutes)
- One-line thoughts that don’t merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
- Replicating education (p. 557) 2017-07-19 (7 minutes)

Notes concerning “Euler method”

• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• Accelerating Euler’s Method on linear time-invariant systems by exponentiating matrices (p. 348) 2019-03-24 (updated 2019-04-02) (7 minutes)

Notes concerning “F-83”

• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)

Notes concerning “Facepalm”

• Barcode receipts (p. 2359) 2007 to 2009 (6 minutes)
• A stack of coordinate contexts (p. 2987) 2007 to 2009 (9 minutes)
• Food miles imply insignificant energy costs (p. 2187) 2007 to 2009 (4 minutes)
• Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)
• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Compression with second-order diffs (p. 2152) 2014-04-24 (3 minutes)
• Plato was not particularly democratic; ἄρχειν is not “participating in politics” (p. 1707) 2014-04-24 (5 minutes)
• Practically decodable random error correction codes with popcount (p. 606) 2015-07-01 (updated 2015-09-03) (6 minutes)
• Robust hashsplitting with sliding Range Minimum Query (p. 733) 2016-09-05 (7 minutes)
• Low-cost green thread locks (p. 252) 2016-09-06 (2 minutes)
• High academic achievement almost certainly depends more on tutoring than group averages by race or sex (p. 113) 2016-09-08 (3 minutes)
• La vibración del hierro, ¿es de baja frecuencia o qué? (p. 2231) 2016-10-07 (3 minutes)
• Analogies between spring–mass–dashpot systems, electrical systems, and fluidic systems (p. 1472) 2016-10-30 (4 minutes)
• The ultimate capacity of human memory if spaced-practice memorization works as advertised (p. 2442) 2017-01-04 (updated 2017-01-08) (14 minutes)
• What’s wrong with CoAP (p. 560) 2017-06-15 (3 minutes)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• What can you build out of 256-byte ROMs? (p. 1468) 2018-12-02 (1 minute)
• A failed attempt to make squares cheaper to compute (p. 1622) 2019-07-09 (updated 2019-07-11) (4 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• What it means that HTML is “not a programming language”, and why the ignorant sometimes think otherwise (p. 1555) 2019-09-09 (updated 2019-10-01) (24 minutes)

Notes concerning “Factionalism”

- Notch scorn (p. 115) 2019-04-20 (5 minutes)
- Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)

Notes concerning “Failure-free computing”

• Constant-space grep (p. 296) 2014-02-24 (3 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Linear trees (p. 1811) 2016-05-19 (updated 2016-05-20) (6 minutes)
• Statically bounding runtime (p. 2398) 2016-07-19 (4 minutes)
• Generic programming with proofs, specification, refinement, and specialization (p. 958) 2017-05-10 (6 minutes)
• The Z-machine memory model (p. 2903) 2017-07-19 (4 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)

Notes concerning “Feedback”

• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)
• Jello printing (p. 2426) 2016-12-14 (8 minutes)

Notes concerning “Fiction”

• He listened to the human intently (p. 2543) 2014-06-29 (4 minutes)

• Buenos Aires seen from behind the mirror (p. 809) 2014-09-02 (7 minutes)
• Statement from the Confederation of Teachers (p. 725) 2016-10-11 (updated 2016-10-12) (4 minutes)
• The imbalance inherent in copyright systems (p. 2158) 2017-07-19 (2 minutes)
• 2017 [Provisional English translation of intercepted transmission] (p. 2192) 2018-04-27 (updated 2018-07-14) (13 minutes)
• The fable of the specialized fox (p. 1076) 2019-08-17 (1 minute)
• GPT-2 sets the scene (p. 2978) 2019-11-22 (updated 2019-12-01) (22 minutes)

Notes concerning “Filesystems”

• Double ended log structured filesystem (p. 1653) 2007 to 2009 (4 minutes)
• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Robust hashsplitting with sliding Range Minimum Query (p. 733) 2016-09-05 (7 minutes)
• ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Raid zim (p. 253) 2019-01-17 (updated 2019-02-08) (1 minute)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)

Notes concerning “Flexures”

• Tapered thread (p. 363) 2015-09-03 (updated 2019-06-10) (4 minutes)
• Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)
• Foil origami robots (p. 2286) 2019-06-13 (updated 2019-06-14) (10 minutes)

Notes concerning “Flux deposition”

- Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) 2015-09-11 (updated 2019-12-20) (49 minutes)
- Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
- 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
- Needle binder injection printing (p. 1492) 2019-08-05 (12 minutes)

Notes concerning “Fonts”

• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• CIC-filter fonts (p. 1229) 2017-06-28 (1 minute)
• Wang tile font (p. 1463) 2018-08-16 (5 minutes)
• Hand drawn font compositing (p. 1810) 2018-10-28 (2 minutes)
• Dilating letterforms (p. 651) 2018-11-04 (15 minutes)
• Dercuano stylesheet notes (p. 374) 2019-04-28 (updated 2019-05-09) (72 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)

Notes concerning “Food storage”

• Storing dry bulk foods in used Coke bottles (p. 2145) 2012-10-15 (updated 2012-10-21) (5 minutes)
• Food storage (p. 2706) 2013-05-11 (updated 2013-05-17) (54 minutes)
• A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206) 2017-06-15 (4 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)

Notes concerning “Formal methods”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Statically bounding runtime (p. 2398) 2016-07-19 (4 minutes)
• Generic programming with proofs, specification, refinement, and specialization (p. 958) 2017-05-10 (6 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)

Notes concerning “Forth”

• Notes on reading eForth (p. 1398) 2007 to 2009 (9 minutes)
• Notes on reading eForth 1.0 for the 8086 (p. 541) 2007 to 2009 (5 minutes)
• Eur-Scheme: a simplified Ur-Scheme (p. 876) 2007 to 2009 (13 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
• Forth with named stacks (p. 2101) 2014-02-24 (7 minutes)
• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Rarely are function-local variables in Forth justified (p. 1055) 2018-04-27 (10 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Assembler bootstrapping (p. 2922) 2019-07-18 (updated 2019-12-08) (16 minutes)
• 10tcl ui (p. 1823) 2019-12-06 (17 minutes)
• Introduction to closures (p. 1403) 2019-12-07 (5 minutes)
• Forth assembling (p. 940) 2019-12-08 (updated 2019-12-11) (18 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)
• Can you eliminate backpatching? (p. 1769) 2019-12-17 (8 minutes)

Notes concerning “Fractals”

• Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
• Rendering iterated function systems (IFSes) with interval arithmetic (p. 2433) 2014-09-02 (6 minutes)
• Fractal palettes (p. 202) 2019-04-02 (7 minutes)

Notes concerning “Free software”

• Free software debugging (p. 136) 2007 to 2009 (2 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Better be weird (p. 1831) 2019-06-17 (updated 2019-06-24) (9 minutes)

Notes concerning “Frustration”

- Prototyping stuff (p. 176) 2016-08-11 (1 minute)
- Frustration (p. 3263) 2018-04-27 (2 minutes)

Notes concerning “Furniture”

- Inflatable stool (p. 1047) 2014-04-24 (6 minutes)
- Oval cam lock (p. 3060) 2019-11-26 (5 minutes)

Notes concerning “Games”

- Gaim group chat (p. 2677) 2007 to 2009 (3 minutes)
- Alien game challenge (p. 2313) 2015-09-03 (6 minutes)
- Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
- An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)
- Progressive revealment crypto (p. 2068) 2019-04-10 (2 minutes)
- Why Minetest is so addictive (p. 1781) 2019-04-20 (8 minutes)

Notes concerning “Garbage collection”

- Simple system language (p. 1659) 2013-05-17 (7 minutes)
- Linear trees (p. 1811) 2016-05-19 (updated 2016-05-20) (6 minutes)

Notes concerning “Garbage”

• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• One-line thoughts that don’t merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• Lab power supply (p. 2421) 2017-02-21 (updated 2018-06-18) (17 minutes)
• Home dehumidifier (p. 3131) 2018-05-20 (updated 2019-04-02) (12 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Cardboard furniture (p. 742) 2019-08-01 (updated 2019-08-11) (15 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29) (31 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Gardening”

- Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
- Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)

Notes concerning “Gelbart”

• Optical lever thermometer (p. 2624) 2015-09-03 (1 minute)
• Tapered thread (p. 363) 2015-09-03 (updated 2019-06-10) (4 minutes)

Notes concerning “Gestures”

• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• Complex linear regression (in the field ℂ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)

Notes concerning “Ghettobotics”

• Charge transfer servo (p. 3017) 2013-05-17 (2 minutes)
• Cheap shit ultrawideband (p. 2776) 2013-05-17 (10 minutes)
• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• You’re pretty much fucked if you want to build an oscilloscope on the AVR’s ADC (p. 1269) 2013-05-17 (3 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• Solar-cell Geiger counters (p. 3241) 2016-07-30 (1 minute)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)
• A 7-segment-display font with 68 glyphs (p. 1798) 2017-02-21 (4 minutes)
• Disk oscilloscope (p. 713) 2017-04-10 (updated 2017-06-20) (3 minutes)
• TV oscilloscope (p. 1253) 2017-04-10 (updated 2017-06-20) (4 minutes)
• Laser printer oscilloscope (p. 449) 2017-04-18 (updated 2017-06-20) (2 minutes)
• Can a simple nonlinear VCO enable super cheap oscilloscopes? (p. 1357) 2017-05-04 (updated 2017-05-10) (5 minutes)
• VCR oscilloscope (p. 213) 2017-05-10 (updated 2017-06-20) (2 minutes)
• CCD oscilloscope (p. 1861) 2017-06-20 (updated 2017-07-04) (7 minutes)
• Macroscopic capacitive DLP (p. 1834) 2019-04-08 (1 minute)
• Rubber wheel pinch drive (p. 2912) 2019-08-16 (updated 2019-08-18) (8 minutes)
• Examination of a shitty USB car charger (p. 286) 2019-10-24 (13 minutes)
• Audio logic analyzer (p. 1801) 2019-11-12 (3 minutes)

Notes concerning “Geographical information systems (GIS)”

- Full res globe (p. 1255) 2014-02-24 (1 minute)
- Fast geographical maps on Android (p. 455) 2015-10-16 (9 minutes)
- Stereographic map app (p. 1281) 2018-12-02 (2 minutes)

Notes concerning “Git”

• Git data (p. 2823) 2007 to 2009 (5 minutes)
• Git learnings (p. 3268) 2007 to 2009 (3 minutes)
• User-per-group (UPG), umask, and “Permission denied” on shared Git repos via ssh (p. 2481) 2007 to 2009 (4 minutes)
• Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)

Notes concerning “Glass”

• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• 3-D printing glass with continuously varying refractive indices for optics without internal surfaces (p. 1156) 2016-10-06 (3 minutes)

Notes concerning “Goertzel”

• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)

Notes concerning “Golang”

Notes concerning “Gossip”

• Gaim group chat (p. 2677) 2007 to 2009 (3 minutes)
• Viral wiki (p. 1024) 2015-10-15 (3 minutes)
• Hash gossip exchange (p. 1470) 2015-11-19 (4 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)

Notes concerning “GPGPU”

Notes concerning “Gradient descent”

• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)

Notes concerning “Gradients”

• Achieving smooth curves in scanline image generation (p. 1507) 2013-05-17 (1 minute)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Affine arithmetic has quadratic convergence when interval arithmetic has linear convergence (p. 1029) 2016-08-24 (updated 2017-01-18) (10 minutes)
• Gradient rendering (p. 583) 2016-09-24 (11 minutes)
• 3-D printing glass with continuously varying refractive indices for optics without internal surfaces (p. 1156) 2016-10-06 (3 minutes)
• Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)
• Gradient overlay (p. 1587) 2018-04-27 (2 minutes)
• Gradient pixels (p. 2202) 2018-08-16 (updated 2018-10-28) (9 minutes)

Notes concerning “Granular hypertext”

• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• A sentence-granularity hypertext editor (p. 2290) 2018-04-27 (4 minutes)
• Agenda hypertext (p. 1836) 2018-07-14 (updated 2018-07-15) (2 minutes)

Notes concerning “Graphics”

• I think I understand how to use libart’s antialiased rendering API now (p. 409) 2007 to 2009 (10 minutes)
• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• Achieving smooth curves in scanline image generation (p. 1507) 2013-05-17 (1 minute)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Simple system language (p. 1659) 2013-05-17 (7 minutes)
• Full res globe (p. 1255) 2014-02-24 (1 minute)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• Handling Landsat 8 images in limited RAM with netpbm (p. 1884) 2014-04-24 (4 minutes)
• Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• Alien game challenge (p. 2313) 2015-09-03 (6 minutes)
• An IMGUI-style drawing API isn’t necessarily just immediate-mode graphics (p. 2671) 2015-09-03 (3 minutes)
• Convolution surface plotting (p. 2264) 2015-09-03 (updated 2015-09-13) (2 minutes)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Hash feature detection (p. 3294) 2015-09-17 (5 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Minimal GUI libraries (p. 663) 2015-11-14 (updated 2015-11-15) (5 minutes)
• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• Interval radiosity (p. 1544) 2016-07-27 (1 minute)
• Kinect modeling (p. 164) 2016-09-16 (1 minute)
• Reconstructing a 3-D Lambertian surface from video with a moving light source (p. 3296) 2016-09-16 (1 minute)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17

(updated 2017-04-18) (23 minutes)
• Gradient rendering (p. 583) 2016-09-24 (11 minutes)
• Changing the basis to a more expressive one with better affordances (p. 1389) 2016-09-29 (5 minutes)
• Texture synthesis with spatial-domain particle filters (p. 857) 2016-10-06 (2 minutes)
• Current hardware trends tend toward raytracing (p. 1351) 2016-10-07 (4 minutes)
• Chintzy depth of field (p. 629) 2016-10-27 (1 minute)
• One-line thoughts that don't merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• What is the type of lerp? (p. 1985) 2017-01-08 (5 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• Constant time sets for pixel painting (p. 1484) 2017-02-07 (2 minutes)
• Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)
• Relational modeling (p. 1102) 2017-05-17 (updated 2017-06-01) (6 minutes)
• A plotter language of 9-bit bytes (p. 2154) 2017-05-29 (updated 2017-06-01) (11 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Pixel stream (p. 617) 2017-06-15 (updated 2018-10-26) (4 minutes)
• Compressing a screen update with a tree of dirty bits (p. 303) 2017-06-21 (1 minute)
• CIC-filter fonts (p. 1229) 2017-06-28 (1 minute)
• Rasterizing polies (p. 2023) 2017-07-19 (3 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• General purpose layout syntax (p. 3117) 2017-11-10 (updated 2019-09-01) (34 minutes)
• Optimization-based painting software (p. 1158) 2018-04-27 (1 minute)
• Gradient overlay (p. 1587) 2018-04-27 (2 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Wang tile font (p. 1463) 2018-08-16 (5 minutes)
• Gradient pixels (p. 2202) 2018-08-16 (updated 2018-10-28) (9 minutes)
• Window systems (p. 1335) 2018-10-26 (1 minute)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Bit difference array (p. 1748) 2018-10-28 (10 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Hand drawn font compositing (p. 1810) 2018-10-28 (2 minutes)
• Cheap textures (p. 736) 2018-10-28 (updated 2019-05-05) (5 minutes)
• Dilating letterforms (p. 651) 2018-11-04 (15 minutes)
• Gauzy shit (p. 2985) 2018-11-04 (4 minutes)
• Antialiased line drawing (p. 1803) 2018-11-13 (updated 2019-09-01)

(4 minutes)
• Stereographic map app (p. 1281) 2018-12-02 (2 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Sample reversal (p. 1353) 2018-12-18 (updated 2019-01-17) (5 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216) 2019-01-04 (updated 2019-11-12) (26 minutes)
• Median filtering (p. 3155) 2019-01-17 (11 minutes)
• Fractal palettes (p. 202) 2019-04-02 (7 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• Some musings on applying Fitts's Law to user interface design and data compression (p. 1164) 2019-05-06 (updated 2019-05-09) (27 minutes)
• An algebra of textures for interactive composition (p. 1283) 2019-05-08 (4 minutes)
• Granite texture (p. 1991) 2019-05-08 (updated 2019-05-09) (5 minutes)
• Dercuano rendering (p. 2300) 2019-05-11 (updated 2019-05-12) (3 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Midpoint method texture mapping (p. 1837) 2019-06-01 (3 minutes)
• Recursive curves (p. 1948) 2019-06-10 (5 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• Cloth structure from shading (p. 84) 2019-09-01 (2 minutes)
• Processing halftoning (p. 915) 2019-09-01 (15 minutes)
• Debokehfication (p. 473) 2019-09-01 (updated 2019-09-12) (4 minutes)
• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)
• Interval raymarching (p. 1342) 2019-11-02 (updated 2019-11-10) (6 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)
• Camera flash extrapolation (p. 2792) 2019-11-12 (6 minutes)
• Byte stream gui applications (p. 128) 2019-11-29 (updated 2019-11-30) (17 minutes)

Notes concerning “Graphs”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Circuit notation (p. 1161) 2016-09-08 (updated 2017-04-18) (7 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Incremental recomputation (p. 1184) 2018-04-27 (12 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)

Notes concerning “Greenarrays”

- Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
- Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
- An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)

Notes concerning “Grt”

- String tuple encoding (p. 2419) 2017-04-28 (2 minutes)
- ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)

Notes concerning “Graphical user interfaces”

• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• An IMGUI-style drawing API isn’t necessarily just immediate-mode graphics (p. 2671) 2015-09-03 (3 minutes)
• Minimal GUI libraries (p. 663) 2015-11-14 (updated 2015-11-15) (5 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• Pixel stream (p. 617) 2017-06-15 (updated 2018-10-26) (4 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• What does a futuristic OS look like? (p. 2163) 2017-08-18 (updated 2019-05-05) (6 minutes)
• General purpose layout syntax (p. 3117) 2017-11-10 (updated 2019-09-01) (34 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Window systems (p. 1335) 2018-10-26 (1 minute)
• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)
• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• Byte stream gui applications (p. 128) 2019-11-29 (updated 2019-11-30) (17 minutes)

Notes concerning “Hadamard matrices”

• Constructing error-correcting codes using Hadamard transforms (p. 1474) 2013-05-17 (updated 2013-05-20) (22 minutes)
• Hadamard rhythms (p. 831) 2019-11-01 (6 minutes)

Notes concerning “Hammers”

• Hammering toolhead (p. 3297) 2017-08-18 (6 minutes)
• Electric hammer (p. 1865) 2018-07-02 (updated 2018-07-05) (14 minutes)
• Shaped hammer face giant pressure (p. 3278) 2019-11-10 (21 minutes)

Notes concerning “Hand computers”

Notes concerning “Human–computer interaction”

• Error Reporting is Part of the Programmer's User Interface (p. 2323) 2007 to 2009 (18 minutes)
• Running your regular desktop in QEMU? (p. 292) 2007 to 2009 (3 minutes)
• Why Thunderbird is inadequate for opening a 7-gigabyte mbox (p. 980) 2007 to 2009 (2 minutes)
• wood and stone personal digital assistants (p. 3191) 2007 to 2009 (6 minutes)
• Writing math in Unicode with the Compose key (p. 1863) 2007 to 2009 (2 minutes)
• Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• In what sense is e the optimal branching factor, and what does it mean for menu tree design? (p. 2483) 2012-12-04 (3 minutes)
• How can we take advantage of 16:9 screens for programming? (p. 1982) 2012-12-17 (2 minutes)
• Clickable terminal patterns (p. 859) 2013-05-17 (2 minutes)
• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Precisely how is 3 “optimal” for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• How to use “correct horse battery staple” as an encryption key, including a recommended 4096-word list (p. 2522) 2014-04-24 (44 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Desbarrerarme: a UI for speaking to people (p. 186) 2015-09-03 (5 minutes)
• drag-and-drop calculator for touch devices (p. 1045) 2015-09-03 (5 minutes)
• Would Synthgramelodia make a good base for livecoding music? (p. 2540) 2015-09-03 (8 minutes)
• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• Interactive calculator 0 (p. 1453) 2015-09-17 (2 minutes)
• Writing hypertext is still a pain (p. 715) 2016-02-18 (6 minutes)
• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Do visually expanding images evoke an orienting response, or the startle response, and what does that mean for ZUIs? (p. 1805) 2016-06-03 (14 minutes)
• Prototyping stuff (p. 176) 2016-08-11 (1 minute)

• Toward a language for hacking around with natural-language processing algorithms (p. 1681) 2016-09-08 (7 minutes)
• DReX and "regular string transformations": would an RPN DSL work well? (p. 453) 2016-09-19 (3 minutes)
• Usability of scientific calculators (p. 2379) 2016-09-29 (19 minutes)
• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Generalizing my RPN calculator to support refactoring (p. 969) 2016-10-17 (12 minutes)
• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Quasicard: a hypothetical reimagining of HyperCard and TiddlyWiki (p. 416) 2017-04-18 (updated 2017-06-09) (18 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• Paper editing (p. 1761) 2017-06-15 (3 minutes)
• Database explorer (p. 225) 2017-06-20 (2 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• What does a futuristic OS look like? (p. 2163) 2017-08-18 (updated 2019-05-05) (6 minutes)
• Interactive calculator (p. 2771) 2018-04-26 (16 minutes)
• Interactive geometry (p. 508) 2018-04-26 (1 minute)
• Two-thumb quasimodal multitouch interaction techniques (p. 1765) 2018-04-26 (11 minutes)
• How can we do online pitch detection? (p. 1869) 2018-04-27 (updated 2018-04-30) (6 minutes)
• Clisweep (p. 2705) 2018-06-06 (3 minutes)
• Multitouch livecoding (p. 122) 2018-06-17 (1 minute)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• What would a better Unix shell look like? (p. 2831) 2018-11-27 (1 minute)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• Commentaries on reading Engelbart's "Augmenting Human Intellect" (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• Audio video boustrophedon sync (p. 858) 2019-04-03 (2 minutes)
• An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)
• Why Minetest is so addictive (p. 1781) 2019-04-20 (8 minutes)
• Fencepost cognitive interface errors in text editing (p. 993) 2019-04-24 (24 minutes)
• Dercuano stylesheet notes (p. 374) 2019-04-28 (updated 2019-05-09) (72 minutes)
• Dercuano formula display (p. 495) 2019-04-30 (5 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• Dercuano calculation (p. 3135) 2019-05-01 (3 minutes)

• Some musings on applying Fitts's Law to user interface design and data compression (p. 1164) 2019-05-06 (updated 2019-05-09) (27 minutes)
• Designing a drawing editor for well-factored drawings (p. 2115) 2019-05-07 (9 minutes)
• An algebra of textures for interactive composition (p. 1283) 2019-05-08 (4 minutes)
• Dercuano rendering (p. 2300) 2019-05-11 (updated 2019-05-12) (3 minutes)
• First impressions on using the μMath+ calculator program for Android (p. 195) 2019-05-21 (13 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Microsoft Windows uses \ for filenames because OS/8 programs used / for switches (p. 3098) 2019-05-25 (2 minutes)
• Recursive curves (p. 1948) 2019-06-10 (5 minutes)
• Prolog table outlining (p. 2837) 2019-07-05 (11 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• Complex linear regression (in the field ℂ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Text relational query (p. 1223) 2019-08-28 (10 minutes)
• Multitouch and accelerometer puppeteering (p. 1785) 2019-08-29 (updated 2019-09-01) (12 minutes)
• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)
• Hadamard rhythms (p. 831) 2019-11-01 (6 minutes)
• Audio logic analyzer (p. 1801) 2019-11-12 (3 minutes)

Notes concerning “Health”

- High-risk behavior in context (p. 1485) 2007 to 2009 (5 minutes)
- Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder (p. 527) 2013-05-17 (5 minutes)
- Only a constant factor worse (p. 1648) 2013-05-17 (16 minutes)

Notes concerning “Heat exchangers”

- Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
- Regenerative fuel air cutting (p. 2622) 2016-09-06 (4 minutes)
- Recuperator heat storage (p. 594) 2016-11-01 (updated 2019-08-21) (4 minutes)
- Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
- Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)

Notes concerning “Heating”

- Passive dehumidifier (p. 3256) 2017-03-20 (14 minutes)
- ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
- Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
- Radiant heating (p. 493) 2018-05-20 (3 minutes)
- Heating my apartment with a plastic tub of hot water (p. 1310) 2018-06-17 (3 minutes)
- Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
- Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
- Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)
- The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Heckballs”

• Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) 2016-08-17 (updated 2016-08-30) (24 minutes)
• Extending Heckballs (p. 3239) 2019-11-26 (6 minutes)

Notes concerning “History”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Emacs22 annoyances (p. 3197) 2007 to 2009 (4 minutes)
• Learning low level stuff is not just fun, but also useful (p. 815) 2007 to 2009 (5 minutes)
• Copyright status of the Oxford English Dictionary: relevant data (p. 82) 2007 to 2009 (3 minutes)
• Notes on Raph Levien's "Io" Programming Language (p. 1740) 2007 to 2009 (10 minutes)
• The AL programming language, dimensional analysis, and typing: do different dimensions really exist? (p. 731) 2007 to 2009 (2 minutes)

• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Saturation detector (p. 1588) 2013-05-17 (3 minutes)
• Cristina Fernández de Kirchner tweets about the attempt to kidnap Assange (p. 985) 2014-04-24 (3 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• What might Diamond-Age-like phyles look like in the real 21st century? (p. 1599) 2014-04-24 (9 minutes)
• Plato was not particularly democratic; ἄρχειν is not “participating in politics” (p. 1707) 2014-04-24 (5 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• Editor buffers (p. 1328) 2015-07-15 (updated 2015-09-03) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Ternary mergesort (p. 2161) 2015-09-03 (2 minutes)
• Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Transmission line computer (p. 509) 2016-07-11 (updated 2019-07-23) (7 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)

• Vitruvius could have taken photographs (p. 992) 2016-07-30 (1 minute)
• The etymology of "tradeoff" (p. 165) 2016-08-11 (5 minutes)
• Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)
• State of the world 2016 (p. 2973) 2016-09-05 (10 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• Hybrid RAM (p. 2877) 2016-09-24 (5 minutes)
• Changing the basis to a more expressive one with better affordances (p. 1389) 2016-09-29 (5 minutes)
• World War III is starting (?) (p. 346) 2016-10-17 (2 minutes)
• Academic lineage (p. 2292) 2016-10-30 (updated 2019-11-24) (15 minutes)
• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• Non-inverting logic (p. 861) 2017-02-18 (updated 2019-07-20) (8 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• The continuous-web press and the continuous press of the World-Wide Web (p. 1134) 2017-03-20 (6 minutes)
• The history of NoSQL and dbm (p. 45) 2017-04-10 (16 minutes)
• Studies support authority (p. 1541) 2017-04-10 (2 minutes)
• Quasicard: a hypothetical reimagining of HyperCard and TiddlyWiki (p. 416) 2017-04-18 (updated 2017-06-09) (18 minutes)
• Hipster stack 2017 (p. 2242) 2017-04-28 (updated 2017-05-04) (26 minutes)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• Vector instructions (p. 2977) 2017-07-19 (2 minutes)
• Multiplication with squares (p. 1983) 2017-07-19 (updated 2019-07-09) (5 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Some notes on reverse-engineering The Wizard's Castle (p. 1970) 2018-04-26 (9 minutes)
• 2017 [Provisional English translation of intercepted transmission] (p. 2192) 2018-04-27 (updated 2018-07-14) (13 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Gradient pixels (p. 2202) 2018-08-16 (updated 2018-10-28) (9 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)
• The Stretch book is truly alien (p. 1888) 2018-11-27 (6 minutes)
• Commentaries on reading Engelbart's "Augmenting Human Intellect" (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)

• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)
• Transactional event handlers (p. 139) 2019-01-24 (14 minutes)
• Elastic metamaterials (p. 719) 2019-03-19 (17 minutes)
• Fractal palettes (p. 202) 2019-04-02 (7 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)
• Notch scorn (p. 115) 2019-04-20 (5 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Microsoft Windows uses \ for filenames because OS/8 programs used / for switches (p. 3098) 2019-05-25 (2 minutes)
• Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)
• On the method of finite differences used in Babbage's Difference Engine (p. 827) 2019-05-31 (6 minutes)
• Computation with strain (p. 2812) 2019-06-13 (17 minutes)
• Everything is money? (p. 1859) 2019-08-31 (4 minutes)

Notes concerning “Holograms”

- Holographic archival (p. 766) 2014-04-24 (10 minutes)
- Analemma sundial (p. 1955) 2019-07-05 (11 minutes)
- Some extensions of William Beaty’s scratch holograms (p. 2536) 2019-07-11 (9 minutes)

Notes concerning “Household management and home economics”

• Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
• Pensamientos acerca de diseñar un calefón solar (p. 117) 2012-10-15 (2 minutes)
• Más pensamientos acerca de diseñar un calefón solar (p. 1713) 2012-10-15 (5 minutes)
• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Storing dry bulk foods in used Coke bottles (p. 2145) 2012-10-15 (updated 2012-10-21) (5 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Food storage (p. 2706) 2013-05-11 (updated 2013-05-17) (54 minutes)
• Only a constant factor worse (p. 1648) 2013-05-17 (16 minutes)
• Evaporation chimney (p. 2147) 2013-05-17 (13 minutes)
• Bottle washing (p. 921) 2014-04-24 (7 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Passivhaus seasonal thermal store (p. 1723) 2017-03-02 (updated 2017-03-07) (2 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
• A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206) 2017-06-15 (4 minutes)
• Complementary goods in home economics (p. 1878) 2017-07-19 (3 minutes)
• Energy storage in a personal water tower: pretty impractical (p. 2044) 2017-07-19 (2 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
• Deep freeze (p. 1465) 2017-08-22 (updated 2019-01-22) (7 minutes)

• Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)
• Notes on a possible household air filter (p. 1961) 2018-05-05 (updated 2018-05-15) (10 minutes)
• You can stuff a UHMWPE hammock in your wallet (p. 799) 2018-05-15 (updated 2018-10-28) (11 minutes)

• Radiant heating (p. 493) 2018-05-20 (3 minutes)
• Home dehumidifier (p. 3131) 2018-05-20 (updated 2019-04-02) (12 minutes)
• UHMWPE clothes could be lightweight and sturdy (p. 3071) 2018-06-05 (3 minutes)
• Heating my apartment with a plastic tub of hot water (p. 1310) 2018-06-17 (3 minutes)
• Barrel safety (p. 1097) 2018-07-14 (3 minutes)
• Hot water bottles (p. 3066) 2018-07-14 (4 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Balcony battery (p. 2377) 2019-02-11 (updated 2019-12-06) (6 minutes)
• Mayonnaise (p. 1320) 2019-03-19 (updated 2019-06-10) (10 minutes)
• Sous vide (p. 2921) 2019-04-02 (2 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)
• Waterfryer (p. 1462) 2019-04-20 (1 minute)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• Cardboard furniture (p. 742) 2019-08-01 (updated 2019-08-11) (15 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Nomadic furniture optimization (p. 1658) 2019-12-15 (2 minutes)
• Phase change unplugged oven (p. 1433) 2019-12-15 (0 minutes)
• Argentine electric bill (p. 2898) 2019-12-18 (3 minutes)

Notes concerning “Housing”

• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Pipe dome (p. 3068) 2017-07-19 (7 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Hp 9100”

• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)

Notes concerning “HTML”

• HTML is terser and more robust than S-expressions (p. 562) 2007 to 2009 (4 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Web prefetch (p. 3046) 2017-06-15 (1 minute)
• Dercuano stylesheet notes (p. 374) 2019-04-28 (updated 2019-05-09) (72 minutes)
• Dercuano formula display (p. 495) 2019-04-30 (5 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)

Notes concerning “HTTP”

• Stuff I’ve posted to kragen-tol over the years about post-HTTP (p. 1815) 2014-02-24 (12 minutes)
• Micro pubsub (p. 1504) 2017-06-15 (8 minutes)
• Compressing REST transactions with per-connection state (p. 1117) 2018-04-27 (11 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)

Notes concerning “Human rights”

• In a world with ubiquitous surveillance, what does politics look like? (p. 1615) 2014-04-24 (11 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• World War III is starting (?) (p. 346) 2016-10-17 (2 minutes)
• Studies support authority (p. 1541) 2017-04-10 (2 minutes)
• The imbalance inherent in copyright systems (p. 2158) 2017-07-19 (2 minutes)
• Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)

Notes concerning “Humor”

Notes concerning “Hypertext”

• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• A proposal to support hypertext links in ANSI terminals (p. 486) 2013-05-17 (updated 2019-12-26) (13 minutes)
• Writing hypertext is still a pain (p. 715) 2016-02-18 (6 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• The book written in itself (p. 2400) 2016-06-12 (updated 2016-06-14) (18 minutes)
• Quasicard: a hypothetical reimagining of HyperCard and TiddlyWiki (p. 416) 2017-04-18 (updated 2017-06-09) (18 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• A sentence-granularity hypertext editor (p. 2290) 2018-04-27 (4 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Agenda hypertext (p. 1836) 2018-07-14 (updated 2018-07-15) (2 minutes)
• Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) 2018-11-09 (3 minutes)
• Commentaries on reading Engelbart’s “Augmenting Human Intellect” (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• Dercuano backlinks (p. 475) 2019-05-22 (7 minutes)

Notes concerning “Ice vests”

• Ice pants (p. 298) 2017-04-04 (updated 2019-01-22) (17 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)

Notes concerning “Image approximation”

• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)

Notes concerning “Immediate-mode GUIs”

• How to generate unique IDs for IMGUI object persistence? (p. 2035) 2014-09-02 (3 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• An IMGUI-style drawing API isn’t necessarily just immediate-mode graphics (p. 2671) 2015-09-03 (3 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Trees as code (p. 2488) 2016-05-10 (4 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)
• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)

Notes concerning “Incentive design”

- Barcode receipts (p. 2359) 2007 to 2009 (6 minutes)
- Free software debugging (p. 136) 2007 to 2009 (2 minutes)
- Microfinance (p. 2875) 2007 to 2009 (6 minutes)
- Dollar auctions and tournaments in human society (p. 884) 2013-05-17 (7 minutes)
- Dutch auction raffle (p. 2474) 2018-06-05 (3 minutes)

Notes concerning "Incremental computation"

• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• How can we usefully cache screen images for incrementalization? (p. 1077) 2013-05-17 (18 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
• A reactive crawler using Amygdala (p. 2492) 2014-09-02 (updated 2014-09-19) (4 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870) 2015-05-28 (updated 2016-06-22) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• Incremental recomputation (p. 1184) 2018-04-27 (12 minutes)
• Composing code gobbets with implicit dependencies (p. 2437) 2018-04-27 (updated 2019-05-21) (3 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)
• Robust local search in vector spaces using adaptive step sizes, and

thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)

Notes concerning “Incremental search”

• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Fencepost cognitive interface errors in text editing (p. 993) 2019-04-24 (24 minutes)

Notes concerning “Independence”

• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Food storage (p. 2706) 2013-05-11 (updated 2013-05-17) (54 minutes)
• Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• Ultraslow radio for resilient global communication (p. 2071) 2013-05-17 (updated 2013-05-20) (26 minutes)
• The Tinkerer’s Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Bike charger (p. 2099) 2014-04-24 (2 minutes)
• Offline datasets (p. 599) 2014-04-24 (15 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• The book written in itself (p. 2400) 2016-06-12 (updated 2016-06-14) (18 minutes)
• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)
• Regenerative fuel air cutting (p. 2622) 2016-09-06 (4 minutes)
• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162) 2016-09-19 (5 minutes)
• MiniOS (p. 1091) 2016-12-28 (updated 2017-01-03) (6 minutes)
• Lab power supply (p. 2421) 2017-02-21 (updated 2018-06-18) (17 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• Minimum hardware and software to get a flexible notetaking device running (p. 535) 2017-04-28 (4 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Distributed computing environment (p. 776) 2017-07-19 (8 minutes)
• The imbalance inherent in copyright systems (p. 2158) 2017-07-19 (2 minutes)
• Pipe dome (p. 3068) 2017-07-19 (7 minutes)
• Options for bootstrapping a compiler from a tiny compiler using Brainfuck (p. 2958) 2017-07-19 (2 minutes)
• Solar computer 2 (p. 414) 2017-07-19 (3 minutes)
• Energy storage in a personal water tower: pretty impractical (p. 2044) 2017-07-19 (2 minutes)

• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• What does a futuristic OS look like? (p. 2163) 2017-08-18 (updated 2019-05-05) (6 minutes)
• Deep freeze (p. 1465) 2017-08-22 (updated 2019-01-22) (7 minutes)

• Frustration (p. 3263) 2018-04-27 (2 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Some notes on FullPliant and Pliant (p. 866) 2018-04-27 (9 minutes)
• Urban autarkic network (p. 1026) 2018-04-27 (1 minute)
• Barrel safety (p. 1097) 2018-07-14 (3 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• A phase-change soldering iron (p. 2270) 2019-05-08 (updated 2019-05-09) (14 minutes)
• Assembler bootstrapping (p. 2922) 2019-07-18 (updated 2019-12-08) (16 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)
• Cardboard furniture (p. 742) 2019-08-01 (updated 2019-08-11) (15 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29) (31 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated 2019-08-28) (37 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)
• Bokeh pointcasting (p. 92) 2019-09-08 (updated 2019-09-09) (16 minutes)
• Hearing aids for disability compensation, protection, and augmentation (p. 764) 2019-09-08 (updated 2019-09-09) (4 minutes)

• Expanded mineral beads (p. 2166) 2019-10-01 (12 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Underwater energy autonomy (p. 1662) 2019-11-25 (9 minutes)

• 10tcl ui (p. 1823) 2019-12-06 (17 minutes)
• Forth assembling (p. 940) 2019-12-08 (updated 2019-12-11) (18 minutes)
• Really simple lab power supply (p. 240) 2019-12-10 (7 minutes)
• Berlinite gel (p. 848) 2019-12-14 (updated 2019-12-15) (10 minutes)
• Nomadic furniture optimization (p. 1658) 2019-12-15 (2 minutes)
• Can you eliminate backpatching? (p. 1769) 2019-12-17 (8 minutes)

Notes concerning “Induction”

- Hot wire saw (p. 3159) 2015-12-28 (updated 2019-06-02) (10 minutes)
- Inductor thermocouple sensing (p. 2037) 2019-06-01 (21 minutes)
- Induction kiln (p. 2352) 2019-06-02 (19 minutes)

Notes concerning “Information theory”

• In what sense is e the optimal branching factor, and what does it mean for menu tree design? (p. 2483) 2012-12-04 (3 minutes)
• Constructing error-correcting codes using Hadamard transforms (p. 1474) 2013-05-17 (updated 2013-05-20) (22 minutes)
• Ultraslow radio for resilient global communication (p. 2071) 2013-05-17 (updated 2013-05-20) (26 minutes)
• Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)
• Practically decodable random error correction codes with popcount (p. 606) 2015-07-01 (updated 2015-09-03) (6 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• Broadcast ECC with graceful degradation, or avoiding the cliff effect (p. 2045) 2018-12-18 (5 minutes)
• Some musings on applying Fitts’s Law to user interface design and data compression (p. 1164) 2019-05-06 (updated 2019-05-09) (27 minutes)
• Free space optical coding gain (p. 1244) 2019-05-08 (updated 2019-05-09) (4 minutes)

Notes concerning “Input devices”

• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• Coinductive keyboard (p. 1893) 2016-07-30 (4 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)

Notes concerning "Instruction sets"

• Using bytecode won't make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• Notes on reading eForth (p. 1398) 2007 to 2009 (9 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• Tagged dataflow (p. 405) 2007 to 2009 (2 minutes)
• The Dontmove archival virtual machine (p. 2113) 2014-06-29 (5 minutes)
• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• XCHG: An Archival Swap Machine (p. 2997) 2014-06-29 (7 minutes)
• Practically decodable random error correction codes with popcount (p. 606) 2015-07-01 (updated 2015-09-03) (6 minutes)
• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• Designing an archival virtual machine (p. 3203) 2016-05-12 (6 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• Bitsliced operations with a hypercube of shuffle operations (p. 2363) 2016-11-30 (2 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• Golomb-coding operands as belt offsets likely won't increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

• Compact code cpu (p. 397) 2017-07-19 (3 minutes)
• Vector instructions (p. 2977) 2017-07-19 (2 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Obscurity platform (p. 2991) 2018-04-27 (1 minute)
• Word stream architecture (p. 2215) 2018-06-17 (13 minutes)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Bit difference array (p. 1748) 2018-10-28 (10 minutes)

• Digital noise generators (p. 1137) 2018-10-28 (2 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Performance properties of sets of bitwise operations (p. 636) 2018-11-06 (updated 2018-11-07) (16 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)
• Parallel register file (p. 2952) 2018-11-27 (2 minutes)
• The Stretch book is truly alien (p. 1888) 2018-11-27 (6 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)
• Memory safe virtual machines (p. 975) 2019-12-04 (14 minutes)
• My very first toddling steps in ARM assembly language (p. 1684) 2019-12-10 (updated 2019-12-13) (46 minutes)

Notes concerning "Interval and affine arithmetic"

• An algebraic approach to 3D geometry (p. 669) 2014-06-03 (updated 2014-06-29) (22 minutes)
• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• Rendering iterated function systems (IFSes) with interval arithmetic (p. 2433) 2014-09-02 (6 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Bayesian and Gricean programming (p. 711) 2015-08-20 (3 minutes)

• Convolution with intervals (p. 1044) 2015-09-07 (1 minute)
• Interactive calculator 0 (p. 1453) 2015-09-17 (2 minutes)
• Interval filters (p. 1282) 2015-09-17 (2 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Interval radiosity (p. 1544) 2016-07-27 (1 minute)
• Affine arithmetic has quadratic convergence when interval arithmetic has linear convergence (p. 1029) 2016-08-24 (updated 2017-01-18) (10 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)
• High-precision control of low-stiffness sytems with bounded-Q resonances (p. 1002) 2017-05-29 (updated 2017-06-01) (4 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Affine arithmetic optimization (p. 2801) 2017-07-19 (updated 2019-09-15) (3 minutes)
• Solving initial-value problems faster and with guaranteed error bounds with affine arithmetic (p. 836) 2019-04-02 (5 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Query evaluation with interval-annotated trees over sequences (p. 1423) 2019-08-30 (updated 2019-09-03) (30 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)
• Interval raymarching (p. 1342) 2019-11-02 (updated 2019-11-10) (6 minutes)

Notes concerning “Io”

• Notes on Raph Levien's "Io" Programming Language (p. 1740) 2007 to 2009 (10 minutes)
• Implementing flatMap in terms of call/cc, as in Raph Levien’s Io (p. 3248) 2015-09-03 (3 minutes)

Notes concerning “The Jaquet-Droz automata”

• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Differential spiral cam (p. 512) 2017-07-19 (9 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)

Notes concerning “Java”

• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• A reactive crawler using Amygdala (p. 2492) 2014-09-02 (updated 2014-09-19) (4 minutes)
• Ndarray java (p. 2261) 2015-05-28 (1 minute)
• Storing CSV records in minimal memory in Java (p. 524) 2015-09-03 (6 minutes)
• Notes on higher-order programming on the JVM (p. 1355) 2016-09-06 (6 minutes)

Notes concerning “Journal”

• Vanagon mail (p. 2032) 2007 to 2009 (3 minutes)
• Smoky day (p. 3010) 2008-04-19 (4 minutes)
• Personal notes from 2013-06-06 (p. 2673) 2013-06-06 (updated 2014-04-24) (11 minutes)
• Some personal notes from February 2014 (p. 2134) 2014-02-13 (8 minutes)
• A Sunday in 2014 (p. 1089) 2014-02-24 (3 minutes)
• Jim Weirich’s death and my daily life (p. 829) 2014-04-24 (5 minutes)
• Notes from a Buenos Aires blackout, summer 2013-2014 (p. 267) 2014-04-24 (15 minutes)
• Ostinatto (p. 2780) 2014-04-24 (4 minutes)
• José, the Galician mover (p. 3076) 2015-11-09 (2 minutes)
• ¿Qué necesito para relación de pareja? (p. 1298) 2016-03-09 (6 minutes)
• Phase-change heat reservoirs for household climate control (p. 2257) 2016-06-14 (updated 2016-06-17) (13 minutes)

Notes concerning “JS”

• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• A type-inferred dialect of JS (p. 265) 2016-04-22 (4 minutes)
• Hipster stack 2017 (p. 2242) 2017-04-28 (updated 2017-05-04) (26 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• JIT-compiling array computation graphs in JS (p. 155) 2017-07-19 (1 minute)
• Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
• How small can we make a comfortable subset of JS? (p. 1348) 2018-11-27 (updated 2018-12-02) (3 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)
• Introduction to closures (p. 1403) 2019-12-07 (5 minutes)

Notes concerning “JSON”

- Twingler (p. 2547) 2014-02-24 (7 minutes)
- Tagging parsers (p. 208) 2018-11-23 (updated 2018-12-10) (9 minutes)

Notes concerning “Jupyter”

• Quasicard: a hypothetical reimagining of HyperCard and TiddlyWiki (p. 416) 2017-04-18 (updated 2017-06-09) (18 minutes)
• Literate programs should include example output, like Jupyter, but Jupyter is imperfect (p. 1308) 2018-04-27 (3 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)

Notes concerning “Kanthal”

- Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
- An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
- Radiant heating (p. 493) 2018-05-20 (3 minutes)

Notes concerning “Keyboards”

- Writing math in Unicode with the Compose key (p. 1863) 2007 to 2009 (2 minutes)
- Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)
- Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
- Coinductive keyboard (p. 1893) 2016-07-30 (4 minutes)
- Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)

Notes concerning “Kilns”

• Improvising high-temperature refractory materials for pottery kilns (p. 2701) 2013-05-17 (4 minutes)
• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Inductor thermocouple sensing (p. 2037) 2019-06-01 (21 minutes)
• Induction kiln (p. 2352) 2019-06-02 (19 minutes)

Notes concerning “Kogluktualuk”

• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)

Notes concerning “Laser cutters”

Notes concerning “Lasers”

• Phosphorescent laser display (p. 1987) 2016-08-16 (8 minutes)
• Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times? (p. 1027) 2016-09-06 (3 minutes)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)

Notes concerning “Latency”

• Why Thunderbird is inadequate for opening a 7-gigabyte mbox (p. 980) 2007 to 2009 (2 minutes)
• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Interactive calculator 0 (p. 1453) 2015-09-17 (2 minutes)
• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Cheap textures (p. 736) 2018-10-28 (updated 2019-05-05) (5 minutes)
• Transactional event handlers (p. 139) 2019-01-24 (14 minutes)

Notes concerning “Law”

• Copyright status of the Oxford English Dictionary: relevant data (p. 82) 2007 to 2009 (3 minutes)
• What might Diamond-Age-like phyles look like in the real 21st century? (p. 1599) 2014-04-24 (9 minutes)

Notes concerning “Layout”

• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• General purpose layout syntax (p. 3117) 2017-11-10 (updated 2019-09-01) (34 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)

Notes concerning “Laziness”

- Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
- Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
- Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)

Notes concerning “LevelDB”

• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Mail reader (p. 3290) 2018-04-27 (updated 2018-06-18) (7 minutes)

• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)

Notes concerning “The LGP-30 computer”

• Transmission line computer (p. 509) 2016-07-11 (updated 2019-07-23) (7 minutes)
• Non-inverting logic (p. 861) 2017-02-18 (updated 2019-07-20) (8 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)

Notes concerning “Li ion”

• Lithium battery welder (p. 2846) 2018-06-21 (updated 2019-01-22) (2 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Light deflection”

• Lenticular deflector (p. 2612) 2019-09-08 (updated 2019-09-09) (9 minutes)
• Kerr snow display (p. 3220) 2019-11-12 (3 minutes)

Notes concerning “Lighting”

Notes concerning “Linear algebra”

• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Complex linear regression (in the field $\mathbb{C}$ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Differentiable neighborhood regression (p. 2944) 2019-08-31 (15 minutes)

Notes concerning “Lisp”

• Enumerating binary trees and their elements (p. 1445) 2007 to 2009 (4 minutes)
• Eur-Scheme: a simplified Ur-Scheme (p. 876) 2007 to 2009 (13 minutes)
• Designing a Scheme for APL-like array computations, like Lush (p. 661) 2007 to 2009 (4 minutes)
• Quasiquote patterns (p. 3021) 2007 to 2009 (9 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Linear trees (p. 1811) 2016-05-19 (updated 2016-05-20) (6 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)

Notes concerning “Lithium”

• Lithium fission energy (p. 3285) 2016-09-06 (updated 2019-09-16) (6 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)

Notes concerning “Logging”

• Desbarrerarme: a UI for speaking to people (p. 186) 2015-09-03 (5 minutes)
• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)
• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)

Notes concerning “Log-structured merge trees (LSM-trees)”

• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
• Set hashing (p. 2485) 2017-03-09 (9 minutes)
• Incremental persistent binary array sets (p. 1008) 2017-04-10 (4 minutes)

Notes concerning “Lua”

• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• Improving Lua #L with incremental prefix sum in the ∧ monoid (p. 2008) 2018-12-18 (7 minutes)
• B-Tree ropes (p. 2762) 2019-09-24 (updated 2019-09-25) (19 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)

Notes concerning “Magic kazoo”

• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Another candidate lightweight frequency tracking algorithm (p. 2069) 2017-08-18 (4 minutes)
• How can we do online pitch detection? (p. 1869) 2018-04-27 (updated 2018-04-30) (6 minutes)

Notes concerning "Manufacturing"

• Polycaprolactone (p. 1813) 2007 to 2009 (3 minutes)
• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Notes on 3-D printing a mechanical LUT (p. 1326) 2014-04-24 (3 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• Comparison of the PCO-1810 and PCO-1881 plastic bottlecap standards (p. 3223) 2014-05-25 (updated 2016-07-27) (2 minutes)
• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Hot wire saw (p. 3159) 2015-12-28 (updated 2019-06-02) (10 minutes)
• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)
• Prototyping stuff (p. 176) 2016-08-11 (1 minute)
• Regenerative fuel air cutting (p. 2622) 2016-09-06 (4 minutes)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)
• Filling hollow FDM things with other materials (p. 2119) 2016-09-07 (5 minutes)
• Laser ablation of zinc or pewter for printed circuit boards (p. 2799) 2016-09-19 (4 minutes)
• Spark particulate sieve (p. 2047) 2016-10-06 (updated 2016-10-11) (7 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• Servoing a V-plotter with a webcam? (p. 62) 2017-02-16 (3 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• Vibratory powder delivery (p. 1747) 2017-02-25 (2 minutes)
• How cheap can laser-cut boxes be? (p. 2545) 2017-06-01 (2 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Hammering toolhead (p. 3297) 2017-08-18 (6 minutes)
• A brief note on autonomous cyclic fabrication systems from inorganic raw materials (p. 2965) 2018-04-27 (1 minute)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• UHMWPE clothes could be lightweight and sturdy (p. 3071)

2018-06-05 (3 minutes)
• Electrolytic anodizing, with a small movable electrode (p. 3059) 2018-10-28 (2 minutes)
• Single-point incremental forming of aluminum foil (p. 769) 2019-03-11 (updated 2019-06-10) (14 minutes)
• Elastic metamaterials (p. 719) 2019-03-19 (17 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)
• Caustic business card (p. 255) 2019-04-08 (3 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Induction kiln (p. 2352) 2019-06-02 (19 minutes)
• Drone cutting (p. 1106) 2019-06-11 (12 minutes)
• Cardboard furniture (p. 742) 2019-08-01 (updated 2019-08-11) (15 minutes)
• Needle binder injection printing (p. 1492) 2019-08-05 (12 minutes)
• Printed circuits on fired-clay ceramic (p. 960) 2019-08-13 (11 minutes)
• Harmonic motion chain robot (p. 2197) 2019-08-16 (2 minutes)
• Gold leaf trusses (p. 3055) 2019-08-31 (11 minutes)
• Expanded mineral beads (p. 2166) 2019-10-01 (12 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)
• Hot lye granite cutting (p. 581) 2019-11-01 (2 minutes)
• Shaped hammer face giant pressure (p. 3278) 2019-11-10 (21 minutes)
• Incremental roller comb forming (p. 3146) 2019-11-27 (4 minutes)
• Berlinite gel (p. 848) 2019-12-14 (updated 2019-12-15) (10 minutes)
• Nomadic furniture optimization (p. 1658) 2019-12-15 (2 minutes)

Notes concerning “Materials”

• Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
• Polycaprolactone (p. 1813) 2007 to 2009 (3 minutes)
• The economics of solar energy (p. 1175) 2008 (27 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Improvising high-temperature refractory materials for pottery kilns (p. 2701) 2013-05-17 (4 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• Holographic archival (p. 766) 2014-04-24 (10 minutes)
• Inflatable stool (p. 1047) 2014-04-24 (6 minutes)
• An extremely simple electromechanical state machine (p. 50) 2014-04-24 (16 minutes)
• Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Waterproofing (p. 429) 2015-09-03 (4 minutes)
• Tapered thread (p. 363) 2015-09-03 (updated 2019-06-10) (4 minutes)
• Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) 2015-09-11 (updated 2019-12-20) (49 minutes)
• Hot wire saw (p. 3159) 2015-12-28 (updated 2019-06-02) (10 minutes)
• Material merits (p. 1496) 2016-05-08 (6 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Spring energy density (p. 3106) 2016-05-28 (updated 2016-06-06) (13 minutes)
• Phase-change heat reservoirs for household climate control (p. 2257) 2016-06-14 (updated 2016-06-17) (13 minutes)
• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• How would you maximize the energy density of a capacitor? (p. 42) 2016-07-27 (5 minutes)
• Electroluminescent matrix (p. 974) 2016-07-27 (2 minutes)
• Vitruvius could have taken photographs (p. 992) 2016-07-30 (1 minute)
• Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)
• Hot oil cutter (p. 3287) 2016-08-16 (updated 2016-08-17)

(8 minutes)
• Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) 2016-08-17 (updated 2016-08-30) (24 minutes)
• Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• Regenerative fuel air cutting (p. 2622) 2016-09-06 (4 minutes)
• Sun cutter (p. 56) 2016-09-06 (9 minutes)
• Lithium fission energy (p. 3285) 2016-09-06 (updated 2019-09-16) (6 minutes)
• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Filling hollow FDM things with other materials (p. 2119) 2016-09-07 (5 minutes)
• Laser ablation of zinc or pewter for printed circuit boards (p. 2799) 2016-09-19 (4 minutes)
• Cross current zone melting (p. 1872) 2016-10-06 (1 minute)
• Freeze distillation at 1 Hz (p. 2796) 2016-10-06 (5 minutes)
• 3-D printing glass with continuously varying refractive indices for optics without internal surfaces (p. 1156) 2016-10-06 (3 minutes)
• Hot air ice shaping (p. 864) 2016-10-06 (4 minutes)
• Spark particulate sieve (p. 2047) 2016-10-06 (updated 2016-10-11) (7 minutes)
• Recuperator heat storage (p. 594) 2016-11-01 (updated 2019-08-21) (4 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Clay fabrication objectives (p. 2111) 2017-01-16 (updated 2017-01-17) (3 minutes)
• Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
• Passive dehumidifier (p. 3256) 2017-03-20 (14 minutes)
• Ice pants (p. 298) 2017-04-04 (updated 2019-01-22) (17 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Fast sea salt evaporator (p. 1087) 2017-06-01 (3 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)
• Dyneema (p. 123) 2017-07-19 (2 minutes)
• Pipe dome (p. 3068) 2017-07-19 (7 minutes)
• Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Deep freeze (p. 1465) 2017-08-22 (updated 2019-01-22) (7 minutes)

• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08)

(12 minutes)
• Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)
• A brief note on autonomous cyclic fabrication systems from inorganic raw materials (p. 2965) 2018-04-27 (1 minute)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)
• Laser cut next step (p. 824) 2018-04-27 (updated 2018-04-30) (7 minutes)
• Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642) 2018-04-30 (updated 2018-07-05) (29 minutes)
• Notes on a possible household air filter (p. 1961) 2018-05-05 (updated 2018-05-15) (10 minutes)
• You can stuff a UHMWPE hammock in your wallet (p. 799) 2018-05-15 (updated 2018-10-28) (11 minutes)
• UHMWPE clothes could be lightweight and sturdy (p. 3071) 2018-06-05 (3 minutes)
• Why is there so much anti-plastic sentiment? Visibility, Arcadian primitivism, conspicuous consumption, and profit. (p. 3270) 2018-06-21 (7 minutes)
• Notes on circuitry for the Nutra seed activator (p. 3099) 2018-08-16 (20 minutes)
• Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
• Electrolytic anodizing, with a small movable electrode (p. 3059) 2018-10-28 (2 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Friction-cutting plastic (p. 2412) 2019-02-25 (8 minutes)
• Ultralight tunnel personal rapid transit (p. 706) 2019-03-11 (15 minutes)
• Single-point incremental forming of aluminum foil (p. 769) 2019-03-11 (updated 2019-06-10) (14 minutes)
• Elastic metamaterials (p. 719) 2019-03-19 (17 minutes)
• Paper/foil relays (p. 3273) 2019-04-02 (updated 2019-10-23) (13 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Plastic cutters (p. 1074) 2019-04-20 (5 minutes)
• Plasma glazing (p. 71) 2019-04-24 (1 minute)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• A phase-change soldering iron (p. 2270) 2019-05-08 (updated 2019-05-09) (14 minutes)
• Inductor thermocouple sensing (p. 2037) 2019-06-01 (21 minutes)
• Induction kiln (p. 2352) 2019-06-02 (19 minutes)
• Drone cutting (p. 1106) 2019-06-11 (12 minutes)
• Foil origami robots (p. 2286) 2019-06-13 (updated 2019-06-14) (10 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033) 2019-07-16 (updated 2019-07-17)

(28 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)
• Cardboard furniture (p. 742) 2019-08-01 (updated 2019-08-11) (15 minutes)
• Needle binder injection printing (p. 1492) 2019-08-05 (12 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29) (31 minutes)
• Printed circuits on fired-clay ceramic (p. 960) 2019-08-13 (11 minutes)
• Gold leaf trusses (p. 3055) 2019-08-31 (11 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• Expanded mineral beads (p. 2166) 2019-10-01 (12 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)
• Hot lye granite cutting (p. 581) 2019-11-01 (2 minutes)
• Shaped hammer face giant pressure (p. 3278) 2019-11-10 (21 minutes)
• Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) 2019-11-24 (2 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Bootstrapping rope bridges and other tensile structures with UHMWPE-bearing drones (p. 2950) 2019-11-25 (5 minutes)
• Extending Heckballs (p. 3239) 2019-11-26 (6 minutes)
• Incremental roller comb forming (p. 3146) 2019-11-27 (4 minutes)
• Berlinite gel (p. 848) 2019-12-14 (updated 2019-12-15) (10 minutes)
• Phase change unplugged oven (p. 1433) 2019-12-15 (0 minutes)
• Sulfuric acid dehydration printing (p. 174) 2019-12-18 (updated 2019-12-19) (3 minutes)

Notes concerning "Math"

• The Gelfand Principle, or how to choose educational examples (p. 1967) 2007 to 2009 (8 minutes)
• Additive smoothing for Markov models (p. 2429) 2007 to 2009 (updated 2019-05-19) (11 minutes)
• In what sense is e the optimal branching factor, and what does it mean for menu tree design? (p. 2483) 2012-12-04 (3 minutes)
• Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• Achieving smooth curves in scanline image generation (p. 1507) 2013-05-17 (1 minute)
• Constructing error-correcting codes using Hadamard transforms (p. 1474) 2013-05-17 (updated 2013-05-20) (22 minutes)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• Bottle washing (p. 921) 2014-04-24 (7 minutes)
• Fixed point (p. 807) 2014-04-24 (1 minute)
• lattices, powersets, bitstrings, and efficient OLAP (p. 2345) 2014-04-24 (17 minutes)
• Polynomial-spline FIR kernels by integrating sparse kernels (p. 1819) 2014-04-24 (12 minutes)
• Very composite numbers (p. 2490) 2014-04-24 (4 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• An algebraic approach to 3D geometry (p. 669) 2014-06-03 (updated 2014-06-29) (22 minutes)
• Division (p. 2181) 2014-06-05 (14 minutes)
• Rendering iterated function systems (IFSes) with interval arithmetic (p. 2433) 2014-09-02 (6 minutes)
• Ternary mergesort (p. 2161) 2015-09-03 (2 minutes)
• Very fast FIR filtering with time-domain zero stuffing and splines (p. 1146) 2015-09-03 (updated 2015-09-07) (13 minutes)
• Convolution surface plotting (p. 2264) 2015-09-03 (updated 2015-09-13) (2 minutes)
• Convolution with intervals (p. 1044) 2015-09-07 (1 minute)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Gaussian spline reconstruction (p. 656) 2016-06-05 (updated 2016-06-06) (5 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• Changing the basis to a more expressive one with better affordances (p. 1389) 2016-09-29 (5 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• Analogies between spring–mass–dashpot systems, electrical systems, and fluidic systems (p. 1472) 2016-10-30 (4 minutes)
• Academic lineage (p. 2292) 2016-10-30 (updated 2019-11-24)

(15 minutes)
• Using Aryabhata's pulverizer algorithm to calculate multiplicative inverses in prime Galois fields and other multiplicative groups (p. 2255) 2017-01-06 (updated 2019-07-05) (4 minutes)
• What is the type of lerp? (p. 1985) 2017-01-08 (5 minutes)
• Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)
• Golomb-coding operands as belt offsets likely won't increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)
• Xor 1 to 1 hashing (p. 1595) 2017-07-19 (updated 2017-08-03) (10 minutes)
• Affine arithmetic optimization (p. 2801) 2017-07-19 (updated 2019-09-15) (3 minutes)
• The tangent of the sum of two angles (p. 2191) 2018-04-27 (1 minute)
• Caustic simulation (p. 1454) 2018-09-10 (updated 2018-11-04) (2 minutes)
• Quintic upsampling of time-series with 1½ multiplies per sample (p. 2844) 2018-10-28 (2 minutes)
• Time domain analog chaos (p. 2198) 2018-10-28 (4 minutes)
• Sparse filters (p. 834) 2018-12-02 (4 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Accelerating Euler's Method on linear time-invariant systems by exponentiating matrices (p. 348) 2019-03-24 (updated 2019-04-02) (7 minutes)
• Solving initial-value problems faster and with guaranteed error bounds with affine arithmetic (p. 836) 2019-04-02 (5 minutes)
• Karatsuba (p. 2090) 2019-04-20 (2 minutes)
• When should you give up waiting for the bus and just walk? (p. 2280) 2019-04-24 (5 minutes)
• Dercuano formula display (p. 495) 2019-04-30 (5 minutes)
• Why the Cartesian product of fields isn't a field (p. 2660) 2019-05-02 (2 minutes)
• An algebra of textures for interactive composition (p. 1283) 2019-05-08 (4 minutes)
• On the method of finite differences used in Babbage's Difference Engine (p. 827) 2019-05-31 (6 minutes)
• Midpoint method texture mapping (p. 1837) 2019-06-01 (3 minutes)
• Smooth hysteresis (p. 422) 2019-06-11 (13 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)
• Fermat primes (p. 1451) 2019-07-07 (4 minutes)
• A failed attempt to make squares cheaper to compute (p. 1622) 2019-07-09 (updated 2019-07-11) (4 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• Human memorable secret sharing (p. 426) 2019-08-10 (2 minutes)
• The miraculous low-rank SVD approximate convolution algorithm

(p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Complex linear regression (in the field $\mathbb{C}$ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Some notes on the landscape of linear optimization software and applications (p. 1285) 2019-08-21 (updated 2019-08-25) (35 minutes)
• Differentiable neighborhood regression (p. 2944) 2019-08-31 (15 minutes)
• Everything is money? (p. 1859) 2019-08-31 (4 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• Cloth structure from shading (p. 84) 2019-09-01 (2 minutes)
• Processing halftoning (p. 915) 2019-09-01 (15 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)
• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)
• A formal language for defining implicitly parameterized functions (p. 144) 2019-09-05 (updated 2019-09-30) (29 minutes)
• Pythagorean cement pipes for your shower singing (p. 156) 2019-09-08 (updated 2019-09-09) (7 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)
• Sparse sinc (p. 1880) 2019-09-15 (10 minutes)
• Hadamard rhythms (p. 831) 2019-11-01 (6 minutes)
• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)
• Interval raymarching (p. 1342) 2019-11-02 (updated 2019-11-10) (6 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)
• Rediscovering successive parabolic interpolation: derivative-free optimization of scalar functions by fitting a parabola (p. 727) 2019-11-26 (updated 2019-11-27) (8 minutes)

Notes concerning “MathJax”

• Dercuano formula display (p. 495) 2019-04-30 (5 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)

Notes concerning “Mechanical computation”

• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Nobody has yet constructed a mechanical universal digital computer (p. 1053) 2014-04-24 (6 minutes)
• An extremely simple electromechanical state machine (p. 50) 2014-04-24 (16 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Simple state machines (p. 760) 2016-09-19 (updated 2016-09-24) (8 minutes)
• Non-inverting logic (p. 861) 2017-02-18 (updated 2019-07-20) (8 minutes)
• Computation with strain (p. 2812) 2019-06-13 (17 minutes)

Notes concerning "Mechanical things"

• Vanagon mail (p. 2032) 2007 to 2009 (3 minutes)
• Nobody has yet constructed a mechanical universal digital computer (p. 1053) 2014-04-24 (6 minutes)
• Planar lookup tables (p. 3105) 2014-04-24 (2 minutes)
• An extremely simple electromechanical state machine (p. 50) 2014-04-24 (16 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Slotted tape with skewed involute roulette bristles as an alternative to hose clamps and possibly screws (p. 395) 2014-07-02 (6 minutes)
• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• A one-motor robot (p. 118) 2015-09-03 (13 minutes)
• Tapered thread (p. 363) 2015-09-03 (updated 2019-06-10) (4 minutes)
• Hot wire saw (p. 3159) 2015-12-28 (updated 2019-06-02) (10 minutes)
• Material merits (p. 1496) 2016-05-08 (6 minutes)
• Spring energy density (p. 3106) 2016-05-28 (updated 2016-06-06) (13 minutes)
• Mechanical buck converter (p. 1876) 2016-06-20 (5 minutes)
• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• Phosphorescent laser display (p. 1987) 2016-08-16 (8 minutes)
• Hot oil cutter (p. 3287) 2016-08-16 (updated 2016-08-17) (8 minutes)
• Pulley generator (p. 3148) 2016-09-05 (2 minutes)
• Spring energy density (p. 1010) 2016-09-05 (updated 2019-04-20) (3 minutes)
• Simple state machines (p. 760) 2016-09-19 (updated 2016-09-24) (8 minutes)
• Cross current zone melting (p. 1872) 2016-10-06 (1 minute)
• Clay fabrication objectives (p. 2111) 2017-01-16 (updated 2017-01-17) (3 minutes)
• Differential spiral cam (p. 512) 2017-07-19 (9 minutes)
• Piezoelectric engraving (p. 1070) 2017-07-19 (4 minutes)
• Pipe dome (p. 3068) 2017-07-19 (7 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Hammering toolhead (p. 3297) 2017-08-18 (6 minutes)
• Three phase oscillating belt (p. 214) 2018-10-28 (4 minutes)
• Paper/foil relays (p. 3273) 2019-04-02 (updated 2019-10-23) (13 minutes)
• Plastic cutters (p. 1074) 2019-04-20 (5 minutes)
• Drone cutting (p. 1106) 2019-06-11 (12 minutes)
• Computation with strain (p. 2812) 2019-06-13 (17 minutes)

• Foil origami robots (p. 2286) 2019-06-13 (updated 2019-06-14) (10 minutes)
• Spiral chinese windlass (p. 2915) 2019-07-23 (updated 2019-07-24) (7 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29) (31 minutes)
• Harmonic motion chain robot (p. 2197) 2019-08-16 (2 minutes)
• Rubber wheel pinch drive (p. 2912) 2019-08-16 (updated 2019-08-18) (8 minutes)
• Gold leaf trusses (p. 3055) 2019-08-31 (11 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)
• Lenticular deflector (p. 2612) 2019-09-08 (updated 2019-09-09) (9 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Shaped hammer face giant pressure (p. 3278) 2019-11-10 (21 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Oval cam lock (p. 3060) 2019-11-26 (5 minutes)

Notes concerning “Memex”

• Writing hypertext is still a pain (p. 715) 2016-02-18 (6 minutes)
• Quasicard: a hypothetical reimagining of HyperCard and TiddlyWiki (p. 416) 2017-04-18 (updated 2017-06-09) (18 minutes)

Notes concerning “Memory models”

• Simple system language (p. 1659) 2013-05-17 (7 minutes)
• Constant-space grep (p. 296) 2014-02-24 (3 minutes)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• Linear trees (p. 1811) 2016-05-19 (updated 2016-05-20) (6 minutes)
• Similarities between Golang and Rust (p. 1523) 2017-01-11 (updated 2017-01-17) (7 minutes)
• Cartesian product storage (p. 3012) 2017-03-20 (3 minutes)
• Parametric polymorphism and columns (p. 1835) 2017-07-19 (2 minutes)
• The Z-machine memory model (p. 2903) 2017-07-19 (4 minutes)
• Constant space flexible data (p. 352) 2018-04-27 (5 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Constant space lists (p. 3062) 2018-12-10 (10 minutes)
• India rubber memory (p. 579) 2019-03-19 (4 minutes)
• A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)

Notes concerning “Merkle DAGs”

• ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
• Distributed computing environment (p. 776) 2017-07-19 (8 minutes)

Notes concerning “Messaging”

• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)

Notes concerning “Metaballs”

• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• Convolution surface plotting (p. 2264) 2015-09-03 (updated 2015-09-13) (2 minutes)

Notes concerning “Metallurgy”

• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• Vitruvius could have taken photographs (p. 992) 2016-07-30 (1 minute)
• Immersion plating of copper on iron with blue vitriol (p. 1099) 2016-09-24 (8 minutes)
• Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)

Notes concerning “Metamaterials”

- Elastic metamaterials (p. 719) 2019-03-19 (17 minutes)
- Gold leaf trusses (p. 3055) 2019-08-31 (11 minutes)
- Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)

Notes concerning “Method of secants”

• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Rediscovering successive parabolic interpolation: derivative-free optimization of scalar functions by fitting a parabola (p. 727) 2019-11-26 (updated 2019-11-27) (8 minutes)

Notes concerning “Metrology”

• Charge transfer servo (p. 3017) 2013-05-17 (2 minutes)
• The Tinkerer’s Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Optical lever thermometer (p. 2624) 2015-09-03 (1 minute)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Solar-cell Geiger counters (p. 3241) 2016-07-30 (1 minute)
• Augmenting a slow precise ADC with a sloppy fast high-pass filtered parallel ADC (p. 1469) 2017-03-20 (2 minutes)
• Disk oscilloscope (p. 713) 2017-04-10 (updated 2017-06-20) (3 minutes)
• TV oscilloscope (p. 1253) 2017-04-10 (updated 2017-06-20) (4 minutes)
• Laser printer oscilloscope (p. 449) 2017-04-18 (updated 2017-06-20) (2 minutes)
• Can a simple nonlinear VCO enable super cheap oscilloscopes? (p. 1357) 2017-05-04 (updated 2017-05-10) (5 minutes)
• VCR oscilloscope (p. 213) 2017-05-10 (updated 2017-06-20) (2 minutes)
• CCD oscilloscope (p. 1861) 2017-06-20 (updated 2017-07-04) (7 minutes)
• Arduino curve tracer (p. 591) 2018-06-17 (10 minutes)
• Turning off the power supply for every sample to reduce noise (p. 239) 2018-06-18 (2 minutes)
• Measuring submicron displacements by pitch bending a slide guitar (p. 905) 2019-05-05 (18 minutes)
• Inductor thermocouple sensing (p. 2037) 2019-06-01 (21 minutes)
• Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033) 2019-07-16 (updated 2019-07-17) (28 minutes)
• Phase relations (p. 2200) 2019-07-23 (updated 2019-07-24) (4 minutes)

Notes concerning "Microcontrollers"

• Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
• Programming paradigms for tiny microcontrollers (p. 2104) 2007 to 2009 (6 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• The Tinkerer's Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)
• Randomizing delta-sigma conversion to eliminate aliasing (p. 2834) 2014-04-24 (7 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• How can we build an efficient microcontroller-based amplifier? (p. 2821) 2016-07-13 (5 minutes)
• Jellybean ICs 2016 (p. 817) 2016-07-14 (updated 2019-05-05) (17 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• Current hardware trends tend toward raytracing (p. 1351) 2016-10-07 (4 minutes)
• The Magic Kazoo: a synthesizer you stick in your mouth (p. 1873) 2017-04-04 (updated 2019-05-12) (6 minutes)
• Minimum hardware and software to get a flexible notetaking device running (p. 535) 2017-04-28 (4 minutes)
• How can we do online pitch detection? (p. 1869) 2018-04-27 (updated 2018-04-30) (6 minutes)
• Arduino curve tracer (p. 591) 2018-06-17 (10 minutes)
• Transistors vs. Microcontrollers (p. 2918) 2018-06-17 (updated 2018-07-05) (8 minutes)
• The TWI and I²C buses and better alternatives like CAN and RS-485 (p. 1638) 2018-06-28 (updated 2018-07-05) (24 minutes)
• The Adafruit Feather (p. 2966) 2018-06-30 (1 minute)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)

• Microlens vibrating lightfield (p. 1219) 2018-07-14 (updated 2018-07-15) (11 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• Audio logic analyzer (p. 1801) 2019-11-12 (3 minutes)

Notes concerning “Microprint”

• Full res globe (p. 1255) 2014-02-24 (1 minute)
• Holographic archival (p. 766) 2014-04-24 (10 minutes)
• Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
• Quadratic opacity holograms (p. 3073) 2015-09-03 (7 minutes)
• Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
• Microprint visor (p. 523) 2016-09-07 (2 minutes)
• Piezoelectric engraving (p. 1070) 2017-07-19 (4 minutes)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)

Notes concerning “Microscopy”

- Compressed sensing microscope (p. 306) 2016-10-06 (7 minutes)
- Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
- Debokehfication (p. 473) 2019-09-01 (updated 2019-09-12) (4 minutes)

Notes concerning “Mill”

• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Generalizing my RPN calculator to support refactoring (p. 969) 2016-10-17 (12 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Golomb-coding operands as belt offsets likely won’t increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

• Compact code cpu (p. 397) 2017-07-19 (3 minutes)
• A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)

Notes concerning "miniKANREN"

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)

Notes concerning “Minsky algorithm”

• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)

Notes concerning “Minimal Instruction Set Computing”

• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Golomb-coding operands as belt offsets likely won’t increase code density much (p. 1605) 2017-06-15 (updated 2017-06-20) (6 minutes)

Notes concerning “Moon”

• Can you read the lunar lander’s plaque from Earth? Or write a new one? (p. 125) 2015-09-03 (9 minutes)
• Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias (p. 309) 2019-04-10 (updated 2019-04-16) (2 minutes)

Notes concerning “Morphology”

• Dilating letterforms (p. 651) 2018-11-04 (15 minutes)
• Some notes on morphology, including improvements on Urbach and Wilkinson’s erosion/dilation algorithm (p. 216) 2019-01-04 (updated 2019-11-12) (26 minutes)
• Median filtering (p. 3155) 2019-01-17 (11 minutes)
• Tabulating your top event of the month efficiently using RMQ algorithms (p. 619) 2019-03-19 (8 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)

Notes concerning “Multiplication”

- Multiplication with squares (p. 1983) 2017-07-19 (updated 2019-07-09) (5 minutes)
- Hardware multiplication with square tables (p. 1886) 2019-02-08 (updated 2019-07-09) (4 minutes)
- Karatsuba (p. 2090) 2019-04-20 (2 minutes)

Notes concerning “Multitouch”

• drag-and-drop calculator for touch devices (p. 1045) 2015-09-03 (5 minutes)
• Interactive calculator (p. 2771) 2018-04-26 (16 minutes)
• Interactive geometry (p. 508) 2018-04-26 (1 minute)
• Two-thumb quasimodal multitouch interaction techniques (p. 1765) 2018-04-26 (11 minutes)
• Multitouch livecoding (p. 122) 2018-06-17 (1 minute)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• First impressions on using the µMath+ calculator program for Android (p. 195) 2019-05-21 (13 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• Multitouch and accelerometer puppeteering (p. 1785) 2019-08-29 (updated 2019-09-01) (12 minutes)
• Audio tablet (p. 2178) 2019-09-28 (7 minutes)

Notes concerning “The MuP21 MISC microcontroller”

• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)

Notes concerning “Music”

• Alastair thesis review (p. 1784) 2013-05-17 (1 minute)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Would Synthgramelodia make a good base for livecoding music? (p. 2540) 2015-09-03 (8 minutes)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Virtual instruments (p. 658) 2015-11-09 (3 minutes)
• The Magic Kazoo: a synthesizer you stick in your mouth (p. 1873) 2017-04-04 (updated 2019-05-12) (6 minutes)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Multitouch livecoding (p. 122) 2018-06-17 (1 minute)
• Sample reversal (p. 1353) 2018-12-18 (updated 2019-01-17) (5 minutes)
• Honk development (p. 1188) 2019-03-21 (2 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Measuring submicron displacements by pitch bending a slide guitar (p. 905) 2019-05-05 (18 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Pythagorean cement pipes for your shower singing (p. 156) 2019-09-08 (updated 2019-09-09) (7 minutes)
• Hadamard rhythms (p. 831) 2019-11-01 (6 minutes)
• Audio logic analyzer (p. 1801) 2019-11-12 (3 minutes)
• Applying FM synthesis to natural sounds such as voices (p. 596) 2019-11-12 (2 minutes)

Notes concerning “Networking”

• A survey of small TCP/IP implementations (p. 2616) 2007 to 2009 (4 minutes)
• The internet is probably not going to collapse for economic reasons (p. 3194) 2016-09-06 (9 minutes)
• Web prefetch (p. 3046) 2017-06-15 (1 minute)
• A REST interface to a software transactional memory (p. 3014) 2017-06-21 (2 minutes)
• Interactive bandwidth (p. 779) 2017-08-03 (2 minutes)
• How inefficient is SNAT hole-punching via random port trials? (p. 1155) 2018-04-27 (2 minutes)
• Notes on SIP VoIP in 2019 (p. 1064) 2019-06-07 (updated 2019-06-28) (8 minutes)

Notes concerning “Newton–Raphson iteration (“Newton’s method”)”

• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• Rediscovering successive parabolic interpolation: derivative-free optimization of scalar functions by fitting a parabola (p. 727) 2019-11-26 (updated 2019-11-27) (8 minutes)

Notes concerning “Non-imaging optics”

• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)

Notes concerning “Natural-language processing”

• Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• Alphanumerenglish (p. 2882) 2015-04-06 (updated 2016-07-27) (6 minutes)
• Toward a language for hacking around with natural-language processing algorithms (p. 1681) 2016-09-08 (7 minutes)
• Surrealist code (p. 1325) 2016-10-11 (3 minutes)
• English diphones (p. 2061) 2019-12-03 (5 minutes)

Notes concerning “Noise”

- Digital noise generators (p. 1137) 2018-10-28 (2 minutes)
- Time domain analog chaos (p. 2198) 2018-10-28 (4 minutes)

Notes concerning “Nuclear”

• Solar-cell Geiger counters (p. 3241) 2016-07-30 (1 minute)
• Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times? (p. 1027) 2016-09-06 (3 minutes)
• Lithium fission energy (p. 3285) 2016-09-06 (updated 2019-09-16) (6 minutes)

Notes concerning “Numpy”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)
• Antialiased line drawing (p. 1803) 2018-11-13 (updated 2019-09-01) (4 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)

Notes concerning “Oberon”

• What does a futuristic OS look like? (p. 2163) 2017-08-18 (updated 2019-05-05) (6 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• A review of Wirth’s Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)

Notes concerning “OCaml”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• OMeta contains Wadler's "Views" (p. 842) 2007 to 2009 (updated 2019-05-20) (13 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Simple system language (p. 1659) 2013-05-17 (7 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• B-Tree ropes (p. 2762) 2019-09-24 (updated 2019-09-25) (19 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)

Notes concerning “ODEs”

• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• Accelerating Euler’s Method on linear time-invariant systems by exponentiating matrices (p. 348) 2019-03-24 (updated 2019-04-02) (7 minutes)

Notes concerning “OLAP”

• lattices, powersets, bitstrings, and efficient OLAP (p. 2345) 2014-04-24 (17 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)

Notes concerning “OMeta”

• A survey of small TCP/IP implementations (p. 2616) 2007 to 2009 (4 minutes)
• OMeta contains Wadler's "Views" (p. 842) 2007 to 2009 (updated 2019-05-20) (13 minutes)
• Tagging parsers (p. 208) 2018-11-23 (updated 2018-12-10) (9 minutes)

Notes concerning “Object-oriented programming”

• Nested inheritance (p. 340) 2007 to 2009 (2 minutes)
• OMeta contains Wadler's "Views" (p. 842) 2007 to 2009 (updated 2019-05-20) (13 minutes)
• When and why exactly should your code “tell, not ask”? That is, use CPS? (p. 1051) 2014-01-08 (4 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)
• Similarities between Golang and Rust (p. 1523) 2017-01-11 (updated 2017-01-17) (7 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Byte-stream pipe and antipipe façade objects for editor buffers (p. 950) 2017-04-10 (3 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Immediate mode productive grammars (p. 898) 2018-09-13 (8 minutes)

Notes concerning “Opacity holograms”

- Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
- Quadratic opacity holograms (p. 3073) 2015-09-03 (7 minutes)
- Opacity holograms (p. 1448) 2016-08-16 (8 minutes)
- Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
- Compressed sensing microscope (p. 306) 2016-10-06 (7 minutes)

Notes concerning “Operating systems”

Notes concerning “Optics”

• Holographic archival (p. 766) 2014-04-24 (10 minutes)
• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
• Optical lever thermometer (p. 2624) 2015-09-03 (1 minute)
• Can you read the lunar lander’s plaque from Earth? Or write a new one? (p. 125) 2015-09-03 (9 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Phosphorescent laser display (p. 1987) 2016-08-16 (8 minutes)
• Starfield servo (p. 1709) 2016-08-30 (updated 2018-11-07) (13 minutes)
• Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)
• Microprint visor (p. 523) 2016-09-07 (2 minutes)
• Compressed sensing microscope (p. 306) 2016-10-06 (7 minutes)
• 3-D printing glass with continuously varying refractive indices for optics without internal surfaces (p. 1156) 2016-10-06 (3 minutes)
• Jello printing (p. 2426) 2016-12-14 (8 minutes)
• Bubble display (p. 1542) 2017-01-24 (updated 2017-08-03) (1 minute)
• Sparkle wheel display (p. 1738) 2017-05-10 (6 minutes)
• Flying spot reilluminatable stage (p. 2358) 2017-05-15 (1 minute)
• CCD oscilloscope (p. 1861) 2017-06-20 (updated 2017-07-04) (7 minutes)
• Microlens vibrating lightfield (p. 1219) 2018-07-14 (updated 2018-07-15) (11 minutes)
• Caustics (p. 1619) 2018-08-18 (updated 2019-11-08) (8 minutes)
• You can’t construct optical systems with arbitrary light transfers, but you can do some awesome shit (p. 981) 2018-09-10 (11 minutes)
• Electrolytic anodizing, with a small movable electrode (p. 3059) 2018-10-28 (2 minutes)
• Macroscopic capacitive DLP (p. 1834) 2019-04-08 (1 minute)
• Caustic business card (p. 255) 2019-04-08 (3 minutes)
• Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias (p. 309) 2019-04-10 (updated 2019-04-16) (2 minutes)
• Analemma sundial (p. 1955) 2019-07-05 (11 minutes)
• Some extensions of William Beaty’s scratch holograms (p. 2536) 2019-07-11 (9 minutes)
• Debokehfication (p. 473) 2019-09-01 (updated 2019-09-12) (4 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)
• Bokeh pointcasting (p. 92) 2019-09-08 (updated 2019-09-09) (16 minutes)
• Lenticular deflector (p. 2612) 2019-09-08 (updated 2019-09-09) (9 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24

(16 minutes)
• Camera flash extrapolation (p. 2792) 2019-11-12 (6 minutes)
• Kerr snow display (p. 3220) 2019-11-12 (3 minutes)

Notes concerning “Mathematical optimization”

• Modeling trees with slices containing metaballs (p. 2619) 2014-06-29 (updated 2014-07-02) (6 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• Opacity holograms (p. 1448) 2016-08-16 (8 minutes)
• Kinect modeling (p. 164) 2016-09-16 (1 minute)
• Reconstructing a 3-D Lambertian surface from video with a moving light source (p. 3296) 2016-09-16 (1 minute)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• One-line thoughts that don’t merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• High-precision control of low-stiffness sytems with bounded-Q resonances (p. 1002) 2017-05-29 (updated 2017-06-01) (4 minutes)
• Affine arithmetic optimization (p. 2801) 2017-07-19 (updated 2019-09-15) (3 minutes)
• Optimization-based painting software (p. 1158) 2018-04-27 (1 minute)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Gradient descent beyond machine learning (p. 2310) 2018-05-18 (2 minutes)
• Caustic simulation (p. 1454) 2018-09-10 (updated 2018-11-04) (2 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• Complex linear regression (in the field $\mathbb{C}$ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Some notes on the landscape of linear optimization software and applications (p. 1285) 2019-08-21 (updated 2019-08-25) (35 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)

• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)
• Approximate optimization (p. 517) 2019-11-13 (3 minutes)
• Rediscovering successive parabolic interpolation: derivative-free optimization of scalar functions by fitting a parabola (p. 727) 2019-11-26 (updated 2019-11-27) (8 minutes)
• English diphones (p. 2061) 2019-12-03 (5 minutes)
• Nomadic furniture optimization (p. 1658) 2019-12-15 (2 minutes)

Notes concerning “Optimum trits”

• In what sense is e the optimal branching factor, and what does it mean for menu tree design? (p. 2483) 2012-12-04 (3 minutes)
• Precisely how is 3 “optimal” for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)

Notes concerning “Oscilloscopes”

• Argentine oscilloscope pricing 2016 (p. 1965) 2016-08-16 (4 minutes)
• Augmenting a slow precise ADC with a sloppy fast high-pass filtered parallel ADC (p. 1469) 2017-03-20 (2 minutes)
• Disk oscilloscope (p. 713) 2017-04-10 (updated 2017-06-20) (3 minutes)
• TV oscilloscope (p. 1253) 2017-04-10 (updated 2017-06-20) (4 minutes)
• Laser printer oscilloscope (p. 449) 2017-04-18 (updated 2017-06-20) (2 minutes)
• Can a simple nonlinear VCO enable super cheap oscilloscopes? (p. 1357) 2017-05-04 (updated 2017-05-10) (5 minutes)
• VCR oscilloscope (p. 213) 2017-05-10 (updated 2017-06-20) (2 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• CCD oscilloscope (p. 1861) 2017-06-20 (updated 2017-07-04) (7 minutes)
• Oscilloscope screens (p. 578) 2018-06-05 (2 minutes)
• Can you turbocharge the STM32 ADC to build an oscilloscope? (p. 137) 2018-07-14 (5 minutes)
• Phase relations (p. 2200) 2019-07-23 (updated 2019-07-24) (4 minutes)

Notes concerning “OpenStreetMap”

- Full res globe (p. 1255) 2014-02-24 (1 minute)
- Fast geographical maps on Android (p. 455) 2015-10-16 (9 minutes)

Notes concerning “Parallelism”

• Schimmler parallelism asymptotic gain (p. 294) 2007 to 2009 (1 minute)
• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• Internal determinism (p. 2803) 2016-08-17 (2 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• Bitsliced operations with a hypercube of shuffle operations (p. 2363) 2016-11-30 (2 minutes)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)

Notes concerning “Parselov”

• Parsing a conservative approximation of a CFG with a FSM (p. 159) 2015-09-03 (7 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)

Notes concerning “Parsing”

• OMeta contains Wadler's "Views" (p. 842) 2007 to 2009 (updated 2019-05-20) (13 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Parsing a conservative approximation of a CFG with a FSM (p. 159) 2015-09-03 (7 minutes)
• DReX and “regular string transformations”: would an RPN DSL work well? (p. 453) 2016-09-19 (3 minutes)
• Secure, self-describing, self-delimiting serialization for Python (p. 243) 2017-04-11 (8 minutes)
• Toward a minimal PEG parsing engine (p. 955) 2018-06-06 (4 minutes)
• Immediate mode productive grammars (p. 898) 2018-09-13 (8 minutes)
• Tagging parsers (p. 208) 2018-11-23 (updated 2018-12-10) (9 minutes)
• How small can we make a comfortable subset of JS? (p. 1348) 2018-11-27 (updated 2018-12-02) (3 minutes)
• Minimal imperative language (p. 2175) 2018-12-10 (7 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)
• Dercuano grinding (p. 2475) 2019-10-01 (12 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)

Notes concerning “Particle filters”

• Interval filters (p. 1282) 2015-09-17 (2 minutes)
• Texture synthesis with spatial-domain particle filters (p. 857) 2016-10-06 (2 minutes)

Notes concerning “Parsing Expression Grammars (PEGs)”

- OMeta contains Wadler's "Views" (p. 842) 2007 to 2009 (updated 2019-05-20) (13 minutes)
- Toward a minimal PEG parsing engine (p. 955) 2018-06-06 (4 minutes)
- Tagging parsers (p. 208) 2018-11-23 (updated 2018-12-10) (9 minutes)
- How small can we make a comfortable subset of JS? (p. 1348) 2018-11-27 (updated 2018-12-02) (3 minutes)

Notes concerning “Performance”

• Bicicleta maps (p. 582) 2007 to 2009 (2 minutes)
• Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• A cute algorithm for card-image templates (p. 1946) 2007 to 2009 (2 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• Git data (p. 2823) 2007 to 2009 (5 minutes)
• Git learnings (p. 3268) 2007 to 2009 (3 minutes)
• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• Schimmler parallelism asymptotic gain (p. 294) 2007 to 2009 (1 minute)
• Maybe Counting Characters in UTF-8 Strings Isn't Fast After All! (p. 2992) 2007 to 2009 (15 minutes)
• Why Thunderbird is inadequate for opening a 7-gigabyte mbox (p. 980) 2007 to 2009 (2 minutes)
• The Problem: Writing With One Access Pattern, Reading With Another (p. 3004) 2007 to 2009 (19 minutes)
• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)
• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• Use crit-bit trees as the fundamental string-set data structure (p. 1498) 2013-05-17 (3 minutes)
• Cycle sort (p. 2344) 2013-05-17 (1 minute)
• Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation (p. 1508) 2013-05-17 (updated 2013-05-20) (21 minutes)
• Instant hypertext (p. 630) 2013-05-17 (updated 2013-05-20) (14 minutes)
• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• Simple persistent in-memory dictionaries with log² lookups and logarithmic insertion (p. 3110) 2014-02-24 (6 minutes)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• lattices, powersets, bitstrings, and efficient OLAP (p. 2345) 2014-04-24 (17 minutes)
• Precisely how is 3 “optimal” for one-hot state machines, sparse FIR

kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• Some speculative thoughts on DSP algorithms (p. 2590) 2014-04-24 (20 minutes)
• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• Rendering iterated function systems (IFSes) with interval arithmetic (p. 2433) 2014-09-02 (6 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• You can't sort a file whose size is cubic in your RAM size in two passes, only quadratic (p. 2311) 2015-05-28 (5 minutes)
• Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
• Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870) 2015-05-28 (updated 2016-06-22) (16 minutes)
• Editor buffers (p. 1328) 2015-07-15 (updated 2015-09-03) (16 minutes)
• Parsing a conservative approximation of a CFG with a FSM (p. 159) 2015-09-03 (7 minutes)
• Storing CSV records in minimal memory in Java (p. 524) 2015-09-03 (6 minutes)
• Memoize the stack (p. 2021) 2015-09-03 (5 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Ternary mergesort (p. 2161) 2015-09-03 (2 minutes)
• Very fast FIR filtering with time-domain zero stuffing and splines (p. 1146) 2015-09-03 (updated 2015-09-07) (13 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• Bitstream dsp (p. 3153) 2015-09-03 (updated 2019-06-23) (3 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Viral wiki (p. 1024) 2015-10-15 (3 minutes)
• A type-inferred dialect of JS (p. 265) 2016-04-22 (4 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Gaussian spline reconstruction (p. 656) 2016-06-05 (updated 2016-06-06) (5 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• Append only unique string pool (p. 1797) 2016-07-27 (2 minutes)
• Internal determinism (p. 2803) 2016-08-17 (2 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• An almost-in-place mergesort (p. 740) 2016-09-07 (5 minutes)
• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17

(updated 2017-04-18) (23 minutes)
• Gradient rendering (p. 583) 2016-09-24 (11 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• What's the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• Chintzy depth of field (p. 629) 2016-10-27 (1 minute)
• Bitsliced operations with a hypercube of shuffle operations (p. 2363) 2016-11-30 (2 minutes)
• The paradoxical complexity of computing the top N (p. 1890) 2017-01-04 (7 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• Set hashing (p. 2485) 2017-03-09 (9 minutes)
• The continuous-web press and the continuous press of the World-Wide Web (p. 1134) 2017-03-20 (6 minutes)
• Amnesic hash tables for stochastically LRU memoization (p. 502) 2017-04-03 (1 minute)
• The history of NoSQL and dbm (p. 45) 2017-04-10 (16 minutes)
• Incremental persistent binary array sets (p. 1008) 2017-04-10 (4 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)
• Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• What's wrong with CoAP (p. 560) 2017-06-15 (3 minutes)
• Web prefetch (p. 3046) 2017-06-15 (1 minute)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Compressing a screen update with a tree of dirty bits (p. 303) 2017-06-21 (1 minute)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Double heap sequence (p. 2521) 2017-07-19 (2 minutes)
• Parametric polymorphism and columns (p. 1835) 2017-07-19 (2 minutes)
• Vector instructions (p. 2977) 2017-07-19 (2 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)
• Multiplication with squares (p. 1983) 2017-07-19 (updated 2019-07-09) (5 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• Another candidate lightweight frequency tracking algorithm (p. 2069) 2017-08-18 (4 minutes)
• A minimal dependency processing system (p. 911) 2017-09-21 (3 minutes)
• Framed-belt DSP (p. 3095) 2018-04-27 (3 minutes)
• Mail reader (p. 3290) 2018-04-27 (updated 2018-06-18) (7 minutes)

• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)

• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)
• Word stream architecture (p. 2215) 2018-06-17 (13 minutes)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• The TWI and I²C buses and better alternatives like CAN and RS-485 (p. 1638) 2018-06-28 (updated 2018-07-05) (24 minutes)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Comparable counters (p. 2441) 2018-08-16 (1 minute)
• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Bit difference array (p. 1748) 2018-10-28 (10 minutes)
• Quintic upsampling of time-series with 1½ multiplies per sample (p. 2844) 2018-10-28 (2 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033) 2018-10-28 (updated 2019-05-05) (3 minutes)
• Cheap textures (p. 736) 2018-10-28 (updated 2019-05-05) (5 minutes)
• The details of the GPU in this laptop (p. 2970) 2018-10-29 (2 minutes)
• Recurrent comb cascade (p. 483) 2018-11-09 (updated 2018-11-10) (2 minutes)
• Fast gsave (p. 1593) 2018-11-27 (5 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• Improving Lua #L with incremental prefix sum in the ∧ monoid (p. 2008) 2018-12-18 (7 minutes)
• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)
• Median filtering (p. 3155) 2019-01-17 (11 minutes)
• Transactional event handlers (p. 139) 2019-01-24 (14 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)
• Fast secure pubsub (p. 545) 2019-02-04 (updated 2019-12-03)

(2 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)
• Hardware multiplication with square tables (p. 1886) 2019-02-08 (updated 2019-07-09) (4 minutes)
• Karatsuba (p. 2090) 2019-04-20 (2 minutes)
• An algebra of textures for interactive composition (p. 1283) 2019-05-08 (4 minutes)
• Granite texture (p. 1991) 2019-05-08 (updated 2019-05-09) (5 minutes)
• Dercuano rendering (p. 2300) 2019-05-11 (updated 2019-05-12) (3 minutes)
• Dercuano backlinks (p. 475) 2019-05-22 (7 minutes)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Midpoint method texture mapping (p. 1837) 2019-06-01 (3 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Kernel code generation (p. 2302) 2019-07-02 (6 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Complex linear regression (in the field ℂ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Query evaluation with interval-annotated trees over sequences (p. 1423) 2019-08-30 (updated 2019-09-03) (30 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)
• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)
• B-Tree ropes (p. 2762) 2019-09-24 (updated 2019-09-25) (19 minutes)
• Is there an incremental union find algorithm? (p. 1602) 2019-10-01 (8 minutes)
• Interval raymarching (p. 1342) 2019-11-02 (updated 2019-11-10)

(6 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)
• Approximate optimization (p. 517) 2019-11-13 (3 minutes)
• Magic sinewave filter (p. 200) 2019-12-17 (6 minutes)
• Sorting in logic (p. 498) 2019-12-28 (2 minutes)

Notes concerning “Phase change materials”

- Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
- Phase-change heat reservoirs for household climate control (p. 2257) 2016-06-14 (updated 2016-06-17) (13 minutes)
- Passivhaus seasonal thermal store (p. 1723) 2017-03-02 (updated 2017-03-07) (2 minutes)
- Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
- Radiant heating (p. 493) 2018-05-20 (3 minutes)
- Hot water bottles (p. 3066) 2018-07-14 (4 minutes)
- A phase-change soldering iron (p. 2270) 2019-05-08 (updated 2019-05-09) (14 minutes)
- Phase change unplugged oven (p. 1433) 2019-12-15 (0 minutes)

Notes concerning “Philosophy”

• Plato was not particularly democratic; ἄρχειν is not “participating in politics” (p. 1707) 2014-04-24 (5 minutes)
• Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)

Notes concerning “Phonetics”

• Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
• Alphanumerenglish (p. 2882) 2015-04-06 (updated 2016-07-27) (6 minutes)
• English diphones (p. 2061) 2019-12-03 (5 minutes)

Notes concerning “Photosynthesis”

- Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
- Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)

Notes concerning “Physical computation”

• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Notes on 3-D printing a mechanical LUT (p. 1326) 2014-04-24 (3 minutes)
• Nobody has yet constructed a mechanical universal digital computer (p. 1053) 2014-04-24 (6 minutes)
• Planar lookup tables (p. 3105) 2014-04-24 (2 minutes)
• An extremely simple electromechanical state machine (p. 50) 2014-04-24 (16 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Transmission line computer (p. 509) 2016-07-11 (updated 2019-07-23) (7 minutes)
• Matrix memory (p. 503) 2016-07-27 (1 minute)
• Transmission line diode computation (p. 3090) 2016-07-30 (3 minutes)
• Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times? (p. 1027) 2016-09-06 (3 minutes)
• Simple state machines (p. 760) 2016-09-19 (updated 2016-09-24) (8 minutes)
• Nonlinear differential amplification (p. 2949) 2016-12-14 (2 minutes)
• Non-inverting logic (p. 861) 2017-02-18 (updated 2019-07-20) (8 minutes)
• Diode logic (p. 272) 2018-06-17 (16 minutes)
• Snap logic (p. 2580) 2018-06-17 (3 minutes)
• Parallel register file (p. 2952) 2018-11-27 (2 minutes)
• What can you build out of 256-byte ROMs? (p. 1468) 2018-12-02 (1 minute)
• Turning a delay line into a counter with a FSM (p. 1680) 2018-12-10 (1 minute)
• Paper/foil relays (p. 3273) 2019-04-02 (updated 2019-10-23) (13 minutes)
• Hall-effect Wheatstone bridges for impractical steampunk electronic logic gates (p. 2351) 2019-04-24 (2 minutes)
• Smooth hysteresis (p. 422) 2019-06-11 (13 minutes)
• Computation with strain (p. 2812) 2019-06-13 (17 minutes)
• Nonconductive relays (p. 3262) 2019-11-12 (3 minutes)
• High temperature semiconductors (p. 2436) 2019-12-01 (2 minutes)

Notes concerning "Physics"

• Food miles imply insignificant energy costs (p. 2187) 2007 to 2009 (4 minutes)
• The AL programming language, dimensional analysis, and typing: do different dimensions really exist? (p. 731) 2007 to 2009 (2 minutes)

• Pensamientos acerca de diseñar un calefón solar (p. 117) 2012-10-15 (2 minutes)
• Más pensamientos acerca de diseñar un calefón solar (p. 1713) 2012-10-15 (5 minutes)
• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder (p. 527) 2013-05-17 (5 minutes)
• Cheap shit ultrawideband (p. 2776) 2013-05-17 (10 minutes)
• Evaporation chimney (p. 2147) 2013-05-17 (13 minutes)
• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Saturation detector (p. 1588) 2013-05-17 (3 minutes)
• Time domain lightning triggering (p. 2534) 2013-05-17 (4 minutes)
• A unicast phased-array ultrasonic "radio" (p. 3077) 2013-05-17 (4 minutes)
• A resistive-capacitive trackpad made from garbage and three ADC microcontroller pins (p. 852) 2013-05-17 (updated 2013-05-20) (17 minutes)
• Ultraslow radio for resilient global communication (p. 2071) 2013-05-17 (updated 2013-05-20) (26 minutes)
• Bottle washing (p. 921) 2014-04-24 (7 minutes)
• Holographic archival (p. 766) 2014-04-24 (10 minutes)
• Inflatable stool (p. 1047) 2014-04-24 (6 minutes)
• Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
• Slotted tape with skewed involute roulette bristles as an alternative to hose clamps and possibly screws (p. 395) 2014-07-02 (6 minutes)
• Keyboard-powered computers (p. 2220) 2014-10-25 (updated 2018-10-28) (26 minutes)
• A mechano-optical vector display for animation archival (p. 3047) 2014-12-28 (updated 2015-09-03) (28 minutes)
• Can you read the lunar lander's plaque from Earth? Or write a new one? (p. 125) 2015-09-03 (9 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Tapered thread (p. 363) 2015-09-03 (updated 2019-06-10) (4 minutes)
• Hot wire saw (p. 3159) 2015-12-28 (updated 2019-06-02) (10 minutes)
• Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)

• Spring energy density (p. 3106) 2016-05-28 (updated 2016-06-06) (13 minutes)
• Phase-change heat reservoirs for household climate control (p. 2257) 2016-06-14 (updated 2016-06-17) (13 minutes)
• Mechanical buck converter (p. 1876) 2016-06-20 (5 minutes)
• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• Flux deposition for 3-D printing in glass and metals (p. 1366) 2016-07-03 (15 minutes)
• How would you maximize the energy density of a capacitor? (p. 42) 2016-07-27 (5 minutes)
• Coinductive keyboard (p. 1893) 2016-07-30 (4 minutes)
• Calculations about desalination in Israel (p. 2827) 2016-08-11 (3 minutes)
• Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
• Starfield servo (p. 1709) 2016-08-30 (updated 2018-11-07) (13 minutes)
• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• Spring energy density (p. 1010) 2016-09-05 (updated 2019-04-20) (3 minutes)
• Digital logic with lasers, induced X-ray emission, and neutron-induced fission, for femtosecond switching times? (p. 1027) 2016-09-06 (3 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Sun cutter (p. 56) 2016-09-06 (9 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)
• Mic energy harvesting (p. 2583) 2016-09-07 (updated 2016-09-08) (5 minutes)
• License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162) 2016-09-19 (5 minutes)
• Laser ablation of zinc or pewter for printed circuit boards (p. 2799) 2016-09-19 (4 minutes)
• Marking metal surfaces with arcs (p. 1993) 2016-10-06 (4 minutes)
• Spark particulate sieve (p. 2047) 2016-10-06 (updated 2016-10-11) (7 minutes)
• La vibración del hierro, ¿es de baja frecuencia o qué? (p. 2231) 2016-10-07 (3 minutes)
• Analogies between spring–mass–dashpot systems, electrical systems, and fluidic systems (p. 1472) 2016-10-30 (4 minutes)
• Recuperator heat storage (p. 594) 2016-11-01 (updated 2019-08-21) (4 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Passive ultrasound sonar (p. 295) 2016-12-28 (1 minute)
• Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)

• Passivhaus seasonal thermal store (p. 1723) 2017-03-02 (updated 2017-03-07) (2 minutes)
• Passive dehumidifier (p. 3256) 2017-03-20 (14 minutes)
• Ice pants (p. 298) 2017-04-04 (updated 2019-01-22) (17 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
• Fast sea salt evaporator (p. 1087) 2017-06-01 (3 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• FM chirp sonar (p. 351) 2017-07-04 (1 minute)
• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)
• Piezoelectric engraving (p. 1070) 2017-07-19 (4 minutes)
• Rubber air conditioner (p. 2791) 2017-07-19 (2 minutes)
• Energy storage in a personal water tower: pretty impractical (p. 2044) 2017-07-19 (2 minutes)
• Hammering toolhead (p. 3297) 2017-08-18 (6 minutes)
• Bench trash power supply (p. 1457) 2018-04-27 (9 minutes)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)
• Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642) 2018-04-30 (updated 2018-07-05) (29 minutes)
• You can stuff a UHMWPE hammock in your wallet (p. 799) 2018-05-15 (updated 2018-10-28) (11 minutes)
• Heating my apartment with a plastic tub of hot water (p. 1310) 2018-06-17 (3 minutes)
• Lithium battery welder (p. 2846) 2018-06-21 (updated 2019-01-22) (2 minutes)
• Electric hammer (p. 1865) 2018-07-02 (updated 2018-07-05) (14 minutes)
• Notes on circuitry for the Nutra seed activator (p. 3099) 2018-08-16 (20 minutes)
• You can't construct optical systems with arbitrary light transfers, but you can do some awesome shit (p. 981) 2018-09-10 (11 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Friction-cutting plastic (p. 2412) 2019-02-25 (8 minutes)
• Ultralight tunnel personal rapid transit (p. 706) 2019-03-11 (15 minutes)
• Elastic metamaterials (p. 719) 2019-03-19 (17 minutes)
• Paper/foil relays (p. 3273) 2019-04-02 (updated 2019-10-23) (13 minutes)
• Macroscopic capacitive DLP (p. 1834) 2019-04-08 (1 minute)
• Caustic business card (p. 255) 2019-04-08 (3 minutes)
• Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias (p. 309) 2019-04-10 (updated 2019-04-16) (2 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Plastic cutters (p. 1074) 2019-04-20 (5 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21)

(7 minutes)
• Measuring submicron displacements by pitch bending a slide guitar (p. 905) 2019-05-05 (18 minutes)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• Free space optical coding gain (p. 1244) 2019-05-08 (updated 2019-05-09) (4 minutes)
• A phase-change soldering iron (p. 2270) 2019-05-08 (updated 2019-05-09) (14 minutes)
• Drone cutting (p. 1106) 2019-06-11 (12 minutes)
• Computation with strain (p. 2812) 2019-06-13 (17 minutes)
• Foil origami robots (p. 2286) 2019-06-13 (updated 2019-06-14) (10 minutes)
• Analemma sundial (p. 1955) 2019-07-05 (11 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)
• Some extensions of William Beaty's scratch holograms (p. 2536) 2019-07-11 (9 minutes)
• Measuring the moisture content of coffee and other things with dielectric spectroscopy (p. 1033) 2019-07-16 (updated 2019-07-17) (28 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29) (31 minutes)
• Printed circuits on fired-clay ceramic (p. 960) 2019-08-13 (11 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated 2019-08-28) (37 minutes)
• Gold leaf trusses (p. 3055) 2019-08-31 (11 minutes)
• Processing halftoning (p. 915) 2019-09-01 (15 minutes)
• Bokeh pointcasting (p. 92) 2019-09-08 (updated 2019-09-09) (16 minutes)
• Lenticular deflector (p. 2612) 2019-09-08 (updated 2019-09-09) (9 minutes)
• Pythagorean cement pipes for your shower singing (p. 156) 2019-09-08 (updated 2019-09-09) (7 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Shaped hammer face giant pressure (p. 3278) 2019-11-10 (21 minutes)
• Kerr snow display (p. 3220) 2019-11-12 (3 minutes)
• Nonconductive relays (p. 3262) 2019-11-12 (3 minutes)
• Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) 2019-11-24 (2 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Bootstrapping rope bridges and other tensile structures with UHMWPE-bearing drones (p. 2950) 2019-11-25 (5 minutes)
• High temperature semiconductors (p. 2436) 2019-12-01 (2 minutes)

Notes concerning “Plaster”

• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)

Notes concerning “Plating”

- Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
- Immersion plating of copper on iron with blue vitriol (p. 1099) 2016-09-24 (8 minutes)
- Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
- Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)

Notes concerning “Phase-locked loops”

• Karplus–Strong PLLs (p. 2100) 2017-06-09 (1 minute)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)

Notes concerning "Politics"

• Free software debugging (p. 136) 2007 to 2009 (2 minutes)
• Copyright status of the Oxford English Dictionary: relevant data (p. 82) 2007 to 2009 (3 minutes)
• Rich programmers (p. 805) 2007 to 2009 (4 minutes)
• Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
• Dollar auctions and tournaments in human society (p. 884) 2013-05-17 (7 minutes)
• Who is inventing the future in 2013? (p. 897) 2013-05-17 (1 minute)

• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Some personal notes from February 2014 (p. 2134) 2014-02-13 (8 minutes)
• What would a basic income guarantee for Argentina cost? (p. 2409) 2014-04-24 (7 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• Ostinatto (p. 2780) 2014-04-24 (4 minutes)
• What might Diamond-Age-like phyles look like in the real 21st century? (p. 1599) 2014-04-24 (9 minutes)
• Plato was not particularly democratic; ἄρχειν is not "participating in politics" (p. 1707) 2014-04-24 (5 minutes)
• In a world with ubiquitous surveillance, what does politics look like? (p. 1615) 2014-04-24 (11 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• State of the world 2016 (p. 2973) 2016-09-05 (10 minutes)
• Solving the incentive problem for censorship-resistant DHTs (p. 923) 2016-09-07 (updated 2019-05-21) (3 minutes)
• High academic achievement almost certainly depends more on tutoring than group averages by race or sex (p. 113) 2016-09-08 (3 minutes)
• Statement from the Confederation of Teachers (p. 725) 2016-10-11 (updated 2016-10-12) (4 minutes)
• World War III is starting (?) (p. 346) 2016-10-17 (2 minutes)
• Selfish conformity (p. 622) 2016-11-15 (5 minutes)
• The problem is not that people are not turning to real journalism anymore (p. 2934) 2016-11-15 (8 minutes)
• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• The continuous-web press and the continuous press of the World-Wide Web (p. 1134) 2017-03-20 (6 minutes)
• Studies support authority (p. 1541) 2017-04-10 (2 minutes)
• Replicating education (p. 557) 2017-07-19 (7 minutes)
• 2017 [Provisional English translation of intercepted transmission] (p. 2192) 2018-04-27 (updated 2018-07-14) (13 minutes)
• Why is there so much anti-plastic sentiment? Visibility, Arcadian

primitivism, conspicuous consumption, and profit. (p. 3270) 2018-06-21 (7 minutes)
• Ultralight tunnel personal rapid transit (p. 706) 2019-03-11 (15 minutes)
• What are Bitcoin's uses other than sidestepping the law? (p. 2159) 2019-03-11 (updated 2019-07-05) (6 minutes)
• Weregild (p. 3149) 2019-03-24 (3 minutes)
• Notch scorn (p. 115) 2019-04-20 (5 minutes)
• On influencers (p. 660) 2019-05-16 (3 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)
• Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)
• Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) 2019-07-03 (6 minutes)
• The fable of the specialized fox (p. 1076) 2019-08-17 (1 minute)
• Everything is money? (p. 1859) 2019-08-31 (4 minutes)

Notes concerning “Pompous”

• A Sunday in 2014 (p. 1089) 2014-02-24 (3 minutes)
• La vibración del hierro, ¿es de baja frecuencia o qué? (p. 2231) 2016-10-07 (3 minutes)
• Surrealist code (p. 1325) 2016-10-11 (3 minutes)
• Statement from the Confederation of Teachers (p. 725) 2016-10-11 (updated 2016-10-12) (4 minutes)
• Notch scorn (p. 115) 2019-04-20 (5 minutes)
• GPT-2 sets the scene (p. 2978) 2019-11-22 (updated 2019-12-01) (22 minutes)

Notes concerning “Post-scarcity things”

• What would a basic income guarantee for Argentina cost? (p. 2409) 2014-04-24 (7 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)

Notes concerning “Power supplies”

- Lab power supply (p. 2421) 2017-02-21 (updated 2018-06-18) (17 minutes)
- Bench trash power supply (p. 1457) 2018-04-27 (9 minutes)
- Really simple lab power supply (p. 240) 2019-12-10 (7 minutes)

Notes concerning “Predicate logic”

• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• Bayesian and Gricean programming (p. 711) 2015-08-20 (3 minutes)

• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Querying a pile of free-text strings with quasi-Prolog (p. 811) 2017-11-17 (6 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)

Notes concerning “Prefix sums”

• Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• Compression with second-order diffs (p. 2152) 2014-04-24 (3 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Some notes on morphology, including improvements on Urbach and Wilkinson’s erosion/dilation algorithm (p. 216) 2019-01-04 (updated 2019-11-12) (26 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)
• Nonlinear bounded leaky integrator (p. 3150) 2019-09-11 (8 minutes)

Notes concerning "Pricing"

• Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
• A comparison of prices for different forms of energy (p. 3097) 2007 to 2009 (2 minutes)
• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• Polycaprolactone (p. 1813) 2007 to 2009 (3 minutes)
• Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
• The economics of solar energy (p. 1175) 2008 (27 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Food storage (p. 2706) 2013-05-11 (updated 2013-05-17) (54 minutes)
• Illuminating yourself with 10 kilolux of LEDs to combat seasonal affective disorder (p. 527) 2013-05-17 (5 minutes)
• Only a constant factor worse (p. 1648) 2013-05-17 (16 minutes)
• Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
• Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)
• Some personal notes from February 2014 (p. 2134) 2014-02-13 (8 minutes)
• Jim Weirich's death and my daily life (p. 829) 2014-04-24 (5 minutes)
• What would a basic income guarantee for Argentina cost? (p. 2409) 2014-04-24 (7 minutes)
• The future of the human energy market (2014) (p. 1846) 2014-04-24 (19 minutes)
• Building a resilient network out of litter (p. 2107) 2014-04-24 (4 minutes)
• How to use "correct horse battery staple" as an encryption key, including a recommended 4096-word list (p. 2522) 2014-04-24 (44 minutes)
• Comparison of the PCO-1810 and PCO-1881 plastic bottlecap standards (p. 3223) 2014-05-25 (updated 2016-07-27) (2 minutes)
• Waterproofing (p. 429) 2015-09-03 (4 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Likely-feasible non-flux-deposition powder-bed 3-D printing processes (p. 1196) 2015-09-11 (updated 2019-12-20) (49 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Electrodeposition 3d printing (p. 1115) 2016-02-19 (4 minutes)
• Material merits (p. 1496) 2016-05-08 (6 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Wikipedia people (p. 948) 2016-06-01 (6 minutes)
• Transmission line computer (p. 509) 2016-07-11 (updated

2019-07-23) (7 minutes)
• How can we build an efficient microcontroller-based amplifier? (p. 2821) 2016-07-13 (5 minutes)
• Jellybean ICs 2016 (p. 817) 2016-07-14 (updated 2019-05-05) (17 minutes)
• Transmission line diode computation (p. 3090) 2016-07-30 (3 minutes)
• Calculations about desalination in Israel (p. 2827) 2016-08-11 (3 minutes)
• Argentine oscilloscope pricing 2016 (p. 1965) 2016-08-16 (4 minutes)
• Hot oil cutter (p. 3287) 2016-08-16 (updated 2016-08-17) (8 minutes)
• Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) 2016-08-17 (updated 2016-08-30) (24 minutes)
• Pulley generator (p. 3148) 2016-09-05 (2 minutes)
• Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
• The internet is probably not going to collapse for economic reasons (p. 3194) 2016-09-06 (9 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• License-free femtowatt UHF radio transceiver ICs under a μJ per bit (p. 162) 2016-09-19 (5 minutes)
• Usability of scientific calculators (p. 2379) 2016-09-29 (19 minutes)
• My attempt to learn about jellybean electronic components (p. 1974) 2017-02-08 (updated 2019-09-29) (22 minutes)
• Lab power supply (p. 2421) 2017-02-21 (updated 2018-06-18) (17 minutes)
• 3-D printing by flux deposition (p. 466) 2017-02-24 (updated 2019-07-27) (21 minutes)
• Vibratory powder delivery (p. 1747) 2017-02-25 (2 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
• Burst computation (p. 1500) 2017-03-20 (13 minutes)
• The continuous-web press and the continuous press of the World-Wide Web (p. 1134) 2017-03-20 (6 minutes)
• Minimum hardware and software to get a flexible notetaking device running (p. 535) 2017-04-28 (4 minutes)
• Illumination cost (p. 1242) 2017-05-31 (3 minutes)
• How cheap can laser-cut boxes be? (p. 2545) 2017-06-01 (2 minutes)
• A minimal-cost diet with adequate nutrition in Argentina in 2017 is US$0.67 per day (p. 3206) 2017-06-15 (4 minutes)
• CCD oscilloscope (p. 1861) 2017-06-20 (updated 2017-07-04) (7 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)
• Dyneema (p. 123) 2017-07-19 (2 minutes)
• Pipe dome (p. 3068) 2017-07-19 (7 minutes)

• Solar computer 2 (p. 414) 2017-07-19 (3 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Copper plating furniture (p. 1460) 2017-07-19 (updated 2017-09-01) (4 minutes)
• Approaches to 3-D printing in sandstone (p. 1095) 2017-08-03 (5 minutes)
• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)
• Bench trash power supply (p. 1457) 2018-04-27 (9 minutes)
• Earring computer (p. 2505) 2018-04-27 (1 minute)
• Urban autarkic network (p. 1026) 2018-04-27 (1 minute)
• Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642) 2018-04-30 (updated 2018-07-05) (29 minutes)
• Resistor assortment (p. 310) 2018-06-17 (4 minutes)
• Transistors vs. Microcontrollers (p. 2918) 2018-06-17 (updated 2018-07-05) (8 minutes)
• Electric hammer (p. 1865) 2018-07-02 (updated 2018-07-05) (14 minutes)
• Capacitors: some notes on tradeoffs (p. 134) 2018-07-05 (5 minutes)
• Microlens vibrating lightfield (p. 1219) 2018-07-14 (updated 2018-07-15) (11 minutes)
• Notes on circuitry for the Nutra seed activator (p. 3099) 2018-08-16 (20 minutes)
• The details of the GPU in this laptop (p. 2970) 2018-10-29 (2 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)
• Balcony battery (p. 2377) 2019-02-11 (updated 2019-12-06) (6 minutes)
• Fractal palettes (p. 202) 2019-04-02 (7 minutes)
• Notes on SIP VoIP in 2019 (p. 1064) 2019-06-07 (updated 2019-06-28) (8 minutes)
• How to get 6 volts out of a 7805, and why you shouldn't (p. 537) 2019-06-08 (updated 2019-06-10) (8 minutes)
• Energy storage efficiency (p. 1300) 2019-07-30 (4 minutes)
• the oversold-as-low-power Renesas RL78 microcontroller line (p. 504) 2019-08-27 (10 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)
• Can artificially-lit vertical farming compete with greenhouses? (p. 2064) 2019-09-08 (12 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Extending Heckballs (p. 3239) 2019-11-26 (6 minutes)
• Argentine electric bill (p. 2898) 2019-12-18 (3 minutes)
• Sulfuric acid dehydration printing (p. 174) 2019-12-18 (updated 2019-12-19) (3 minutes)

Notes concerning “Printing”

• Full res globe (p. 1255) 2014-02-24 (1 minute)
• Holographic archival (p. 766) 2014-04-24 (10 minutes)
• Archival transparencies (p. 1345) 2014-06-05 (updated 2014-06-29) (7 minutes)
• Quadratic opacity holograms (p. 3073) 2015-09-03 (7 minutes)
• Wikipedia people (p. 948) 2016-06-01 (6 minutes)
• Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)
• Some extensions of William Beaty’s scratch holograms (p. 2536) 2019-07-11 (9 minutes)

Notes concerning “Privacy”

• In a world with ubiquitous surveillance, what does politics look like? (p. 1615) 2014-04-24 (11 minutes)
• Hearing aids for disability compensation, protection, and augmentation (p. 764) 2019-09-08 (updated 2019-09-09) (4 minutes)

Notes concerning “Probabilistic programming”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Bayesian and Gricean programming (p. 711) 2015-08-20 (3 minutes)

Notes concerning “Probability”

• Bayesian and Gricean programming (p. 711) 2015-08-20 (3 minutes)

• Interval filters (p. 1282) 2015-09-17 (2 minutes)
• Texture synthesis with spatial-domain particle filters (p. 857) 2016-10-06 (2 minutes)
• One-line thoughts that don’t merit separate notes (p. 80) 2017-01-04 (updated 2017-02-25) (4 minutes)
• Better be weird (p. 1831) 2019-06-17 (updated 2019-06-24) (9 minutes)

Notes concerning “Process intensification”

• Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
• Freeze distillation at 1 Hz (p. 2796) 2016-10-06 (5 minutes)
• Recuperator heat storage (p. 594) 2016-11-01 (updated 2019-08-21) (4 minutes)
• Clay fabrication objectives (p. 2111) 2017-01-16 (updated 2017-01-17) (3 minutes)
• Fast sea salt evaporator (p. 1087) 2017-06-01 (3 minutes)
• Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)

Notes concerning “Program design”

• I think I understand how to use libart’s antialiased rendering API now (p. 409) 2007 to 2009 (10 minutes)
• Nested inheritance (p. 340) 2007 to 2009 (2 minutes)
• When and why exactly should your code “tell, not ask”? That is, use CPS? (p. 1051) 2014-01-08 (4 minutes)
• Precisely how is 3 “optimal” for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• Simplifying code with concurrent iteration (p. 3293) 2014-04-24 (2 minutes)
• Logarithmic maintainability and coupling (p. 2341) 2015-11-23 (7 minutes)
• Byte-stream pipe and antipipe façade objects for editor buffers (p. 950) 2017-04-10 (3 minutes)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Immediate mode productive grammars (p. 898) 2018-09-13 (8 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)

Notes concerning “Programming by example”

• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Interactive calculator (p. 2771) 2018-04-26 (16 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)

Notes concerning “Programming languages”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Bicicleta maps (p. 582) 2007 to 2009 (2 minutes)
• Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• Erlang musings (p. 2789) 2007 to 2009 (3 minutes)
• Error Reporting is Part of the Programmer's User Interface (p. 2323) 2007 to 2009 (18 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• Designing a Scheme for APL-like array computations, like Lush (p. 661) 2007 to 2009 (4 minutes)
• Nested inheritance (p. 340) 2007 to 2009 (2 minutes)
• Notes on Raph Levien's "Io" Programming Language (p. 1740) 2007 to 2009 (10 minutes)
• The AL programming language, dimensional analysis, and typing: do different dimensions really exist? (p. 731) 2007 to 2009 (2 minutes)

• ML’s value restriction and the Modula-3 typing system (p. 1763) 2007 to 2009 (3 minutes)
• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• Simple system language (p. 1659) 2013-05-17 (7 minutes)
• Forth with named stacks (p. 2101) 2014-02-24 (7 minutes)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• A variety of code fragments for testing proposed language designs (p. 2560) 2016-05-18 (19 minutes)
• Algorithm time capsule (p. 2263) 2016-08-11 (1 minute)
• Toward a language for hacking around with natural-language processing algorithms (p. 1681) 2016-09-08 (7 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p.

1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Relational modeling (p. 1102) 2017-05-17 (updated 2017-06-01) (6 minutes)
• Parametric polymorphism and columns (p. 1835) 2017-07-19 (2 minutes)
• Term rewriting (p. 3221) 2017-07-19 (3 minutes)
• Some notes on FullPliant and Pliant (p. 866) 2018-04-27 (9 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Leconscrip: a family of JS subsets for BubbleOS (p. 2126) 2018-11-23 (2 minutes)
• How small can we make a comfortable subset of JS? (p. 1348) 2018-11-27 (updated 2018-12-02) (3 minutes)
• Minimal imperative language (p. 2175) 2018-12-10 (7 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• Improving Lua #L with incremental prefix sum in the ∧ monoid (p. 2008) 2018-12-18 (7 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)
• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)
• A review of Wirth's Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• An algebra of textures for interactive composition (p. 1283) 2019-05-08 (4 minutes)
• A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)
• Observable transaction possibilities (p. 2086) 2019-06-15 (10 minutes)
• 10tcl ui (p. 1823) 2019-12-06 (17 minutes)
• Introduction to closures (p. 1403) 2019-12-07 (5 minutes)

Notes concerning “Programming”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• A 2007 overview of matrix barcodes (p. 1094) 2007 to 2009 (2 minutes)
• Bicicleta maps (p. 582) 2007 to 2009 (2 minutes)
• Using bytecode won't make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• C bad (p. 2832) 2007 to 2009 (4 minutes)
• The coolest bug in Ur-Scheme (p. 1007) 2007 to 2009 (2 minutes)
• A stack of coordinate contexts (p. 2987) 2007 to 2009 (9 minutes)
• A cute algorithm for card-image templates (p. 1946) 2007 to 2009 (2 minutes)
• Notes on reading eForth (p. 1398) 2007 to 2009 (9 minutes)
• Enumerating binary trees and their elements (p. 1445) 2007 to 2009 (4 minutes)
• Erlang musings (p. 2789) 2007 to 2009 (3 minutes)
• Error Reporting is Part of the Programmer's User Interface (p. 2323) 2007 to 2009 (18 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• Free software debugging (p. 136) 2007 to 2009 (2 minutes)
• IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
• Gaim group chat (p. 2677) 2007 to 2009 (3 minutes)
• Interesting features of the GNU assembler Gas (p. 3000) 2007 to 2009 (2 minutes)
• Learning low level stuff is not just fun, but also useful (p. 815) 2007 to 2009 (5 minutes)
• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• Designing a Scheme for APL-like array computations, like Lush (p. 661) 2007 to 2009 (4 minutes)
• Developing Win32 programs on Debian (p. 609) 2007 to 2009 (12 minutes)
• Nested inheritance (p. 340) 2007 to 2009 (2 minutes)
• Does SAAS make it harder to ship? I doubt it. (p. 1049) 2007 to 2009 (7 minutes)
• Tagged dataflow (p. 405) 2007 to 2009 (2 minutes)
• The Problem: Writing With One Access Pattern, Reading With Another (p. 3004) 2007 to 2009 (19 minutes)
• Programming paradigms for tiny microcontrollers (p. 2104) 2007 to 2009 (6 minutes)
• The AL programming language, dimensional analysis, and typing: do different dimensions really exist? (p. 731) 2007 to 2009 (2 minutes)

• ML's value restriction and the Modula-3 typing system (p. 1763) 2007 to 2009 (3 minutes)
• What's wrong with ../../? (p. 2304) 2007 to 2009 (2 minutes)
• Win32 startup (p. 2439) 2007 to 2009 (2 minutes)
• Index set inference or domain inference for programming with

indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• In what sense is e the optimal branching factor, and what does it mean for menu tree design? (p. 2483) 2012-12-04 (3 minutes)
• Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
• a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
• How can we take advantage of 16:9 screens for programming? (p. 1982) 2012-12-17 (2 minutes)
• Giving Golang a second look for writing a mailreader (in 2012) (p. 290) 2012-12-17 (updated 2013-05-17) (2 minutes)
• Clickable terminal patterns (p. 859) 2013-05-17 (2 minutes)
• Use crit-bit trees as the fundamental string-set data structure (p. 1498) 2013-05-17 (3 minutes)
• Cycle sort (p. 2344) 2013-05-17 (1 minute)
• How can we usefully cache screen images for incrementalization? (p. 1077) 2013-05-17 (18 minutes)
• You're pretty much fucked if you want to build an oscilloscope on the AVR's ADC (p. 1269) 2013-05-17 (3 minutes)
• Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
• The delta from QEmacs,with only 88 commands, to a usable Emacs, is small (p. 1543) 2013-05-17 (2 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Simple system language (p. 1659) 2013-05-17 (7 minutes)
• APL with typed indices (p. 3264) 2013-05-17 (11 minutes)
• Optimizing the Visitor pattern on the DOM using Quaject-style dynamic code generation (p. 1508) 2013-05-17 (updated 2013-05-20) (21 minutes)
• Distinguishing natural languages with 3-grams of characters (p. 2953) 2013-05-17 (updated 2013-05-20) (7 minutes)
• When and why exactly should your code "tell, not ask"? That is, use CPS? (p. 1051) 2014-01-08 (4 minutes)
• Constant-space grep (p. 296) 2014-02-24 (3 minutes)
• Forth with named stacks (p. 2101) 2014-02-24 (7 minutes)
• Embedding objects inside other objects in memory, versus by-reference fields (p. 3112) 2014-02-24 (13 minutes)
• Simple persistent in-memory dictionaries with $\log^2$ lookups and logarithmic insertion (p. 3110) 2014-02-24 (6 minutes)
• Square wave synthesis (p. 3200) 2014-02-24 (2 minutes)
• Twingler (p. 2547) 2014-02-24 (7 minutes)
• Compression with second-order diffs (p. 2152) 2014-04-24 (3 minutes)
• Fixed point (p. 807) 2014-04-24 (1 minute)
• Precisely how is 3 "optimal" for one-hot state machines, sparse FIR kernels, etc.? (p. 450) 2014-04-24 (8 minutes)
• Ostinatto (p. 2780) 2014-04-24 (4 minutes)
• Range literals (p. 1719) 2014-04-24 (6 minutes)
• Simplifying code with concurrent iteration (p. 3293) 2014-04-24

(2 minutes)
• Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
• The Dontmove archival virtual machine (p. 2113) 2014-06-29 (5 minutes)
• How to generate unique IDs for IMGUI object persistence? (p. 2035) 2014-09-02 (3 minutes)
• A reactive crawler using Amygdala (p. 2492) 2014-09-02 (updated 2014-09-19) (4 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• Ndarray java (p. 2261) 2015-05-28 (1 minute)
• Editor buffers (p. 1328) 2015-07-15 (updated 2015-09-03) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Cobstrings (p. 1312) 2015-08-21 (updated 2015-08-31) (5 minutes)
• Raggedcolumns (p. 2703) 2015-08-28 (3 minutes)
• Implementing flatMap in terms of call/cc, as in Raph Levien's Io (p. 3248) 2015-09-03 (3 minutes)
• Parsing a conservative approximation of a CFG with a FSM (p. 159) 2015-09-03 (7 minutes)
• A simple content-addressable storage-server protocol (p. 774) 2015-09-03 (3 minutes)
• Desbarrerarme: a UI for speaking to people (p. 186) 2015-09-03 (5 minutes)
• An IMGUI-style drawing API isn't necessarily just immediate-mode graphics (p. 2671) 2015-09-03 (3 minutes)
• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Storing CSV records in minimal memory in Java (p. 524) 2015-09-03 (6 minutes)
• Assigning consistent order IDs (p. 1042) 2015-09-03 (3 minutes)
• Rhythm codes (p. 2375) 2015-09-03 (4 minutes)
• Would Synthgramelodia make a good base for livecoding music? (p. 2540) 2015-09-03 (8 minutes)
• Ternary mergesort (p. 2161) 2015-09-03 (2 minutes)
• Convolution surface plotting (p. 2264) 2015-09-03 (updated 2015-09-13) (2 minutes)
• Parallel NFA evaluation (p. 2967) 2015-09-03 (updated 2015-10-01) (8 minutes)
• Convolution applications (p. 2930) 2015-09-07 (updated 2019-08-14) (9 minutes)
• Hash feature detection (p. 3294) 2015-09-17 (5 minutes)
• Interactive calculator 0 (p. 1453) 2015-09-17 (2 minutes)
• Interval filters (p. 1282) 2015-09-17 (2 minutes)
• Fast geographical maps on Android (p. 455) 2015-10-16 (9 minutes)
• Virtual instruments (p. 658) 2015-11-09 (3 minutes)
• Hash gossip exchange (p. 1470) 2015-11-19 (4 minutes)
• Logarithmic maintainability and coupling (p. 2341) 2015-11-23 (7 minutes)
• Improving LZ77 compression with a RET bytecode (p. 964) 2016-04-05 (updated 2016-04-06) (3 minutes)

• Anytime realtime (p. 803) 2016-04-22 (4 minutes)
• A type-inferred dialect of JS (p. 265) 2016-04-22 (4 minutes)
• Trees as code (p. 2488) 2016-05-10 (4 minutes)
• A variety of code fragments for testing proposed language designs (p. 2560) 2016-05-18 (19 minutes)
• Statically bounding runtime (p. 2398) 2016-07-19 (4 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)
• Improving lossless image compression with basic machine learning algorithms (p. 2546) 2016-07-27 (2 minutes)
• Append only unique string pool (p. 1797) 2016-07-27 (2 minutes)
• Algorithm time capsule (p. 2263) 2016-08-11 (1 minute)
• Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
• Affine arithmetic has quadratic convergence when interval arithmetic has linear convergence (p. 1029) 2016-08-24 (updated 2017-01-18) (10 minutes)
• Time series data type (p. 304) 2016-08-26 (3 minutes)
• Low-cost green thread locks (p. 252) 2016-09-06 (2 minutes)
• Notes on higher-order programming on the JVM (p. 1355) 2016-09-06 (6 minutes)
• DHT bulletin board (p. 3001) 2016-09-07 (7 minutes)
• An almost-in-place mergesort (p. 740) 2016-09-07 (5 minutes)
• Toward a language for hacking around with natural-language processing algorithms (p. 1681) 2016-09-08 (7 minutes)
• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)
• DReX and "regular string transformations": would an RPN DSL work well? (p. 453) 2016-09-19 (3 minutes)
• Gradient rendering (p. 583) 2016-09-24 (11 minutes)
• Changing the basis to a more expressive one with better affordances (p. 1389) 2016-09-29 (5 minutes)
• Usability of scientific calculators (p. 2379) 2016-09-29 (19 minutes)
• Notations for defining dynamical systems (p. 2872) 2016-10-03 (updated 2016-10-06) (6 minutes)
• Texture synthesis with spatial-domain particle filters (p. 857) 2016-10-06 (2 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• What's the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• Generalizing my RPN calculator to support refactoring (p. 969) 2016-10-17 (12 minutes)
• Chintzy depth of field (p. 629) 2016-10-27 (1 minute)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• MiniOS (p. 1091) 2016-12-28 (updated 2017-01-03) (6 minutes)
• The paradoxical complexity of computing the top N (p. 1890) 2017-01-04 (7 minutes)
• What is the type of lerp? (p. 1985) 2017-01-08 (5 minutes)
• Similarities between Golang and Rust (p. 1523) 2017-01-11 (updated 2017-01-17) (7 minutes)
• Constant time sets for pixel painting (p. 1484) 2017-02-07 (2 minutes)

• Text editor slow keys (p. 808) 2017-02-07 (2 minutes)
• A 7-segment-display font with 68 glyphs (p. 1798) 2017-02-21 (4 minutes)
• Set hashing (p. 2485) 2017-03-09 (9 minutes)
• Cartesian product storage (p. 3012) 2017-03-20 (3 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Amnesic hash tables for stochastically LRU memoization (p. 502) 2017-04-03 (1 minute)
• The history of NoSQL and dbm (p. 45) 2017-04-10 (16 minutes)
• Byte-stream pipe and antipipe façade objects for editor buffers (p. 950) 2017-04-10 (3 minutes)
• Incremental persistent binary array sets (p. 1008) 2017-04-10 (4 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Secure, self-describing, self-delimiting serialization for Python (p. 243) 2017-04-11 (8 minutes)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)
• String tuple encoding (p. 2419) 2017-04-28 (2 minutes)
• Hipster stack 2017 (p. 2242) 2017-04-28 (updated 2017-05-04) (26 minutes)
• Generic programming with proofs, specification, refinement, and specialization (p. 958) 2017-05-10 (6 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)
• Relational modeling (p. 1102) 2017-05-17 (updated 2017-06-01) (6 minutes)
• Karplus–Strong PLLs (p. 2100) 2017-06-09 (1 minute)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• Database explorer (p. 225) 2017-06-20 (2 minutes)
• Compressing a screen update with a tree of dirty bits (p. 303) 2017-06-21 (1 minute)
• A REST interface to a software transactional memory (p. 3014) 2017-06-21 (2 minutes)
• CIC-filter fonts (p. 1229) 2017-06-28 (1 minute)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• Blob computation (p. 2214) 2017-07-19 (2 minutes)
• Double heap sequence (p. 2521) 2017-07-19 (2 minutes)
• Ideal language syntax (p. 3260) 2017-07-19 (1 minute)
• Parametric polymorphism and columns (p. 1835) 2017-07-19 (2 minutes)
• Rasterizing polies (p. 2023) 2017-07-19 (3 minutes)
• Options for bootstrapping a compiler from a tiny compiler using Brainfuck (p. 2958) 2017-07-19 (2 minutes)
• Term rewriting (p. 3221) 2017-07-19 (3 minutes)
• JIT-compiling array computation graphs in JS (p. 155) 2017-07-19 (1 minute)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)

• The Z-machine memory model (p. 2903) 2017-07-19 (4 minutes)
• Xor 1 to 1 hashing (p. 1595) 2017-07-19 (updated 2017-08-03) (10 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• Minimal transaction system (p. 2460) 2017-09-21 (5 minutes)
• Querying a pile of free-text strings with quasi-Prolog (p. 811) 2017-11-17 (6 minutes)
• Interactive geometry (p. 508) 2018-04-26 (1 minute)
• Two-thumb quasimodal multitouch interaction techniques (p. 1765) 2018-04-26 (11 minutes)
• Constant space flexible data (p. 352) 2018-04-27 (5 minutes)
• Rarely are function-local variables in Forth justified (p. 1055) 2018-04-27 (10 minutes)
• Framed-belt DSP (p. 3095) 2018-04-27 (3 minutes)
• Incremental recomputation (p. 1184) 2018-04-27 (12 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Literate programs should include example output, like Jupyter, but Jupyter is imperfect (p. 1308) 2018-04-27 (3 minutes)
• Some notes on FullPliant and Pliant (p. 866) 2018-04-27 (9 minutes)
• How inefficient is SNAT hole-punching via random port trials? (p. 1155) 2018-04-27 (2 minutes)
• Mail reader (p. 3290) 2018-04-27 (updated 2018-06-18) (7 minutes)

• Composing code gobbets with implicit dependencies (p. 2437) 2018-04-27 (updated 2019-05-21) (3 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Accelerating convolution and correlation with short periodic waveforms using OLAP marginal prefix sums (p. 895) 2018-06-05 (4 minutes)
• Clisweep (p. 2705) 2018-06-06 (3 minutes)
• Toward a minimal PEG parsing engine (p. 955) 2018-06-06 (4 minutes)
• Multitouch livecoding (p. 122) 2018-06-17 (1 minute)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Flexible text query (p. 2900) 2018-07-14 (4 minutes)
• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)
• Top algorithms (p. 913) 2018-07-29 (4 minutes)
• Caustic simulation (p. 1454) 2018-09-10 (updated 2018-11-04) (2 minutes)
• Immediate mode productive grammars (p. 898) 2018-09-13 (8 minutes)
• Golang bugs (p. 3087) 2018-09-13 (updated 2018-10-28) (6 minutes)
• Bit difference array (p. 1748) 2018-10-28 (10 minutes)

• Digital noise generators (p. 1137) 2018-10-28 (2 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Notes on the Intel N3700 i915 GPU in this ASUS E403S laptop (p. 2033) 2018-10-28 (updated 2019-05-05) (3 minutes)
• Dilating letterforms (p. 651) 2018-11-04 (15 minutes)
• Performance properties of sets of bitwise operations (p. 636) 2018-11-06 (updated 2018-11-07) (16 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)
• Leconscrip: a family of JS subsets for BubbleOS (p. 2126) 2018-11-23 (2 minutes)
• Tagging parsers (p. 208) 2018-11-23 (updated 2018-12-10) (9 minutes)
• Fast gsave (p. 1593) 2018-11-27 (5 minutes)
• What would a better Unix shell look like? (p. 2831) 2018-11-27 (1 minute)
• How small can we make a comfortable subset of JS? (p. 1348) 2018-11-27 (updated 2018-12-02) (3 minutes)
• Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)
• Sparse filters (p. 834) 2018-12-02 (4 minutes)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)
• Constant space lists (p. 3062) 2018-12-10 (10 minutes)
• Minimal imperative language (p. 2175) 2018-12-10 (7 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)
• Evaluating DSP operations in minimal buffer space by pipelining (p. 321) 2018-12-18 (updated 2018-12-19) (20 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• Commentaries on reading Engelbart's "Augmenting Human Intellect" (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)
• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• Supervisor children for fault-tolerant Unix command-line programs (p. 3224) 2019-01-04 (3 minutes)
• Some notes on morphology, including improvements on Urbach and Wilkinson's erosion/dilation algorithm (p. 216) 2019-01-04 (updated 2019-11-12) (26 minutes)
• Median filtering (p. 3155) 2019-01-17 (11 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)
• A review of Wirth's Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• My notes from learning the Golang standard library (p. 2739)

2019-02-08 (20 minutes)
• India rubber memory (p. 579) 2019-03-19 (4 minutes)
• Fractal palettes (p. 202) 2019-04-02 (7 minutes)
• Audio video boustrophedon sync (p. 858) 2019-04-03 (2 minutes)
• An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)
• Dercuano calculation (p. 3135) 2019-05-01 (3 minutes)
• An algebra of textures for interactive composition (p. 1283) 2019-05-08 (4 minutes)
• A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)
• First impressions on using the µMath+ calculator program for Android (p. 195) 2019-05-21 (13 minutes)
• Profile-guided parser optimization should enable parsing of gigabytes per second (p. 2283) 2019-05-23 (8 minutes)
• Microsoft Windows uses \ for filenames because OS/8 programs used / for switches (p. 3098) 2019-05-25 (2 minutes)
• Midpoint method texture mapping (p. 1837) 2019-06-01 (3 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Kernel code generation (p. 2302) 2019-07-02 (6 minutes)
• Prolog table outlining (p. 2837) 2019-07-05 (11 minutes)
• Assembler bootstrapping (p. 2922) 2019-07-18 (updated 2019-12-08) (16 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)
• Using the method of secants for general optimization (p. 1773) 2019-07-22 (updated 2019-11-26) (13 minutes)
• $1 recognizer diagrams (p. 1264) 2019-08-11 (updated 2019-10-24) (15 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Complex linear regression (in the field ℂ of complex numbers) (p. 3018) 2019-08-17 (updated 2019-08-18) (9 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)
• Some notes on the landscape of linear optimization software and applications (p. 1285) 2019-08-21 (updated 2019-08-25) (35 minutes)
• Text relational query (p. 1223) 2019-08-28 (10 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)
• Query evaluation with interval-annotated trees over sequences (p. 1423) 2019-08-30 (updated 2019-09-03) (30 minutes)
• Differentiable neighborhood regression (p. 2944) 2019-08-31 (15 minutes)

• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• Dercuano plotting (p. 2885) 2019-09-03 (updated 2019-09-05) (34 minutes)
• A formal language for defining implicitly parameterized functions (p. 144) 2019-09-05 (updated 2019-09-30) (29 minutes)
• Isotropic nonlinear texture effects for letterforms from a scale-space representation (p. 1609) 2019-09-10 (16 minutes)
• Nonlinear bounded leaky integrator (p. 3150) 2019-09-11 (8 minutes)
• Fast mathematical optimization with affine arithmetic (p. 3163) 2019-09-15 (5 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)
• Sparse sinc (p. 1880) 2019-09-15 (10 minutes)
• Notes on local file browsing (p. 484) 2019-09-15 (updated 2019-09-28) (4 minutes)
• B-Tree ropes (p. 2762) 2019-09-24 (updated 2019-09-25) (19 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)
• Dercuano grinding (p. 2475) 2019-10-01 (12 minutes)
• Is there an incremental union find algorithm? (p. 1602) 2019-10-01 (8 minutes)
• Notes on Óscar Toledo G.'s bootOS (p. 277) 2019-10-07 (updated 2019-10-08) (28 minutes)
• Resurrecting duckling hashing (p. 3128) 2019-10-26 (updated 2019-11-10) (8 minutes)
• Negative weight undirected graphs (p. 1590) 2019-11-01 (8 minutes)

• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)
• Interval raymarching (p. 1342) 2019-11-02 (updated 2019-11-10) (6 minutes)
• Some thoughts on SDF raymarching (p. 312) 2019-11-11 (updated 2019-12-10) (31 minutes)
• Rediscovering successive parabolic interpolation: derivative-free optimization of scalar functions by fitting a parabola (p. 727) 2019-11-26 (updated 2019-11-27) (8 minutes)
• Byte stream gui applications (p. 128) 2019-11-29 (updated 2019-11-30) (17 minutes)
• Memory safe virtual machines (p. 975) 2019-12-04 (14 minutes)
• 10tcl ui (p. 1823) 2019-12-06 (17 minutes)
• Introduction to closures (p. 1403) 2019-12-07 (5 minutes)
• Forth assembling (p. 940) 2019-12-08 (updated 2019-12-11) (18 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)
• My very first toddling steps in ARM assembly language (p. 1684) 2019-12-10 (updated 2019-12-13) (46 minutes)
• Can you eliminate backpatching? (p. 1769) 2019-12-17 (8 minutes)
• Hypothesis evolution (p. 2143) 2019-12-17 (4 minutes)
• Sorting in logic (p. 498) 2019-12-28 (2 minutes)

Notes concerning “Prolog and logic programming”

• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• MiniOS (p. 1091) 2016-12-28 (updated 2017-01-03) (6 minutes)
• Querying a pile of free-text strings with quasi-Prolog (p. 811) 2017-11-17 (6 minutes)
• Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)
• Relational modeling and APL (p. 1217) 2019-05-20 (updated 2019-05-21) (5 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• Prolog table outlining (p. 2837) 2019-07-05 (11 minutes)
• Sorting in logic (p. 498) 2019-12-28 (2 minutes)

Notes concerning “Protocols”

• What’s wrong with ../../? (p. 2304) 2007 to 2009 (2 minutes)
• Stuff I’ve posted to kragen-tol over the years about post-HTTP (p. 1815) 2014-02-24 (12 minutes)
• A simple content-addressable storage-server protocol (p. 774) 2015-09-03 (3 minutes)
• Hash gossip exchange (p. 1470) 2015-11-19 (4 minutes)
• DHT bulletin board (p. 3001) 2016-09-07 (7 minutes)
• Hipster stack 2017 (p. 2242) 2017-04-28 (updated 2017-05-04) (26 minutes)
• A plotter language of 9-bit bytes (p. 2154) 2017-05-29 (updated 2017-06-01) (11 minutes)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• What’s wrong with CoAP (p. 560) 2017-06-15 (3 minutes)
• Pixel stream (p. 617) 2017-06-15 (updated 2018-10-26) (4 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Service-oriented email (p. 1302) 2017-06-20 (updated 2017-06-21) (15 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• Compressing REST transactions with per-connection state (p. 1117) 2018-04-27 (11 minutes)
• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Notes on SIP VoIP in 2019 (p. 1064) 2019-06-07 (updated 2019-06-28) (8 minutes)
• Resurrecting duckling hashing (p. 3128) 2019-10-26 (updated 2019-11-10) (8 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)
• Byte stream gui applications (p. 128) 2019-11-29 (updated 2019-11-30) (17 minutes)

Notes concerning “Psychology”

• On hanging out with cranks (p. 1715) 2008-04 (4 minutes)
• Ostinatto (p. 2780) 2014-04-24 (4 minutes)
• How to use “correct horse battery staple” as an encryption key, including a recommended 4096-word list (p. 2522) 2014-04-24 (44 minutes)
• Ideas to ship in 2014 (p. 1409) 2014-04-24 (updated 2019-05-05) (35 minutes)
• He listened to the human intently (p. 2543) 2014-06-29 (4 minutes)

• Buenos Aires seen from behind the mirror (p. 809) 2014-09-02 (7 minutes)
• ¿Qué necesito para relación de pareja? (p. 1298) 2016-03-09 (6 minutes)
• Do visually expanding images evoke an orienting response, or the startle response, and what does that mean for ZUIs? (p. 1805) 2016-06-03 (14 minutes)
• Selfish conformity (p. 622) 2016-11-15 (5 minutes)
• The ultimate capacity of human memory if spaced-practice memorization works as advertised (p. 2442) 2017-01-04 (updated 2017-01-08) (14 minutes)
• A tournament to decide which notes to devote attention to polishing (p. 1195) 2017-07-19 (2 minutes)
• Frustration (p. 3263) 2018-04-27 (2 minutes)
• Life octaves (p. 173) 2018-10-28 (2 minutes)
• An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)
• A note on meditation (p. 494) 2019-04-20 (1 minute)
• Categorical zero sum prohibition (p. 2553) 2019-05-27 (updated 2019-06-01) (23 minutes)
• Human memorable secret sharing (p. 426) 2019-08-10 (2 minutes)
• Autism is overfitting (p. 2692) 2019-08-31 (1 minute)

Notes concerning “Pubsub”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Micro pubsub (p. 1504) 2017-06-15 (8 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• Fast secure pubsub (p. 545) 2019-02-04 (updated 2019-12-03) (2 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)
• Bytecode pubsub (p. 205) 2019-12-04 (6 minutes)

Notes concerning “Python”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Using bytecode won't make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• A cute algorithm for card-image templates (p. 1946) 2007 to 2009 (2 minutes)
• Enumerating binary trees and their elements (p. 1445) 2007 to 2009 (4 minutes)
• Error Reporting is Part of the Programmer's User Interface (p. 2323) 2007 to 2009 (18 minutes)
• Index set inference or domain inference for programming with indexed families (p. 1434) 2007 to 2009 (updated 2019-05-05) (27 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Range literals (p. 1719) 2014-04-24 (6 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Toward a language for hacking around with natural-language processing algorithms (p. 1681) 2016-09-08 (7 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)
• Rarely are function-local variables in Forth justified (p. 1055) 2018-04-27 (10 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Yeso notes (p. 2585) 2018-12-25 (updated 2019-01-01) (11 minutes)
• Observable transaction possibilities (p. 2086) 2019-06-15 (10 minutes)
• Reducing the cost of self-verifying arithmetic with array operations (p. 2205) 2019-06-23 (15 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Robust local search in vector spaces using adaptive step sizes, and

Notes concerning “Qemu”

• Notes on running QEMU on Debian Etch (p. 1646) 2007 to 2009 (3 minutes)
• Running your regular desktop in QEMU? (p. 292) 2007 to 2009 (3 minutes)

Notes concerning “Quasimodal”

• Interactive geometry (p. 508) 2018-04-26 (1 minute)
• Two-thumb quasimodal multitouch interaction techniques (p. 1765) 2018-04-26 (11 minutes)

Notes concerning “Quasimodes”

- Interactive calculator (p. 2771) 2018-04-26 (16 minutes)
- Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)

Notes concerning “Radio”

• Arduino radio (p. 169) 2016-07-30 (4 minutes)
• Could you do DDS of comprehensible radio signals with an Arduino? (p. 1829) 2017-03-31 (4 minutes)
• Dumb vocoder (p. 1391) 2017-05-10 (2 minutes)
• Urban autarkic network (p. 1026) 2018-04-27 (1 minute)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Can you bitbang wireless communication between AVRs? How about AM-radio energy harvesting? (p. 2509) 2019-08-27 (updated 2019-08-28) (37 minutes)
• Examination of a shitty USB car charger (p. 286) 2019-10-24 (13 minutes)
• Transmitting low-power TV signals around your house via RF modulation with an SDR (p. 1950) 2019-12-01 (6 minutes)

Notes concerning “Raytracing”

- Current hardware trends tend toward raytracing (p. 1351) 2016-10-07 (4 minutes)
- Reduced affine arithmetic raytracer (p. 2007) 2017-05-10 (1 minute)

Notes concerning “Refractories”

• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)

Notes concerning “Regenerators”

• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• Regenerative fuel air cutting (p. 2622) 2016-09-06 (4 minutes)
• Freeze distillation at 1 Hz (p. 2796) 2016-10-06 (5 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)

Notes concerning “Regexps”

- Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
- Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)

Notes concerning “Relays”

• Paper/foil relays (p. 3273) 2019-04-02 (updated 2019-10-23) (13 minutes)
• Bistable magnetic electromechanical display (p. 1016) 2019-10-24 (16 minutes)
• Nonconductive relays (p. 3262) 2019-11-12 (3 minutes)

Notes concerning "Reproducibility"

• The book written in itself (p. 2400) 2016-06-12 (updated 2016-06-14) (18 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)

Notes concerning “Research”

• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• Academic lineage (p. 2292) 2016-10-30 (updated 2019-11-24) (15 minutes)
• Commentaries on reading Engelbart’s “Augmenting Human Intellect” (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• A review of Wirth’s Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• Some musings on applying Fitts’s Law to user interface design and data compression (p. 1164) 2019-05-06 (updated 2019-05-09) (27 minutes)

Notes concerning “REpresentational State Transfer”

• Stuff I’ve posted to kragen-tol over the years about post-HTTP (p. 1815) 2014-02-24 (12 minutes)
• A simple content-addressable storage-server protocol (p. 774) 2015-09-03 (3 minutes)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• Micro pubsub (p. 1504) 2017-06-15 (8 minutes)
• Service-oriented email (p. 1302) 2017-06-20 (updated 2017-06-21) (15 minutes)
• A REST interface to a software transactional memory (p. 3014) 2017-06-21 (2 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• Compressing REST transactions with per-connection state (p. 1117) 2018-04-27 (11 minutes)

Notes concerning "Retrocomputing"

• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Alien game challenge (p. 2313) 2015-09-03 (6 minutes)
• Changing the basis to a more expressive one with better affordances (p. 1389) 2016-09-29 (5 minutes)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• Pixel stream (p. 617) 2017-06-15 (updated 2018-10-26) (4 minutes)
• Some notes on reverse-engineering The Wizard's Castle (p. 1970) 2018-04-26 (9 minutes)
• Oscilloscope screens (p. 578) 2018-06-05 (2 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• A review of Wirth's Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)
• Techniques for, e.g., avoiding indexed-offset addressing on the 8080 (p. 3166) 2019-07-20 (updated 2019-07-24) (27 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)

Notes concerning “The range minimum query problem”

- Worst-case-logarithmic-time reduction over arbitrary intervals over arbitrary semigroups (p. 1021) 2012-12-04 (5 minutes)
- a logarithmic-time alternative to summed-area tables for reducing arbitrary semigroup operations over arbitrary ranges (a generalization of RMQ segment trees) (p. 1190) 2012-12-06 (updated 2013-05-17) (10 minutes)
- Robust hashsplitting with sliding Range Minimum Query (p. 733) 2016-09-05 (7 minutes)
- Some notes on morphology, including improvements on Urbach and Wilkinson’s erosion/dilation algorithm (p. 216) 2019-01-04 (updated 2019-11-12) (26 minutes)
- Tabulating your top event of the month efficiently using RMQ algorithms (p. 619) 2019-03-19 (8 minutes)

Notes concerning “Robotics”

• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• A one-motor robot (p. 118) 2015-09-03 (13 minutes)
• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)

Notes concerning “Robots”

• Charge transfer servo (p. 3017) 2013-05-17 (2 minutes)
• Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
• Servoing a V-plotter with a webcam? (p. 62) 2017-02-16 (3 minutes)
• High-precision control of low-stiffness sytems with bounded-Q resonances (p. 1002) 2017-05-29 (updated 2017-06-01) (4 minutes)
• Ideas to pursue (p. 1084) 2018-05-05 (updated 2018-08-16) (6 minutes)
• Image approximation (p. 2394) 2019-05-14 (10 minutes)
• Harmonic motion chain robot (p. 2197) 2019-08-16 (2 minutes)
• Rubber wheel pinch drive (p. 2912) 2019-08-16 (updated 2019-08-18) (8 minutes)
• Derivative based control (p. 2829) 2019-11-12 (6 minutes)

Notes concerning “Rosetta project”

- Holographic archival (p. 766) 2014-04-24 (10 minutes)
- Rosetta opacity hologram (p. 98) 2016-09-05 (8 minutes)

Notes concerning “Rust”

- Simple system language (p. 1659) 2013-05-17 (7 minutes)
- Similarities between Golang and Rust (p. 1523) 2017-01-11 (updated 2017-01-17) (7 minutes)

Notes concerning "Systems architecture"

• Running your regular desktop in QEMU? (p. 292) 2007 to 2009 (3 minutes)
• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• When and why exactly should your code "tell, not ask"? That is, use CPS? (p. 1051) 2014-01-08 (4 minutes)
• Stuff I've posted to kragen-tol over the years about post-HTTP (p. 1815) 2014-02-24 (12 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Fault-tolerant in-memory cluster computations using containers; or, SPARK, simplified and made flexible (p. 870) 2015-05-28 (updated 2016-06-22) (16 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Viral wiki (p. 1024) 2015-10-15 (3 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• DHT bulletin board (p. 3001) 2016-09-07 (7 minutes)
• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)
• MiniOS (p. 1091) 2016-12-28 (updated 2017-01-03) (6 minutes)
• Quicklayout (p. 2189) 2017-01-10 (updated 2017-01-18) (3 minutes)

• Burst computation (p. 1500) 2017-03-20 (13 minutes)
• Caching screen contents (p. 2362) 2017-06-14 (2 minutes)
• Nova RDOS (p. 1724) 2017-06-15 (22 minutes)
• Pixel stream (p. 617) 2017-06-15 (updated 2018-10-26) (4 minutes)
• Fast message router (p. 1853) 2017-06-15 (updated 2019-07-23) (15 minutes)
• Service-oriented email (p. 1302) 2017-06-20 (updated 2017-06-21) (15 minutes)
• A REST interface to a software transactional memory (p. 3014) 2017-06-21 (2 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• Cached SOA desktop (p. 2229) 2017-08-03 (updated 2018-10-26) (6 minutes)
• A minimal dependency processing system (p. 911) 2017-09-21 (3 minutes)
• Minimal transaction system (p. 2460) 2017-09-21 (5 minutes)
• Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
• Compressing REST transactions with per-connection state (p. 1117)

2018-04-27 (11 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• The TWI and I²C buses and better alternatives like CAN and RS-485 (p. 1638) 2018-06-28 (updated 2018-07-05) (24 minutes)
• Byte prefix tuple space (p. 427) 2018-07-14 (updated 2018-07-15) (4 minutes)
• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)
• Supervisor children for fault-tolerant Unix command-line programs (p. 3224) 2019-01-04 (3 minutes)
• Transactional event handlers (p. 139) 2019-01-24 (14 minutes)
• A review of Wirth's Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• Fast secure pubsub (p. 545) 2019-02-04 (updated 2019-12-03) (2 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Broken computer frustrations (p. 102) 2019-08-11 (2 minutes)
• Resurrecting duckling hashing (p. 3128) 2019-10-26 (updated 2019-11-10) (8 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)
• Byte stream gui applications (p. 128) 2019-11-29 (updated 2019-11-30) (17 minutes)
• Bytecode pubsub (p. 205) 2019-12-04 (6 minutes)
• Memory safe virtual machines (p. 975) 2019-12-04 (14 minutes)

Notes concerning “Safety”

• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Phosphorescent laser display (p. 1987) 2016-08-16 (8 minutes)
• Approaches to limiting self-replication (p. 1004) 2016-11-30 (7 minutes)
• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)
• Notes on a possible household air filter (p. 1961) 2018-05-05 (updated 2018-05-15) (10 minutes)
• Barrel safety (p. 1097) 2018-07-14 (3 minutes)
• Arduino safety (p. 3015) 2018-12-10 (4 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)

Notes concerning “Scheme”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Enumerating binary trees and their elements (p. 1445) 2007 to 2009 (4 minutes)
• Eur-Scheme: a simplified Ur-Scheme (p. 876) 2007 to 2009 (13 minutes)
• IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
• Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
• Separating implementation, optimization, and proofs (p. 780) 2019-06-26 (updated 2019-07-22) (41 minutes)
• A homoiconic language with a finite-map-based data model rather than lists? (p. 2630) 2019-09-25 (updated 2019-09-28) (46 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)

Notes concerning “Scholarship”

- Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
- Notch scorn (p. 115) 2019-04-20 (5 minutes)

Notes concerning “Scrubbers”

• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• House scrubber (p. 248) 2016-09-06 (updated 2019-11-25) (13 minutes)
• Scrubber mask (p. 90) 2019-05-08 (5 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Signed distance functions (SDFs)”

Notes concerning “Sdr”

• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Transmitting low-power TV signals around your house via RF modulation with an SDR (p. 1950) 2019-12-01 (6 minutes)

Notes concerning “Search”

• A filesystem design sketch modeled on Lucene (p. 1624) 2007 to 2009 (43 minutes)
• Constant-space grep (p. 296) 2014-02-24 (3 minutes)
• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)
• Fencepost cognitive interface errors in text editing (p. 993) 2019-04-24 (24 minutes)
• Dercuano search (p. 1532) 2019-05-16 (2 minutes)

Notes concerning “The Secure Scuttlebutt protocol”

- Gaim group chat (p. 2677) 2007 to 2009 (3 minutes)
- Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)
- Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
- Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
- Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)

Notes concerning “Security”

• lattices, powersets, bitstrings, and efficient OLAP (p. 2345) 2014-04-24 (17 minutes)
• How to use “correct horse battery staple” as an encryption key, including a recommended 4096-word list (p. 2522) 2014-04-24 (44 minutes)
• A simple content-addressable storage-server protocol (p. 774) 2015-09-03 (3 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• What’s wrong with CoAP (p. 560) 2017-06-15 (3 minutes)
• Distributed computing environment (p. 776) 2017-07-19 (8 minutes)
• Obscurity platform (p. 2991) 2018-04-27 (1 minute)
• Bokeh pointcasting (p. 92) 2019-09-08 (updated 2019-09-09) (16 minutes)
• Resurrecting duckling hashing (p. 3128) 2019-10-26 (updated 2019-11-10) (8 minutes)

Notes concerning “Umut Acar’s “self-adjusting computation””

• Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)
• Fast secure pubsub (p. 545) 2019-02-04 (updated 2019-12-03) (2 minutes)
• Robust local search in vector spaces using adaptive step sizes, and thoughts on extending quasi-Newton methods (p. 1138) 2019-08-17 (updated 2019-09-15) (15 minutes)

Notes concerning "Self-replication"

• The economics of solar energy (p. 1175) 2008 (27 minutes)
• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
• Planar lookup tables (p. 3105) 2014-04-24 (2 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
• The book written in itself (p. 2400) 2016-06-12 (updated 2016-06-14) (18 minutes)
• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• 2016 outlook for automated fabrication and 3-D printing (p. 2316) 2016-08-11 (20 minutes)
• Clanking replicators (p. 171) 2016-11-30 (3 minutes)
• Approaches to limiting self-replication (p. 1004) 2016-11-30 (7 minutes)
• Self replication changes (p. 2842) 2017-01-16 (5 minutes)
• Clay fabrication objectives (p. 2111) 2017-01-16 (updated 2017-01-17) (3 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• A brief note on autonomous cyclic fabrication systems from inorganic raw materials (p. 2965) 2018-04-27 (1 minute)
• Elastic metamaterials (p. 719) 2019-03-19 (17 minutes)
• Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)
• Maximal-flexibility designs for printable building blocks (p. 1839) 2019-04-20 (18 minutes)
• Plastic cutters (p. 1074) 2019-04-20 (5 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Computation with strain (p. 2812) 2019-06-13 (17 minutes)
• Foil origami robots (p. 2286) 2019-06-13 (updated 2019-06-14) (10 minutes)

Notes concerning “Self-sustaining systems”

• Simplified computing, down to the level of mining raw materials (p. 691) 2015-09-03 (22 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• MiniOS (p. 1091) 2016-12-28 (updated 2017-01-03) (6 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Real time windowing (p. 891) 2017-08-03 (9 minutes)
• Frustration (p. 3263) 2018-04-27 (2 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)

Notes concerning “Self”

• Bicicleta maps (p. 582) 2007 to 2009 (2 minutes)
• Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)

Notes concerning “Sensors”

• Ghettobotics: making robots out of trash (p. 2747) 2013-05-17 (41 minutes)
• The Tinkerer’s Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Starfield servo (p. 1709) 2016-08-30 (updated 2018-11-07) (13 minutes)
• Compressed sensing microscope (p. 306) 2016-10-06 (7 minutes)
• Passive ultrasound sonar (p. 295) 2016-12-28 (1 minute)
• FM chirp sonar (p. 351) 2017-07-04 (1 minute)
• Measuring submicron displacements by pitch bending a slide guitar (p. 905) 2019-05-05 (18 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Processing halftoning (p. 915) 2019-09-01 (15 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)
• Audio tablet (p. 2178) 2019-09-28 (7 minutes)
• Camera flash extrapolation (p. 2792) 2019-11-12 (6 minutes)

Notes concerning “Serialization”

• HTML is terser and more robust than S-expressions (p. 562) 2007 to 2009 (4 minutes)
• Fast geographical maps on Android (p. 455) 2015-10-16 (9 minutes)
• Secure, self-describing, self-delimiting serialization for Python (p. 243) 2017-04-11 (8 minutes)
• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Immediate mode productive grammars (p. 898) 2018-09-13 (8 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)

Notes concerning “Sewage”

• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Home dehumidifier (p. 3131) 2018-05-20 (updated 2019-04-02) (12 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Structure from shading”

- Reconstructing a 3-D Lambertian surface from video with a moving light source (p. 3296) 2016-09-16 (1 minute)
- Cloth structure from shading (p. 84) 2019-09-01 (2 minutes)

Notes concerning “Sheet cutting”

• Planar lookup tables (p. 3105) 2014-04-24 (2 minutes)
• Making a mechanical state machine via sheet cutting (p. 1013) 2014-04-24 (updated 2015-09-03) (7 minutes)
• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• Heckballs: a laser-cuttable MDF set of building blocks (p. 2782) 2016-08-17 (updated 2016-08-30) (24 minutes)
• Sun cutter (p. 56) 2016-09-06 (9 minutes)
• Simple state machines (p. 760) 2016-09-19 (updated 2016-09-24) (8 minutes)
• Laser cut next step (p. 824) 2018-04-27 (updated 2018-04-30) (7 minutes)
• Friction-cutting plastic (p. 2412) 2019-02-25 (8 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)
• Extending Heckballs (p. 3239) 2019-11-26 (6 minutes)

Notes concerning “SIMD instructions”

• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• Designing an archival virtual machine (p. 3203) 2016-05-12 (6 minutes)
• Further notes on algebras for dark silicon (p. 1753) 2016-09-17 (updated 2017-04-18) (23 minutes)
• Gradient rendering (p. 583) 2016-09-24 (11 minutes)
• Counting the number of spaces to the left in parallel (p. 1067) 2016-10-11 (5 minutes)
• Vector instructions (p. 2977) 2017-07-19 (2 minutes)
• Vectorized prefix sum (p. 529) 2017-07-19 (5 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• Bootstrapping instruction set (p. 459) 2018-11-06 (updated 2019-05-03) (19 minutes)
• Observable transaction possibilities (p. 2086) 2019-06-15 (10 minutes)

Notes concerning “Physical system simulation”

• We should use end-to-end optimization algorithms for 3-D printing design (p. 1550) 2015-09-03 (14 minutes)
• Topics to study in 2016 (p. 678) 2016-10-27 (updated 2016-11-15) (37 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Matrix exponentiation linear circuits (p. 355) 2018-12-18 (4 minutes)

Notes concerning “Sketchpad”

• Constant space lists (p. 3062) 2018-12-10 (10 minutes)
• Dercuano drawings (p. 64) 2019-04-30 (updated 2019-05-30) (18 minutes)
• Designing a drawing editor for well-factored drawings (p. 2115) 2019-05-07 (9 minutes)

Notes concerning “Small is beautiful”

• Notes on reading eForth (p. 1398) 2007 to 2009 (9 minutes)
• Notes on reading eForth 1.0 for the 8086 (p. 541) 2007 to 2009 (5 minutes)
• Eur-Scheme: a simplified Ur-Scheme (p. 876) 2007 to 2009 (13 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
• Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
• A survey of small TCP/IP implementations (p. 2616) 2007 to 2009 (4 minutes)
• Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
• mechanical computation: with Merkle gates, height fields, and thread (p. 2494) 2010-06-28 (36 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• An algebraic approach to 3D geometry (p. 669) 2014-06-03 (updated 2014-06-29) (22 minutes)
• Archival with a universal virtual computer (UVC) (p. 399) 2014-06-29 (17 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• Simplifying computing systems by having fewer kinds of graphics (p. 1110) 2015-10-13 (10 minutes)
• Minimal GUI libraries (p. 663) 2015-11-14 (updated 2015-11-15) (5 minutes)
• Logarithmic maintainability and coupling (p. 2341) 2015-11-23 (7 minutes)
• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)
• Executable scholarship, or algorithmic scholarly communication (p. 2137) 2016-08-11 (13 minutes)
• What’s the dumbest register allocator that might give you reasonable performance? (p. 2596) 2016-10-11 (15 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Cheap frequency detection (p. 3026) 2017-06-29 (updated 2019-06-19) (50 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177)

2017-07-19 (updated 2019-07-10) (21 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• A minimal window system (p. 1545) 2018-04-27 (updated 2018-10-26) (12 minutes)
• Toward a minimal PEG parsing engine (p. 955) 2018-06-06 (4 minutes)
• Whistle detection (p. 357) 2018-06-06 (updated 2018-12-02) (18 minutes)
• Minimal imperative language (p. 2175) 2018-12-10 (7 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• A review of Wirth's Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• Honk development (p. 1188) 2019-03-21 (2 minutes)
• An 8080 opcode map in octal (p. 1059) 2019-08-28 (updated 2019-11-24) (11 minutes)
• Notes on Óscar Toledo G.'s bootOS (p. 277) 2019-10-07 (updated 2019-10-08) (28 minutes)
• 10tcl ui (p. 1823) 2019-12-06 (17 minutes)
• Forth assembling (p. 940) 2019-12-08 (updated 2019-12-11) (18 minutes)
• Immediate-mode PEG parsers in assembly language (p. 365) 2019-12-10 (updated 2019-12-11) (21 minutes)
• Can you eliminate backpatching? (p. 1769) 2019-12-17 (8 minutes)

Notes concerning “Smalltalk”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• Using bytecode won’t make your interpreter fast (p. 226) 2007 to 2009 (26 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Compact namespace sharing (p. 237) 2016-07-25 (7 minutes)
• Queueing messages to amortize dynamic dispatch and take advantage of hardware heterogeneity (p. 586) 2016-09-17 (13 minutes)
• Thredsnek: a tiny Python-flavored programming language (p. 1172) 2017-03-20 (7 minutes)
• Compact code cpu (p. 397) 2017-07-19 (3 minutes)
• Rarely are function-local variables in Forth justified (p. 1055) 2018-04-27 (10 minutes)
• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Introduction to closures (p. 1403) 2019-12-07 (5 minutes)

Notes concerning “Solar”

• The economics of solar energy (p. 1175) 2008 (27 minutes)
• Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)
• Pensamientos acerca de diseñar un calefón solar (p. 117) 2012-10-15 (2 minutes)
• Más pensamientos acerca de diseñar un calefón solar (p. 1713) 2012-10-15 (5 minutes)
• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Food storage (p. 2706) 2013-05-11 (updated 2013-05-17) (54 minutes)
• The future of the human energy market (2014) (p. 1846) 2014-04-24 (19 minutes)
• A hand-powered computer? (p. 624) 2015-09-03 (updated 2017-07-19) (11 minutes)
• Solar dehumidifier (p. 717) 2016-08-11 (5 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Sun cutter (p. 56) 2016-09-06 (9 minutes)
• Soldering with a compound parabolic concentrator or even just an imaging lens (p. 101) 2016-09-07 (2 minutes)
• Solar-powered portable computers (p. 2959) 2016-09-17 (updated 2018-10-28) (15 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
• Fast sea salt evaporator (p. 1087) 2017-06-01 (3 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• Japan can achieve energy autarky via solar energy, but not much before 2027 (p. 2819) 2017-07-12 (4 minutes)
• Solar computer 2 (p. 414) 2017-07-19 (3 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Terrestrial lithium supplies provide adequate energy storage to reach Kardashev Type 1 (p. 2123) 2019-07-25 (6 minutes)
• Energy storage efficiency (p. 1300) 2019-07-30 (4 minutes)
• Heliogen (p. 597) 2019-11-19 (6 minutes)

• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Phase change unplugged oven (p. 1433) 2019-12-15 (0 minutes)

Notes concerning “Sonar”

- Passive ultrasound sonar (p. 295) 2016-12-28 (1 minute)
- FM chirp sonar (p. 351) 2017-07-04 (1 minute)
- Audio tablet (p. 2178) 2019-09-28 (7 minutes)

Notes concerning “Sorting”

• Cycle sort (p. 2344) 2013-05-17 (1 minute)
• You can’t sort a file whose size is cubic in your RAM size in two passes, only quadratic (p. 2311) 2015-05-28 (5 minutes)
• Ternary mergesort (p. 2161) 2015-09-03 (2 minutes)
• An almost-in-place mergesort (p. 740) 2016-09-07 (5 minutes)
• The paradoxical complexity of computing the top N (p. 1890) 2017-01-04 (7 minutes)
• String tuple encoding (p. 2419) 2017-04-28 (2 minutes)
• ASCIIbetically homomorphic encodings of general data structures (p. 3261) 2017-06-15 (2 minutes)
• Sorting in logic (p. 498) 2019-12-28 (2 minutes)

Notes concerning “Spaced practice”

• How to use “correct horse battery staple” as an encryption key, including a recommended 4096-word list (p. 2522) 2014-04-24 (44 minutes)
• The ultimate capacity of human memory if spaced-practice memorization works as advertised (p. 2442) 2017-01-04 (updated 2017-01-08) (14 minutes)

Notes concerning “Spark”

Notes concerning “Sparkling”

- Starfield servo (p. 1709) 2016-08-30 (updated 2018-11-07) (13 minutes)
- Sparkle wheel display (p. 1738) 2017-05-10 (6 minutes)
- Photodiode camera (p. 2265) 2019-09-04 (16 minutes)

Notes concerning “Sparks”

• Marking metal surfaces with arcs (p. 1993) 2016-10-06 (4 minutes)
• Compressed sensing microscope (p. 306) 2016-10-06 (7 minutes)
• Spark particulate sieve (p. 2047) 2016-10-06 (updated 2016-10-11) (7 minutes)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)

Notes concerning “Sparse filters”

• Some speculative thoughts on DSP algorithms (p. 2590) 2014-04-24 (20 minutes)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Recurrent comb cascade (p. 483) 2018-11-09 (updated 2018-11-10) (2 minutes)
• Sparse filters (p. 834) 2018-12-02 (4 minutes)
• Real-time bokeh algorithms, and other convolution tricks (p. 2661) 2018-12-18 (updated 2019-08-15) (23 minutes)
• The miraculous low-rank SVD approximate convolution algorithm (p. 747) 2019-08-14 (updated 2019-08-15) (31 minutes)
• Image filtering with an approximate Gabor wavelet or Morlet wavelet using a cascade of sparse convolution kernels (p. 547) 2019-08-31 (updated 2019-09-08) (28 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)
• Sparse sinc (p. 1880) 2019-09-15 (10 minutes)
• Sparse filter optimization (p. 2610) 2019-11-01 (5 minutes)
• Magic sinewave filter (p. 200) 2019-12-17 (6 minutes)

Notes concerning “Speech synthesis”

• Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
• Alphanumerenglish (p. 2882) 2015-04-06 (updated 2016-07-27) (6 minutes)
• English diphones (p. 2061) 2019-12-03 (5 minutes)

Notes concerning “Splines”

• Achieving smooth curves in scanline image generation (p. 1507) 2013-05-17 (1 minute)
• Polynomial-spline FIR kernels by integrating sparse kernels (p. 1819) 2014-04-24 (12 minutes)
• Very fast FIR filtering with time-domain zero stuffing and splines (p. 1146) 2015-09-03 (updated 2015-09-07) (13 minutes)
• Gaussian spline reconstruction (p. 656) 2016-06-05 (updated 2016-06-06) (5 minutes)
• Rasterizing polies (p. 2023) 2017-07-19 (3 minutes)
• A bag of candidate techniques for sparse filter design (p. 3250) 2019-09-01 (18 minutes)

Notes concerning “Spreadsheets”

- Automatic dependency management (p. 881) 2015-05-28 (updated 2015-09-03) (5 minutes)
- Usability of scientific calculators (p. 2379) 2016-09-29 (19 minutes)
- An IDE modeled on video games (p. 1959) 2019-04-08 (5 minutes)

Notes concerning “SQL”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Gittable sql (p. 85) 2015-09-25 (updated 2015-09-26) (6 minutes)
• The history of NoSQL and dbm (p. 45) 2017-04-10 (16 minutes)
• Database explorer (p. 225) 2017-06-20 (2 minutes)
• Toward a lightweight, high-performance software prototyping environment (p. 2938) 2018-12-10 (15 minutes)
• What it means that HTML is “not a programming language”, and why the ignorant sometimes think otherwise (p. 1555) 2019-09-09 (updated 2019-10-01) (24 minutes)

Notes concerning “Stacks”

• Why John Backus Was on the Wrong Track (p. 2722) 2007 (updated 2019-05-05) (48 minutes)
• A stack of coordinate contexts (p. 2987) 2007 to 2009 (9 minutes)
• bytecode interpreters for tiny computers (p. 2847) 2007-09 (61 minutes)
• Forth with named stacks (p. 2101) 2014-02-24 (7 minutes)
• Memoize the stack (p. 2021) 2015-09-03 (5 minutes)
• Would Synthgramelodia make a good base for livecoding music? (p. 2540) 2015-09-03 (8 minutes)
• Making the CPU instruction set a usable interactive user interface (p. 59) 2015-09-17 (8 minutes)
• A one-operand stack machine (p. 3242) 2016-07-24 (updated 2016-07-25) (12 minutes)
• Circuit notation (p. 1161) 2016-09-08 (updated 2017-04-18) (7 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Generalizing my RPN calculator to support refactoring (p. 969) 2016-10-17 (12 minutes)
• A sketch of a minimalist bytecode for object-oriented languages (p. 1790) 2017-03-20 (updated 2017-06-20) (13 minutes)
• Reflections on rebraining calculators with this RPN calculator code I just wrote (p. 1717) 2017-04-11 (4 minutes)
• Secure, self-describing, self-delimiting serialization for Python (p. 243) 2017-04-11 (8 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• A stack of stacks for simple modular electronics (p. 1779) 2017-06-27 (5 minutes)
• An RPN CPU instruction set doubling as user interface (p. 177) 2017-07-19 (updated 2019-07-10) (21 minutes)
• Lisp 1.5 in a stack bytecode: can we get from machine code to Lisp in 45 lines of code? (p. 952) 2018-04-27 (4 minutes)
• A simple virtual machine for vector math? (p. 986) 2018-11-06 (updated 2018-11-09) (15 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• Three-stack generic macro assembler (design sketch) (p. 1336) 2019-04-30 (8 minutes)

Notes concerning “State machines”

• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Simple state machines (p. 760) 2016-09-19 (updated 2016-09-24) (8 minutes)
• Finite function circuits (p. 2050) 2017-02-16 (updated 2019-05-17) (29 minutes)
• Parallel DFA execution (p. 2337) 2017-04-18 (9 minutes)

Notes concerning “VPRI STEPS”

- Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)
- More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
- Frustration (p. 3263) 2018-04-27 (2 minutes)

Notes concerning “STM32 microcontrollers”

• Low-power microcontrollers for a low-power computer (p. 2602) 2016-09-06 (updated 2018-10-28) (18 minutes)
• A stack of stacks for simple modular electronics (p. 1779) 2017-06-27 (5 minutes)
• The TWI and I²C buses and better alternatives like CAN and RS-485 (p. 1638) 2018-06-28 (updated 2018-07-05) (24 minutes)
• Notes on the STM32 microcontroller family (p. 3176) 2018-06-30 (updated 2018-11-12) (42 minutes)
• Can you turbocharge the STM32 ADC to build an oscilloscope? (p. 137) 2018-07-14 (5 minutes)
• Inductor thermocouple sensing (p. 2037) 2019-06-01 (21 minutes)
• Really simple lab power supply (p. 240) 2019-12-10 (7 minutes)

Notes concerning “Strategy”

- Barcode receipts (p. 2359) 2007 to 2009 (6 minutes)
- High-risk behavior in context (p. 1485) 2007 to 2009 (5 minutes)
- Improving “science” in eSpeak's lexicon (p. 188) 2007 to 2009 (updated 2019-06-27) (15 minutes)
- Only a constant factor worse (p. 1648) 2013-05-17 (16 minutes)
- Dollar auctions and tournaments in human society (p. 884) 2013-05-17 (7 minutes)
- Complementary goods in home economics (p. 1878) 2017-07-19 (3 minutes)
- A tournament to decide which notes to devote attention to polishing (p. 1195) 2017-07-19 (2 minutes)
- Dutch auction raffle (p. 2474) 2018-06-05 (3 minutes)
- When should you give up waiting for the bus and just walk? (p. 2280) 2019-04-24 (5 minutes)
- Better be weird (p. 1831) 2019-06-17 (updated 2019-06-24) (9 minutes)

Notes concerning “Subterranean living”

Notes concerning “Surveys”

- Intro to algorithms (p. 2625) 2016-09-06 (4 minutes)
- Top algorithms (p. 913) 2018-07-29 (4 minutes)

Notes concerning “Sync”

• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
• Rsync message base (p. 2463) 2019-11-08 (updated 2019-11-30) (29 minutes)

Notes concerning “Syntax”

• Enumerating binary trees and their elements (p. 1445) 2007 to 2009 (4 minutes)
• Forth looping (p. 2025) 2007 to 2009 (16 minutes)
• IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
• HTML is terser and more robust than S-expressions (p. 562) 2007 to 2009 (4 minutes)
• Notes on Raph Levien's "Io" Programming Language (p. 1740) 2007 to 2009 (10 minutes)
• Iterative string formatting (p. 1392) 2013-05-17 (9 minutes)
• Forth with named stacks (p. 2101) 2014-02-24 (7 minutes)
• Range literals (p. 1719) 2014-04-24 (6 minutes)
• An algebraic approach to 3D geometry (p. 669) 2014-06-03 (updated 2014-06-29) (22 minutes)
• Entry-C: a Simula-like backwards-compatible object-oriented C (p. 564) 2015-04-05 (updated 2017-04-03) (24 minutes)
• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• Toward a language for hacking around with natural-language processing algorithms (p. 1681) 2016-09-08 (7 minutes)
• Circuit notation (p. 1161) 2016-09-08 (updated 2017-04-18) (7 minutes)
• Graph construction (p. 3226) 2016-09-08 (updated 2017-07-19) (23 minutes)
• Pattern matching and finite functions (p. 1235) 2017-05-10 (14 minutes)
• Relational modeling (p. 1102) 2017-05-17 (updated 2017-06-01) (6 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• Ideal language syntax (p. 3260) 2017-07-19 (1 minute)
• General purpose layout syntax (p. 3117) 2017-11-10 (updated 2019-09-01) (34 minutes)
• Flexible text query (p. 2900) 2018-07-14 (4 minutes)
• Leconscrip: a family of JS subsets for BubbleOS (p. 2126) 2018-11-23 (2 minutes)
• How small can we make a comfortable subset of JS? (p. 1348) 2018-11-27 (updated 2018-12-02) (3 minutes)
• Binate and KANREN (p. 3189) 2018-12-02 (3 minutes)
• Minimal imperative language (p. 2175) 2018-12-10 (7 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)
• IMGUI programming language (p. 103) 2019-01-01 (updated 2019-07-30) (21 minutes)
• A review of Wirth’s Project Oberon book (p. 431) 2019-02-04 (updated 2019-03-19) (63 minutes)
• Dercuano grinding (p. 2475) 2019-10-01 (12 minutes)

Notes concerning “Synthesis”

• Trees as code (p. 2488) 2016-05-10 (4 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)

Notes concerning “Tcl/Tk”

• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• IMGUI programming compared to Tcl/Tk (p. 2333) 2018-12-24 (updated 2018-12-31) (8 minutes)

Notes concerning “TCP/IP”

• A survey of small TCP/IP implementations (p. 2616) 2007 to 2009 (4 minutes)
• How inefficient is SNAT hole-punching via random port trials? (p. 1155) 2018-04-27 (2 minutes)

Notes concerning “Telescopes”

• Can you read the lunar lander’s plaque from Earth? Or write a new one? (p. 125) 2015-09-03 (9 minutes)
• Seeing the Apollo flags from Earth would require a telescope 27× the size of the Gran Telescopio Canarias (p. 309) 2019-04-10 (updated 2019-04-16) (2 minutes)

Notes concerning “Terminals”

• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• A proposal to support hypertext links in ANSI terminals (p. 486) 2013-05-17 (updated 2019-12-26) (13 minutes)
• What does a futuristic OS look like? (p. 2163) 2017-08-18 (updated 2019-05-05) (6 minutes)
• General purpose layout syntax (p. 3117) 2017-11-10 (updated 2019-09-01) (34 minutes)
• Speculative plans for BubbleOS (p. 2128) 2018-10-28 (updated 2019-02-24) (12 minutes)
• Dehydrating processes and other interaction models (p. 3208) 2018-12-28 (updated 2019-01-01) (36 minutes)

Notes concerning “Testing”

• Literate programs should include example output, like Jupyter, but Jupyter is imperfect (p. 1308) 2018-04-27 (3 minutes)
• The uses of introspection, reflection, and personal supercomputers in software testing (p. 2306) 2019-02-04 (updated 2019-03-11) (12 minutes)

Notes concerning “Textiles”

• UHMWPE clothes could be lightweight and sturdy (p. 3071) 2018-06-05 (3 minutes)
• Sandwich theory (p. 2450) 2019-08-05 (updated 2019-08-29) (31 minutes)
• Cloth structure from shading (p. 84) 2019-09-01 (2 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)

Notes concerning "The future"

- The economics of solar energy (p. 1175) 2008 (27 minutes)
- Predictions for future technological development (2008) (p. 341) 2008-04-19 (11 minutes)
- Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
- Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
- Who is inventing the future in 2013? (p. 897) 2013-05-17 (1 minute)

- Review notes for Chris Anderson's "Makers" (p. 1072) 2013-05-17 (5 minutes)
- The future of the human energy market (2014) (p. 1846) 2014-04-24 (19 minutes)
- 2025 manufacturing and economics scenario (p. 699) 2014-04-24 (24 minutes)
- What might Diamond-Age-like phyles look like in the real 21st century? (p. 1599) 2014-04-24 (9 minutes)
- In a world with ubiquitous surveillance, what does politics look like? (p. 1615) 2014-04-24 (11 minutes)
- Exponential technology and capital (p. 406) 2016-02-18 (updated 2017-07-19) (8 minutes)
- The internet is probably not going to collapse for economic reasons (p. 3194) 2016-09-06 (9 minutes)
- Hybrid RAM (p. 2877) 2016-09-24 (5 minutes)
- Current hardware trends tend toward raytracing (p. 1351) 2016-10-07 (4 minutes)
- Where did the Rubius comic book come from? (p. 1564) 2017-01-10 (4 minutes)
- Japan can achieve energy autarky via solar energy, but not much before 2027 (p. 2819) 2017-07-12 (4 minutes)
- The imbalance inherent in copyright systems (p. 2158) 2017-07-19 (2 minutes)
- Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
- Commentaries on reading Engelbart's "Augmenting Human Intellect" (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
- Gardening machines (p. 2365) 2019-04-02 (updated 2019-04-24) (32 minutes)

Notes concerning "Thermodynamics"

• Air conditioning (p. 1665) 2007 to 2009 (21 minutes)
• Pensamientos acerca de diseñar un calefón solar (p. 117) 2012-10-15 (2 minutes)
• Más pensamientos acerca de diseñar un calefón solar (p. 1713) 2012-10-15 (5 minutes)
• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Evaporation chimney (p. 2147) 2013-05-17 (13 minutes)
• Heat exchangers modeled on retia mirabilia might reach 4 TW/m³ (p. 1487) 2014-07-16 (updated 2019-08-21) (14 minutes)
• Can you read the lunar lander's plaque from Earth? Or write a new one? (p. 125) 2015-09-03 (9 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Phase-change heat reservoirs for household climate control (p. 2257) 2016-06-14 (updated 2016-06-17) (13 minutes)
• Thermodynamic systems in housing (p. 2804) 2016-06-28 (24 minutes)
• Regenerator gas kiln (p. 2653) 2016-09-05 (updated 2017-04-10) (9 minutes)
• Regenerative fuel air cutting (p. 2622) 2016-09-06 (4 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Sun cutter (p. 56) 2016-09-06 (9 minutes)
• Laser ablation of zinc or pewter for printed circuit boards (p. 2799) 2016-09-19 (4 minutes)
• Cross current zone melting (p. 1872) 2016-10-06 (1 minute)
• Recuperator heat storage (p. 594) 2016-11-01 (updated 2019-08-21) (4 minutes)
• A design sketch of an air conditioner powered by solar thermal power (p. 2233) 2016-12-22 (updated 2017-01-04) (29 minutes)
• Millikiln (p. 2581) 2017-01-17 (updated 2017-03-02) (4 minutes)
• An electric furnace the size of a sake cup (p. 666) 2017-02-25 (updated 2017-03-02) (10 minutes)
• Passivhaus seasonal thermal store (p. 1723) 2017-03-02 (updated 2017-03-07) (2 minutes)
• Passive dehumidifier (p. 3256) 2017-03-20 (14 minutes)
• Ice pants (p. 298) 2017-04-04 (updated 2019-01-22) (17 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
• Fast sea salt evaporator (p. 1087) 2017-06-01 (3 minutes)
• A quintuple-acting vacuum cascade to recycle heat for more efficient distillation and desalination (p. 519) 2017-06-21 (updated 2019-12-27) (3 minutes)
• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)

• Pipe dome (p. 3068) 2017-07-19 (7 minutes)
• Rubber air conditioner (p. 2791) 2017-07-19 (2 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Deep freeze (p. 1465) 2017-08-22 (updated 2019-01-22) (7 minutes)

• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• Data archival on gold leaf or Mylar with DVD-writer lasers or sparks (p. 1455) 2018-04-27 (5 minutes)
• Radiant heating (p. 493) 2018-05-20 (3 minutes)
• Heating my apartment with a plastic tub of hot water (p. 1310) 2018-06-17 (3 minutes)
• Hot water bottles (p. 3066) 2018-07-14 (4 minutes)
• Notes on circuitry for the Nutra seed activator (p. 3099) 2018-08-16 (20 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• Friction-cutting plastic (p. 2412) 2019-02-25 (8 minutes)
• Sous vide (p. 2921) 2019-04-02 (2 minutes)
• Waterfryer (p. 1462) 2019-04-20 (1 minute)
• A phase-change soldering iron (p. 2270) 2019-05-08 (updated 2019-05-09) (14 minutes)
• Things in Dercuano that would be big if true (p. 3136) 2019-05-24 (updated 2019-08-21) (24 minutes)
• Reducing nighttime bedroom CO_2 levels (p. 478) 2019-07-08 (updated 2019-07-09) (14 minutes)
• Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)
• Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) 2019-11-24 (2 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)
• Phase change unplugged oven (p. 1433) 2019-12-15 (0 minutes)

Notes concerning “Time domain”

- Cheap shit ultrawideband (p. 2776) 2013-05-17 (10 minutes)
- Time domain lightning triggering (p. 2534) 2013-05-17 (4 minutes)

Notes concerning “Time series”

• Efficiently querying a log of everything that ever happened (p. 2506) 2015-09-03 (7 minutes)
• Interactive calculator 0 (p. 1453) 2015-09-17 (2 minutes)
• Time series data type (p. 304) 2016-08-26 (3 minutes)
• Kafka-like feeds for offline-first browser apps (p. 903) 2017-08-03 (5 minutes)
• Minimal distributed streams (p. 1844) 2018-04-27 (5 minutes)
• Archival of hypertext with arbitrary interactive programs: a design outline (p. 2472) 2018-11-09 (3 minutes)

Notes concerning “The Tinkerer’s Tricorder”

• The Tinkerer’s Tricorder (p. 72) 2013-05-17 (updated 2014-04-24) (27 minutes)
• Trellis-coded buttons to run a whole keyboard off two microcontroller pins (p. 2011) 2013-05-17 (updated 2019-06-13) (30 minutes)

Notes concerning “Toledo family”

Notes concerning “Toxicology”

• Coolants (p. 3235) 2017-07-04 (updated 2017-07-12) (11 minutes)
• Cold plasma oxidation (p. 2406) 2019-05-01 (updated 2019-08-21) (7 minutes)

Notes concerning “Trading”

• Incremental MapReduce for Abelian-group reduction functions (p. 331) 2015-09-03 (4 minutes)
• Assigning consistent order IDs (p. 1042) 2015-09-03 (3 minutes)
• Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) 2019-07-03 (6 minutes)
• An affine-arithmetic database index for rapid historical securities formula queries (p. 2275) 2019-09-15 (15 minutes)

Notes concerning “Transactions”

• How can we usefully cache screen images for incrementalization? (p. 1077) 2013-05-17 (18 minutes)
• Simple dependencies in software (p. 2447) 2014-06-05 (9 minutes)
• Transactional screen updates (p. 2907) 2015-04-01 (10 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)
• Kogluktualuk: an operating system based on caching coarse-grained deterministic computations (p. 257) 2016-07-23 (21 minutes)
• A REST interface to a software transactional memory (p. 3014) 2017-06-21 (2 minutes)
• A minimal dependency processing system (p. 911) 2017-09-21 (3 minutes)
• Minimal transaction system (p. 2460) 2017-09-21 (5 minutes)
• Composing code gobbets with implicit dependencies (p. 2437) 2018-04-27 (updated 2019-05-21) (3 minutes)
• Transactional event handlers (p. 139) 2019-01-24 (14 minutes)
• Transactional memory, immediate-mode structured graphics, serialization, backtracking, and parsing (p. 1123) 2019-01-25 (7 minutes)
• Fast secure pubsub (p. 545) 2019-02-04 (updated 2019-12-03) (2 minutes)
• Immutability-based filesystems: interfaces, problems, and benefits (p. 1672) 2019-02-08 (updated 2019-03-19) (23 minutes)
• Memory safe virtual machines (p. 975) 2019-12-04 (14 minutes)

Notes concerning “Transport”

• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• Ultralight tunnel personal rapid transit (p. 706) 2019-03-11 (15 minutes)

Notes concerning “Tree rewriting”

- IRC bots with object-oriented equational rewrite rules (p. 838) 2007 to 2009 (6 minutes)
- Term rewriting (p. 3221) 2017-07-19 (3 minutes)

Notes concerning “Types”

• APL with typed indices (p. 3264) 2013-05-17 (11 minutes)
• A principled rethinking of array languages like APL (p. 1995) 2015-05-16 (updated 2019-09-30) (31 minutes)
• What is the type of lerp? (p. 1985) 2017-01-08 (5 minutes)
• Generic programming with proofs, specification, refinement, and specialization (p. 958) 2017-05-10 (6 minutes)
• Patterns for failure-free, bounded-space, and bounded-time programming (p. 925) 2018-04-27 (updated 2019-09-10) (42 minutes)

Notes concerning “Typing”

• The AL programming language, dimensional analysis, and typing: do different dimensions really exist? (p. 731) 2007 to 2009 (2 minutes)

• ML’s value restriction and the Modula-3 typing system (p. 1763) 2007 to 2009 (3 minutes)
• Fixed point (p. 807) 2014-04-24 (1 minute)

Notes concerning “Typography”

- The continuous-web press and the continuous press of the World-Wide Web (p. 1134) 2017-03-20 (6 minutes)
- Ideal language syntax (p. 3260) 2017-07-19 (1 minute)
- Dercuano stylesheet notes (p. 374) 2019-04-28 (updated 2019-05-09) (72 minutes)
- How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)
- Dercuano grinding (p. 2475) 2019-10-01 (12 minutes)

Notes concerning “Ubicomp”

Notes concerning “UHMWPE”

• Spring energy density (p. 3106) 2016-05-28 (updated 2016-06-06) (13 minutes)
• String cutting cardboard (p. 515) 2016-06-30 (5 minutes)
• Differential spiral cam (p. 512) 2017-07-19 (9 minutes)
• Dyneema (p. 123) 2017-07-19 (2 minutes)
• Absurd household materials (p. 532) 2018-04-26 (updated 2018-05-18) (8 minutes)
• You can stuff a UHMWPE hammock in your wallet (p. 799) 2018-05-15 (updated 2018-10-28) (11 minutes)
• UHMWPE clothes could be lightweight and sturdy (p. 3071) 2018-06-05 (3 minutes)
• Three phase oscillating belt (p. 214) 2018-10-28 (4 minutes)
• Spiral chinese windlass (p. 2915) 2019-07-23 (updated 2019-07-24) (7 minutes)
• Fabric optimization (p. 1526) 2019-10-28 (updated 2019-10-29) (17 minutes)
• Bootstrapping rope bridges and other tensile structures with UHMWPE-bearing drones (p. 2950) 2019-11-25 (5 minutes)

Notes concerning “Ultrasound”

• A unicast phased-array ultrasonic “radio” (p. 3077) 2013-05-17 (4 minutes)
• Passive ultrasound sonar (p. 295) 2016-12-28 (1 minute)
• Bubble display (p. 1542) 2017-01-24 (updated 2017-08-03) (1 minute)
• The Bleep ultrasonic modem for local data communication (p. 966) 2018-12-10 (updated 2018-12-11) (8 minutes)

Notes concerning “Uncorp”

Notes concerning “Unix”

• Git data (p. 2823) 2007 to 2009 (5 minutes)
• User-per-group (UPG), umask, and “Permission denied” on shared Git repos via ssh (p. 2481) 2007 to 2009 (4 minutes)
• Writing math in Unicode with the Compose key (p. 1863) 2007 to 2009 (2 minutes)
• Handling Landsat 8 images in limited RAM with netpbm (p. 1884) 2014-04-24 (4 minutes)
• What would a better Unix shell look like? (p. 2831) 2018-11-27 (1 minute)
• Raid zim (p. 253) 2019-01-17 (updated 2019-02-08) (1 minute)
• Text relational query (p. 1223) 2019-08-28 (10 minutes)

Notes concerning “Ur-Scheme”

- The coolest bug in Ur-Scheme (p. 1007) 2007 to 2009 (2 minutes)
- Eur-Scheme: a simplified Ur-Scheme (p. 876) 2007 to 2009 (13 minutes)
- Studies in Simplicity (p. 500) 2007 to 2009 (5 minutes)

Notes concerning "Utopias: proposals unlikely to be realized for improving things"

• What's wrong with ../../? (p. 2304) 2007 to 2009 (2 minutes)
• Notes and calculations on building luxury underground arcologies for whoever wants them (p. 1566) 2013-04-17 (updated 2019-08-27) (66 minutes)
• Critical defense mass (p. 2170) 2013-05-17 (14 minutes)
• A proposal to support hypertext links in ANSI terminals (p. 486) 2013-05-17 (updated 2019-12-26) (13 minutes)
• Logarithmic maintainability and coupling (p. 2341) 2015-11-23 (7 minutes)
• US$10M for a new, much better McMurdo Base, or less (p. 2879) 2016-05-18 (updated 2016-05-19) (7 minutes)
• Wikipedia people (p. 948) 2016-06-01 (6 minutes)
• Subterranean glazing (p. 1126) 2016-09-06 (25 minutes)
• The ultimate capacity of human memory if spaced-practice memorization works as advertised (p. 2442) 2017-01-04 (updated 2017-01-08) (14 minutes)
• Zombie contingency plan (p. 2656) 2017-07-19 (9 minutes)
• Frustration (p. 3263) 2018-04-27 (2 minutes)
• Exploration of using RF current sources instead of ELF voltage sources for mains power (p. 642) 2018-04-30 (updated 2018-07-05) (29 minutes)
• Atmospheric pressure harvesting phoenix egg (p. 2081) 2018-11-23 (14 minutes)
• Household thermal stores (p. 1533) 2018-12-02 (updated 2018-08-19) (27 minutes)
• A two-operand calculator supporting programming by demonstration (p. 2387) 2018-12-11 (22 minutes)
• Commentaries on reading Engelbart's "Augmenting Human Intellect" (p. 2091) 2018-12-24 (updated 2018-12-25) (25 minutes)
• Weregild (p. 3149) 2019-03-24 (3 minutes)
• Replacing fractional-reserve banking with a bond market disintermediated with a blockchain (p. 333) 2019-07-03 (6 minutes)
• The Suburbean: a minimally-mobile dwelling machine with months of autonomy (p. 1271) 2019-11-24 (updated 2019-12-03) (32 minutes)

Notes concerning “Video”

• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• Reconstructing a 3-D Lambertian surface from video with a moving light source (p. 3296) 2016-09-16 (1 minute)
• VCR oscilloscope (p. 213) 2017-05-10 (updated 2017-06-20) (2 minutes)
• Flying spot reilluminatable stage (p. 2358) 2017-05-15 (1 minute)
• Broadcast ECC with graceful degradation, or avoiding the cliff effect (p. 2045) 2018-12-18 (5 minutes)
• Audio video boustrophedon sync (p. 858) 2019-04-03 (2 minutes)
• Photodiode camera (p. 2265) 2019-09-04 (16 minutes)

Notes concerning “Vim”

• Quasimode keyboard (p. 2693) 2018-07-14 (24 minutes)
• Text editor design for e-ink displays (p. 3079) 2018-10-28 (23 minutes)

Notes concerning “Virtualization”

• Notes on running QEMU on Debian Etch (p. 1646) 2007 to 2009 (3 minutes)
• Running your regular desktop in QEMU? (p. 292) 2007 to 2009 (3 minutes)

Notes concerning “Vocoder”

• Dumb vocoder (p. 1391) 2017-05-10 (2 minutes)
• Can you make a vocoder simpler using CIC filters? (p. 2006) 2017-06-28 (updated 2018-06-17) (2 minutes)
• Is a phase vocoder or a bunch of PLLs a more efficient way to listen to all FM radio stations at once? (p. 1405) 2018-06-17 (updated 2019-07-29) (7 minutes)
• Using the Goertzel algorithm, the Minsky algorithm, PLLs, and prefix sums for frequency detection (p. 2679) 2019-06-16 (updated 2019-07-05) (39 minutes)

Notes concerning “Wang tiles”

Notes concerning “Water”

• Passively safe solar hot water (p. 1333) 2012-10-15 (updated 2012-10-16) (6 minutes)
• Waterproofing (p. 429) 2015-09-03 (4 minutes)
• ¿Se puede armar un colector solar de agua caliente que anda en invierno acá en Buenos Aires? (p. 2911) 2017-04-17 (2 minutes)
• Energy storage in a personal water tower: pretty impractical (p. 2044) 2017-07-19 (2 minutes)
• Salt slush refrigeration (p. 1230) 2017-08-22 (updated 2019-10-08) (12 minutes)
• Heating my apartment with a plastic tub of hot water (p. 1310) 2018-06-17 (3 minutes)
• Barrel safety (p. 1097) 2018-07-14 (3 minutes)
• Hot water bottles (p. 3066) 2018-07-14 (4 minutes)
• Sous vide (p. 2921) 2019-04-02 (2 minutes)
• Waterfryer (p. 1462) 2019-04-20 (1 minute)
• Intermittent fluid flow for heat transport (p. 521) 2019-07-10 (4 minutes)
• Methods of pumping ice-vest coolant silently (p. 2415) 2019-09-28 (12 minutes)
• Why you can't run a diesel engine on water and diesel fuel with electrolysis (p. 345) 2019-11-24 (2 minutes)

Notes concerning “The Wercam windowing system”

• A nonscriptable design for the Wercam windowing system (p. 3092) 2018-10-26 (updated 2018-11-13) (6 minutes)
• Scriptable windowing for Wercam (p. 1256) 2018-10-26 (updated 2019-07-24) (26 minutes)

Notes concerning “Wikileaks”

• Cristina Fernández de Kirchner tweets about the attempt to kidnap Assange (p. 985) 2014-04-24 (3 minutes)
• What are Bitcoin’s uses other than sidestepping the law? (p. 2159) 2019-03-11 (updated 2019-07-05) (6 minutes)

Notes concerning “Wikipedia”

- Wikipedia people (p. 948) 2016-06-01 (6 minutes)
- Everything is money? (p. 1859) 2019-08-31 (4 minutes)

Notes concerning “Win32”

- Developing Win32 programs on Debian (p. 609) 2007 to 2009 (12 minutes)
- Win32 startup (p. 2439) 2007 to 2009 (2 minutes)

Notes concerning “Window systems”

• Running your regular desktop in QEMU? (p. 292) 2007 to 2009 (3 minutes)
• How should we design a UI for a new OS? (p. 1159) 2012-10-10 (updated 2012-10-11) (4 minutes)
• How can we usefully cache screen images for incrementalization? (p. 1077) 2013-05-17 (18 minutes)
• Quadtree compression of terminal video RAM to do a megapixel windowing system in 6 KiB (p. 1520) 2013-05-17 (9 minutes)
• More thoughts on powerful primitives for simplified computer systems architecture (p. 1895) 2015-08-18 (updated 2015-11-02) (165 minutes)

Notes concerning “Write-once read-many (WORM) memory”

- Cycle sort (p. 2344) 2013-05-17 (1 minute)
- ISAM designs for Tahoe-LAFS (p. 3199) 2016-09-07 (2 minutes)
- A language whose memory model is a bunch of temporally-indexed logs (p. 1359) 2019-05-12 (updated 2018-05-21) (20 minutes)

Notes concerning “Wrong”

• Steampunk spintronics: magnetoresistive relay logic? (p. 1315) 2013-05-17 (15 minutes)
• Saturation detector (p. 1588) 2013-05-17 (3 minutes)
• Improving LZ77 compression with a RET bytecode (p. 964) 2016-04-05 (updated 2016-04-06) (3 minutes)

Notes concerning “Z machine”

- The Z-machine memory model (p. 2903) 2017-07-19 (4 minutes)
- Constant space flexible data (p. 352) 2018-04-27 (5 minutes)
- Constant space lists (p. 3062) 2018-12-10 (10 minutes)

Notes concerning “Zooming user interfaces (ZUIs)”

• Circle-portal GUI (p. 1151) 2016-06-03 (11 minutes)
• Do visually expanding images evoke an orienting response, or the startle response, and what does that mean for ZUIs? (p. 1805) 2016-06-03 (14 minutes)
• Ideas to explore (p. 2794) 2017-05-29 (updated 2019-09-15) (3 minutes)
• How to make Dercuano work on hand computers? (p. 1371) 2019-05-18 (updated 2019-12-30) (56 minutes)